Linköping Studies in Science and Technology

Thesis No. 1331

Restoring Consistency after Network Partitions

by

Mikael Asplund

Linköping University
INSTITUTE OF TECHNOLOGY

Submitted to Linköping Institute of Technology at Linköping University in partial fulfilment of the requirements for degree of Licentiate of Engineering

Department of Computer and Information Science
Linköping universitet
SE-581 83 Linköping, Sweden

Linköping 2007

Restoring Consistency after Network Partitions

by

Mikael Asplund

October 2007
ISBN 978-91-85895-89-2
Linköping Studies in Science and Technology
Thesis No. 1331
ISSN 0280–7971
LiU–Tek–Lic–2007:40

ABSTRACT

The software industry is facing a great challenge. While systems get more complex and distributed across the world, users are becoming more dependent on their availability. As systems increase in size and complexity so does the risk that some part will fail. Unfortunately, it has proven hard to tackle faults in distributed systems without a rigorous approach. Therefore, it is crucial that the scientific community can provide answers to how distributed computer systems can continue functioning despite faults.

Our contribution in this thesis is regarding a special class of faults which occurs when network links fail in such a way that parts of the network become isolated, such faults are termed network partitions. We consider the problem of how systems that have integrity constraints on data can continue operating in presence of a network partition. Such a system must act optimistically while the network is split and then perform a some kind of reconciliation to restore consistency afterwards.

We have formally described four reconciliation algorithms and proven them correct. The novelty of these algorithms lies in the fact that they can restore consistency after network partitions in a system with integrity constraints and that one of the protocols allows the system to provide service *during* the reconciliation. We have implemented and evaluated the algorithms using simulation and as part of a partition-tolerant CORBA middleware. The results indicate that it pays off to act optimistically and that it is worthwhile to provide service during reconciliation.

This work has been supported by European Community under the FP6 IST project DeDiSys (Dependable Distributed Systems, contract 004152).

Department of Computer and Information Science
Linköping universitet
SE-581 83 Linköping, Sweden

Acknowledgements

First of all I would like to thank my supervisor Simin Nadjm-Tehrani. With a brilliant and challenging mind she has really helped me forward and made me think in ways I couldn't have done by myself. Its been fun as well, with lots of discussions on just about anything.

This work has been financially supported by the European Community under the FP6 IST project DeDiSys. I would very much like to thank all the members of the project. We have had some very fun and interesting times together, and I will never forget the dinner in Slovenia. Special thanks to Stefan Beyer, Klemen Zagar, and Pablo Galdamez with whom a lot of the work in this thesis have been done.

Thanks also to all the past and present members of RTSLAB and my friends at IDA who have made for fun discussions and a nice working environment. There is always someone with a deadline approaching and thus happy to waste an hour or so chatting. A big thanks to Anne Moe who have been able to solve any kind of problem so far.

Thanks to all my friends and family. My wife Ulrika has been very patient with odd ways and I'm deeply grateful for her love and support. Being a PhD student seems to be something that affects the brain irreversibly, and she is the one that has to live with me.

Contents

List of Figures

"Stå
grå,
stå
grå,
stå
grå,
stå
grå,
stå
grå-å-å-å.
Så är gråbergs gråa sång
lå-å-å-å-å-å-ång."

Gustaf Fröding

1 Introduction

The western world has already left the stage of one computer in every home. Nowadays, computers are literally everywhere, and we use them daily for chatting, watching movies, booking tickets and so on. Each of these activities use some service that is accessible through the Internet. So in effect, we have made ourselves dependent on these computer systems being available. Unfortunately, computer systems do not always work as expected.

In fact, we have become used to hearing reports in the media about some computer system that has crashed. Here are a few examples. In august 2004 a system of the British Post Office failed affecting 200 000 customers that could not collect their pensions [9]. In March 2007 Microsoft's live services (including Microsoft Messenger) was unavailable for several days affecting millions of Swedish customers [81]. In February 2007 problems with IT systems of Jetblue Airways caused the company to cancel more than a thousand flights [82].

It goes without saying that such disruptions cause huge financial losses as well as inconvenience for users. For some systems (such as air traffic control) it is also a matter of human safety. Here computer science has an important role to play by reducing the consequences of failing subsystems.

1.1 Motivation

We do not know the exact causes of the failures we mentioned, but as more and more systems become distributed across different geographical locations we believe that communication failures is and will increase to be a cause of computer system failures. Such network failures might seem unlikely for

well-connected networks with a high redundancy of links. However, for geographically dispersed networks it does not take more than some cable being cut off during a construction work, or a malfunctioning router, to make communication between two sites impossible for some period of time.

In addition, more and more computers are becoming connected via wireless networks. There can be several reasons for using wireless networks instead of wired ones. Apart from allowing mobility, it can also be used as a means to reduce infrastructure deployment costs. This could for example be used in traffic surveillance where it would be costly to draw cables to each node, whereas 3G connectivity can be easily achieved. Of course, wireless links are also more likely to fail. This is due to the physical characteristics of the medium like interference and limited signal strength due to obstructions.

Therefore, it is important to find solutions to ensure that computer systems stay operational even when there are failures in the network. Note that a link failure does not automatically mean that communication is impossible. If there are alternative paths to reach the destination node, requests will be rerouted around the failed link and the system can continue operation. However, for most systems there are critical links for which there is no alternative. If such a link fails, the system is said to suffer from a network partition. This is more tricky to deal with.

What we would like to achieve is to allow systems to stay available despite network partitions. However, this requires the system to act optimistically. As an example, consider a city in which the intention is to control a heavy trafficked area to reduce congestion and hazardous pollution. A model is constructed that takes as input cars entering and leaving the area, possible construction sites, accidents, jams etc. A control loop then regulates traffic lights, toll fees and provides information to drivers of alternative routes. To feed the application with information, a number of sensors of different type are positioned around the city. In addition, the system can collect data from cars that are equipped with transponders. To reduce installation costs the nodes are equipped with wireless network devices.

A car entering the system will be detected by multiple sensors. Thus, there must be *integrity constraints* for accepting sensor input so that the traffic model is not fed with duplicate (and conflicting) information. For example, a given car can not enter the system twice without having left it in between. Moreover, the system will not accept data from cars without being able to verify the authenticity at a verification service.

In such a system it would make sense to continue operating during a network partition. The model would continue to make estimations based on partial sensor data. When the network is healed, the system must come back to a consistent state. This is a process to which we refer to as *reconciliation*. However, it may not be an easy task of just merging the states of the partitions. Several operations performed in parallel partitions may invalidate integrity constraints if considered together. This can be alleviated by replaying all the operations (sensor readings) that have taken place during

the degraded mode. In a new (reconstruction of the) state of the healed network, integrity constraints must be valid, and e.g. duplicate sensor readings are thus discarded.

1.2 Problem Formulation

This brings us to the subject of this thesis. We study reconciliation algorithms for systems with eventually strict consistency requirements. These systems can accept consistency to be temporarily violated during a network partition, but require that the system is fully consistent once the system is reconciled. Specifically, we are interested in systems where consistency can be expressed using data integrity constraints as in the car example above.

Our main hypothesis is that network partitions can be effectively tolerated by data-centric applications by using an optimistic approach and to reconcile conflicts afterwards. Moreover, we theorise that this can be done with the help of a general purpose middleware. To support this claim we must explore the possible solution space and answer a number of research questions:

- Does acting optimistically during network partitions pay off, even in presence of integrity constraints?
- Which is preferable, state or operation based reconciliation?
- What can be done to optimise operation based reconciliation?
- Is it possible and/or worthwhile to serve new incoming operations during reconciliation?
- Can such support be integrated as part of a general middleware?

Although this thesis concentrates on the reconciliation part of a partition-tolerant middleware, the work is part of a larger context. In the European DeDiSys project the goal is to create partition-tolerant middleware to increase the availability for applications.

1.3 Contribution

In this thesis we present and analyse several different algorithms that restore consistency after network partitions. The contributions are:

- Formal descriptions of four reconciliation algorithms – three of which are centralised and assume no incoming operations during reconciliation, and one distributed algorithm that allows system availability during reconciliation.

- Proofs that the above algorithms are correct and provision of sufficient conditions for termination.
- New metrics for evaluating performance of systems with optimistic replication and reconciliation.
- An implementation of all the protocols in a simulation environment and of the continuous service protocol as a CORBA component.
- Simulation based performance evaluations of the algorithms as well as performance studies of the CORBA middleware with our fault tolerance extension.

1.4 Publications

The work in this thesis is based on the following publications:

- M. Asplund and S. Nadjm-Tehrani. Post-partition reconciliation protocols for maintaining consistency. In *Proceedings of the 21st ACM/SIGAPP Symposium on Applied Computing (SAC 2006)*, April 2006.
- M. Asplund and S. Nadjm-Tehrani. Formalising reconciliation in partitionable networks with distributed services. In M. Butler, C. Jones, A. Romanovsky, and E. Troubitsyna, editors, *Rigorous Development of Complex Fault-Tolerant Systems*, volume 4157 of *Lecture Notes in Computer Science*, pages 37–58. Springer-Verlag, 2006.
- M. Asplund, S. Nadjm-Tehrani, S. Beyer, and P. Galdamez. Measuring availability in optimistic partition-tolerant systems with data contraints. In *DSN '07: Proceedings of the 2007 International Conference on Dependable Systems and Networks*, IEEE Computer Society, June 2007.

1.5 Outline

The rest of the thesis is organised as follows. Chapter 2 provides the background to our work. In Chapter 3 an overview of our approach is given and the system model is described. The algorithms are described in Chapter 4, and their correctness is shown in Chapter 5. The implementation of the Continuous Service reconciliation protocol is described in Chapter 6. We have experimentally evaluated our solutions and the results are provided in Chapter 7. Finally in Chapter 8 we conclude and give some pointers for future work.

"In the beginning the Universe was created. This has made a lot of people very angry and has been widely regarded as a bad move."

Douglas Adams

2 Background

This chapter lays out the background of our work. We start with dependability and availability concepts as they are the underlying motivation. Then we go through some basic concepts of fault tolerance in distributed systems as they are our building blocks. Next, we consider the problem of consistency as our work deals with fault tolerance in presence of consistency requirements. Finally, we review some alternative solutions for supporting partition tolerance.

2.1 Dependability

The work described in this thesis has one major goal, increasing availability. Thus, it is prudent to give an overview of definitions for availability and put it in a context with other related concepts such as reliability. In this section we put the availability concept in a wider perspective. Availability alone is not enough to capture the requirements on for example the control system in a car. We also need the system to be reliable and safe. These concepts are covered by the idea of dependability. In 1980 a special working group of IFIP (IFIP WG 10.4) was formed to find methods and approaches to create dependable systems. Their work contributed to finding a common terminology for dependable computing. We will outline some of the basic ideas that are summarised by Avižienis et al. [4].

Dependability is defined as the ability to deliver service that can justifiably be trusted. A more precise definition is also given as the ability of a system to avoid failures that are more frequent or more sever and outage durations that are longer than is acceptable to the user(s).

The attributes of dependability are the following [4]:

- availability: readiness for correct service,
- reliability: continuity of correct service,
- safety: absence of catastrophic consequences on the user(s) and the environment,
- confidentiality: absence of unauthorised disclosure of information,
- integrity: absence of improper system state alterations,
- maintainability: ability to undergo repairs and modifications.

These attributes are all necessary for a dependable system. However, for some systems, they come at no cost. For example, a system for booking tickets is inherently safe since it cannot do any harm even if it malfunctions. Which attributes that are more important to fulfil varies as well and so do the interdependencies. Safety of the system may very well be depending on the system's availability.

The difference between availability and reliability is worth pointing out. Availability is related to the proportion of time that a system is operational whereas reliability is related to the length of operational periods. For example, a service that goes down for one second every fifteen minutes provides reasonable availability (99.9%) but very low reliability.

Security is sometimes given as an attribute but could also be seen as a combination of availability, confidentiality and integrity. We have already stated the importance of availability and it is clear that it is a fundamental attribute of any system although the required degree of availability might vary.

2.1.1 Measuring Availability

When dealing with metrics relating to fault tolerance, we are usually dealing with unknowns. We cannot know a priori what types of faults will occur, or when they occur. Therefore, we cannot know what level of quality we can provide for a given service. However, we can still quantify the availability in two ways. First, we can do a probabilistic analysis and calculate a theoretical value for the system availability. The second option is to actually measure the availability for a number of scenarios. Thus, there are two types of metrics that are discussed here, theoretical metrics, and experimental metrics. They can be seen as related in the same way as the mean and variance of a stochastic variable can be estimated using sampling.

Helal et al. [49] summarise some of the definitions for availability. The simplest measure of (instantaneous) availability is the probability $A(t)$, that the system is operational at an arbitrary point in time. The authors make

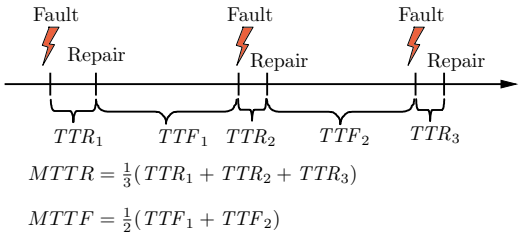

Figure 2.1: Time to failure and repair

the definition a bit more precise by saying that for the system to be operational at time t there are two cases. The system has been functioning properly in the interval $[0, t]$ or the system has failed at some time point but been repaired before time t. As this measure can be rather hard to calculate the authors give an alternative definitions as well.

The limiting availability $\lim_{t\to\infty} A(t)$ is defined as follows:

$$\lim_{t\to\infty} A(t) = \frac{MTTF}{MTTF + MTTR}$$

where $MTTF$ is the mean time to failure and $MTTR$ is the mean time to repair. These are illustrated in Figure 2.1.

Both of these measures are based on theoretical properties of the system. However, in order to effectively evaluate a system, we need a metric that can be measured. Helal et al. give the following general expression for experimentally measuring availability given that a system is monitored over a time period $[0, t]$.

$$A(t) = \frac{\sum_i u_i}{t}$$

Here, u_i are the periods of time where the system is available for operation. If we analyse the behaviour if fault-tolerant systems, Sieworek and Swarz [85] identified up to eight stages in response to a failure that a system may go through. These are: fault confinement, fault detection, diagnosis, reconfiguration, recovery, restart, repair, and reintegration.

The above definitions are very general and even ambiguous. Specifically the term "operational" needs to be clarified. This is done in Section 7.1 in this thesis, giving rise to the terms partially operational and apparently operational. A different approach to measuring availability, that we also employ in this work, is based on counting successful operations [98]. This allows differentiation between different types of operations. Coan et al. [22] defines an availability metric for partitioned networks. The basic idea is to let

$$Availability = \frac{\text{number of transaction successfully completed}}{\text{number of transactions presented to the system}}$$

This can then be divided in two parts, availability given by performing update transactions, and availability given by performing read transactions. This is a measurable metric that naturally has its corresponding theoretical measure [50] which is the probability that a given transaction succeeds.

There is a major problem with this metric when combined with integrity constraints. If an operation or transaction is rejected due to an integrity constraint, then this is part of the normal behaviour of the system and should not be seen as reduction of availability. On the other hand, if we consider rejected operations as successfully completed then we have another problem. If we optimistically accept an operation without being able to check for consistency and revoke it later, should it still be counted as successfully completed?

An alternative way of measuring availability is proposed by Fox and Brewer [38]. They define two metrics, *yield* which is the probability of completing a request and *harvest* which measures the fraction of the data reflected in the response. This approach is best suited for reasoning about availability for read-operations. In this thesis we concentrate on the availability for update operations.

2.1.2 Dependability threats

The taxonomy for dependability also defines the threats to dependability:

- fault: the cause of an error
- error: part of the system state that may cause a subsequent failure
- failure: an event that occurs when the delivered service deviates from correct service

There is a chain of causality between faults, error and failures. A fault can be dormant and thus cause no problem, but when it becomes active it causes an error. This error may propagate and result in more errors in the system. If the error is not treated, e.g. by means of fault tolerance, then this error may propagates to the service interface and thus becomes a failure. If the system is part of a larger system then this failure becomes a *fault* in that system. To summarise we have the following sequence: fault → error 1 → ... → error n → failure → fault → .... Unfortunately, in distributed systems faults and failures are often used interchangeably as a node crash is a failure from the node point of view but a fault from the network point of view.

2.1.3 Dealing with faults

There are four basic approaches [4] for dealing with faults and thereby achieving dependability:

- fault prevention: to prevent the occurrence or introduction of faults,
- fault tolerance: to deliver correct service in the presence of faults,
- fault removal: to reduce the number or severity of faults,
- fault forecasting: to estimate the present number, the future incidence, and the likely consequences of faults.

All of these techniques are necessary for creating highly available services. It would be foolish to rely on just one of them. However, fault tolerance may be the most important property for a system to have since there is no way of completely avoiding faults. This has also been given the most attention in the research community. The techniques described and analysed in this thesis are designed to achieve fault tolerance. In the following section we will elaborate on existing techniques for fault tolerance in distributed systems.

2.2 Fault Tolerance in Distributed Systems

Fault tolerance is still a young research discipline and we have yet to see a common terminology that is widely accepted as well as unambiguous. However, a lot has been done [25, 83, 61, 40, 90] towards creating a common understanding of the concepts.

2.2.1 Fault models

For any fault-tolerant approach to be meaningful it is necessary to specify what faults the system is capable of tolerating. No system will tolerate all kinds of faults, e.g. if all nodes permanently crash. So what we need is a model that describes the allowed set of faults. Sometimes this is referred to as a failure models, which is then a model of how a given component may *fail*. Here we take the system perspective and consider the failure of a component as a fault in the system, therefore we use the term fault model rather than failure model.

When discussing distributed systems, we usually refer to nodes (i.e., computers) and links. However, one can use the more general concept of processes. A process can send and receive messages. Thus, both links and nodes can be seen as special instances of processes. Typical fault models for distributed systems are the following.

- Fail-Stop: A process stops executing and this fact is easily detected by other processors.

- Receive Omission: A process fails by receiving only a subset of the messages that have been sent to it.
- Send Omission: A process fails by only transmitting a subset of the messages that it attempts to send.
- General Omission: Send and/or Receive Omission.
- (Halting) Crash: Special case of General Omission where once messages are dropped they are always dropped.
- Byzantine: Also called arbitrary as the processor may fail in any way such as creating spurious messages or altering messages.

In natural language we sometimes say that a computer crashes when it suffers from a transient fault. After rebooting the computer it can continue functioning as a part of the network. This does not match the above definition of crash failure. The same goes for a faulty link, that can be repaired. To clear this ambiguity we can therefore differentiate between [40]:

- halting crash: a crashed process never restarts,
- pause crash: a crashed process restarts with the same state as before the crash,
- amnesia crash: a crashed process restarts with a predefined initial state which is independent on the messages seen before the crash,

Note that for stateless processes (such as links) pause and amnesia are equivalent. The real challenge in creating fault-tolerant systems is when multiple faults happen simultaneously. Especially of interest in this thesis is the situation where one or more links fail (general omission/pause crash) in such a way that the network is divided in two or more disjoint partitions. This is called a *network partition fault*.

We follow the practice of the distributed computing community and use the term "partition" to refer to an isolated part of the network. This does not correspond to the mathematical definition of a partition which refers to a set of disjoint sets.

2.2.2 Timing models

Time plays a crucial role in the design of distributed systems. The very notion of fault tolerance is practically unachievable without some assumptions on timing. There are three basic timing models [61]:

- the synchronous model,
- the asynchronous model,

- the partially synchronous model.

Each model translates to a set of assumptions on the underlying network. The synchronous model is the most restrictive model, as it requires that there is a bound on the message delivery delay and on the relative execution speeds of all processes. Although it is possible to create systems that fulfil this requirement, it is very expensive. However, since the model allows easy support for fault tolerance and verification it is a relevant model for safety critical systems.

The asynchronous model does not imply any timing requirements whatsoever. That is, we cannot bound the message delivery delay or related the rate of execution for the nodes. However, most algorithms require liveness, meaning that the delays are not infinite (but can be arbitrarily long). This model is very attractive since an algorithm that works in an asynchronous setting will work in all possible timing conditions. However, as we will come back to, fault tolerance is hard to achieve in asynchronous networks.

This leaves us with the partially synchronous model [34] where some timing requirements are made on the network but that does not require clock synchronisation. A variant to this is the timed asynchronous model of Cristian and Fetzer[26] that requires time bounds but allows for an unbounded rate of dropped or late messages. Another variation based on the above models is a system that is partially synchronous in the sense that it acts as a synchronous system during most intervals but there are bounded intervals during which the system is acting in an asynchronous fashion. [19]

2.2.3 Consensus

At the very heart of the problems relating to fault tolerance in distributed systems is the consensus problem [37]. It has been formulated in a nice fashion by Chandra and Toueg [18] over a set of processes:

Termination Every correct process eventually decides some value.

Uniform integrity Every process decides at most once.

Agreement No two correct processes decide differently.

Uniform validity If a process decides v, then v was proposed by some process.

Although this problem sounds simple enough, Fischer, Lynch and Paterson showed in [37] that solving consensus with a deterministic algorithm is impossible in a completely asynchronous system if just one process may crash. The intuition behind this result is that it is impossible to for the participating nodes to distinguish a crashed process from a very slow process. So either the system has to wait for all nodes (i.e., forever if one node has crashed) or it must decide even if one node does not respond. However, a very slow node could have decided differently in the mean time.

Unfortunately for a number of problems it has been shown that they are equivalent or harder than the consensus problem. Among these are atomic multicast/broadcast (equivalent [18]) and non-blocking atomic commit (harder [45]).

So if we cannot solve consensus in completely asynchronous systems, what do we have to assume to make consensus possible? There are several answers to this question. Dolev et al. [32] showed what the minimal synchronicity requirements are for achieving consensus in presence of crashed nodes. For example, it suffices to have broadcast transmissions and synchronous message order. However, for consensus to be possible there has to be a bound on both delivery time and difference in processor speeds.

Note that although these bounds need to exist as shown by Dwork and Lynch [34] it is not necessary that these bounds are known. This is important since it allows the creation of consensus algorithms without having to know the exact timing characteristics of the system. It is enough to know that there are bounds, they do not have to be encoded in the algorithm. Alternatively, the bound may be known but it does not hold until after some time t that is unknown.

Another way to solve the consensus problem is to allow probabilistic protocols [10]. Unfortunately, this approach seems to lack efficiency [2].

2.2.4 Failure Detectors

In 1996 Chandra and Toueg [18] presented a concept that has been somewhat of a breakthrough in fault-tolerant computing. By introducing *unreliable failure detectors* they created an abstraction layer that allows consensus protocols to be created without having to adopt a synchronous timing model. Of course, in order to implement such failure detectors, one still needs at least partially synchronous semantics.

The failure detectors are classified according to properties of completeness and accuracy. Completeness means that the crashed process is suspected, whereas accuracy means that a live node is not suspected. The question of who suspects is answered by the difference between strong and weak completeness. Strong completeness means that all nodes must suspect the crashed node and weak completeness that *some* node suspects it. Strong accuracy means that no correct process is suspected (by anyone), and weak accuracy means that some correct process is never suspected (by anyone). Finally, the failure detector can be eventually strong (weak) accurate meaning that there is a time after which no (a) correct process is (not) suspected, but before which correct processes may be suspected. This is the reason for the term unreliable failure detectors, they may suspect healthy processes as long as they eventually change their mind.

This categorisation gives rise to eight types of failure detectors. Chandra and Toueg [18] show that in a system with only node crashes, weak completeness can be reduced to strong completeness and therefore there are

actually only four different classes. They also show that these four classes are really distinct and cannot be mutually reduced.

Since the work of Chandra and Toueg only applies to systems with node crashes their work has been extended [33, 7] to include link failures and thus partitions. In systems with link failure, weak and strong completeness are no longer equivalent. Therefore one needs strong completeness and eventually strong accuracy to solve consensus.

There has been considerable research on how to make failure detectors efficient and scalable [93, 46, 20, 48]. Unfortunately, this problem is inherently difficult. In order to achieve scalability, one needs to allow longer time periods between the occurrence of a fault and its detection. This leads to slower consensus since the consensus algorithms will wait for a message from a node or an indication from the failure detector.

2.2.5 Group communication and group membership

Although failure detectors supply an important abstraction level, they do not provide high level support for fault tolerance. What the application programmer really needs is a service that provides information of the reachable nodes as well as atomic multicast primitives. This is the reason for group membership and group communication. Group membership provide the involved processes with views. Each view tells what other processes are currently reachable. On top of this it is possible to create a group communication service that also provides (potentially ordered) multicasts within the group.

The first group membership service was the ISIS [15] system (later replaced by Horus [92]) which was developed at Cornell. In 1987 Birman and Joseph [14] introduced the concept of virtual synchrony that were adopted in the ISIS system. Virtual synchrony states that for any two processes that are members of the view v_2 and which previously were both members of the view v_1 agree on the set of messages that were delivered in v_1. Thus creating the impression for all observers that messages have been delivered synchronously. Both the mentioned formalism and the ISIS implementation were restricted to only allowing a primary partition to continue in case of a network partition fault.

A number of different formalisms [66, 39, 7] and systems have later been proposed with some differences in their specifications. The differences are mainly of three types: what is the underlying services required, when are messages allowed to be sent, and finally what are the message delivery guarantees. Earlier services sometimes rely on hidden assumptions on the network or specifically assume timing properties [27]. Later, the failure detector approach has been used to separate the lower level time-out mechanisms. The second difference relates to whether the service allows messages to be sent during regrouping intervals (such messages cannot be delivered with the same guarantees as ordinary messages). The message delivery guar-

antees range from FIFO multicast to atomic multicast. Moreover, some membership services only guarantee same view delivery (i.e., a message is at the same view for all receiving processes) whereas others guarantee sending view delivery [66] (i.e., the message is delivered in the same view as it is sent).

Transis [3] was the first system that allowed partitions to continue with independent groups. It has been followed by several other such as Totem [67], Moshe [55], Relacs [6], Jgroups [7], Newtop [35], and RMP [54].

For a comprehensive comparison of different group communication services the reader we recommend the paper by Chockler et al. [21].

2.2.6 Fault-tolerant middleware

Since providing fault tolerance is a non-trivial task, requiring efficient algorithms for failure detection, group membership, replication, recovery, etc, it seems to require a lot of effort to produce a fault-tolerant application. Moreover, application writers apparently need to be knowledgeable about these things. However, the complexity of these tasks have contributed to the development of *middleware* that takes care of such matters in a way that is more or less transparent to the application writer. Middlewares first appeared as a way to relieve the complexity of remote invocations by allowing remote procedure calls or remote method invocations.

Here we will briefly describe some of the work that has been done in crash-tolerant middleware. Later in Section 2.4.4 we will relate to partition-tolerant middleware systems.

The CORBA platform has been a popular platform for research on fault-tolerant middleware. Felber and Narasimhan [36] survey some different approaches to making CORBA fault-tolerant. There are basically three ways to enable fault tolerance in CORBA, (1) by integrating fault tolerance in the Object Request Broker (ORB), examples include the Electra [62] and Maestro [94] systems (2) by introducing a separate service as done in DOORS [71] and AqUA [28] (3) by using interceptors and redirecting calls. The interceptor approach is taken in the Eternal system [70] which is also designed to be partition-tolerant (discussed below).

The fault-tolerant CORBA (FT-CORBA) specification [72] was the first major specification of a fault-tolerant middleware. The standard contains interfaces for fault detection and notification. Moreover, several different replication styles are allowed such as warm and cold passive and active replication. Two of the systems discussed above (Eternal and Doors) fulfils the FT-CORBA standard. Unfortunately, the FT-CORBA standard also has some limitations.

There has also been some work done in combining fault tolerance and real-time requirements in middleware. MEAD [69] is a middleware based on RT-CORBA with added fault tolerance. The idea is to adaptively tune fault tolerance parameters such as replication strategy and checkpointing

frequency according to online measurements of system performance. Moreover, there is support for predicting faults that follow a certain pattern and start mitigating actions before the fault actually occurs. The faults considered are crash, communication and timing faults, but not partition faults. The ARMADA project [1] aimed at constructing a middleware for real-time with support for fault tolerance. It allows QoS guarantees for communication and service. It has support for fault tolerance mechanisms using a real-time group communication component, but does not consider partition faults. Huang-Ming and Gill [52] have extended the real-time Ace ORB (TAO) with improvements of the CORBA replication styles so that fault tolerance is achieved for a real-time event service.

Szentivanyi and Nadjm-Tehrani [89, 88] implemented the FT-CORBA standard as well as an alternative approach called fully available CORBA (FA-CORBA) and compared their fault-tolerant properties and overhead performance. The drawback with the FT-CORBA standard is that some of the required services (replication manager and fault notifier) are single points of failure. The FA-CORBA implementation does not have any single points of failure, but requires a majority of nodes to be alive. In terms of overhead the FA-CORBA implementation was more costly due to the fact that it requires a consensus round for each invocation.

2.3 Consistency

As suggested by Fox and Brewer [38], and formally shown (although with heavy restrictions) by Gilbert and Lynch [41] it is impossible to combine consistency, availability and network partitions. This is called the CAP principle by Fox and Brewer. Thus, our goal is to temporarily relax consistency and thereby increase availability.

To explain the concept of consistency in a very general sense, one can say that to ensure consistency is to follow a set of predefined rules. Thus, all computer systems have consistency requirements. Some of the rules are fundamental: object references should point to valid object instances, an IP packet must have a header according to RFC791 etc. These requirements are (or should be) easy to adhere to, and it would not make any sense to brake them. However, there are also consistency requirements that are defined to make it easier to deliver correct service to the user; but which are not crucial. Instead, by relaxing some of these requirements, we can expand the operational space for our system. However, since the rules were created to ease the system design, relaxing them requires more algorithms the deal with cases when the rules are broken. We will continue by looking at three types of consistency requirements that can be traded for increased availability: replica consistency, ordering constraints, and integrity constraints. These are related to the consistency metrics by Yu and Vahdat [97] which we will refer to in our discussion below. However, we will not elaborate on any real-time requirements such as the staleness concept of Yu and Vahdat.

Note the difference between the consistency requirements discussed here and the consensus problem above. Consensus is a basic primitive for cooperating processes to agree on a value. Consistency is a higher level concept that constrains data items and operations. We need consensus to ensure consistency.

2.3.1 Replica consistency

First we give an informal definition of replica consistency: A system is replica consistent if for all observers interacting with the replicas of a given object, it appears as if there is only one replica. Any optimistic replication protocol will need to violate replica consistency at least temporarily.

In databases there exists several concepts that imply replica consistency, but are wider in the sense that they do not only restrict the state of the replicas but also the order in which subtransactions are processed. The most frequently used consistency concept is one-copy-serializability [11]. This does not only stipulate that the replicas all appear as one, it also requires that concurrent transactions are isolated from each other. That is, the transaction schedule is equivalent to a schedule where one transaction comes after the other.

This requirement has been found to be too strict for many systems due to other requirements such as performance and fault tolerance. As a solution to this Pu and Leff [78] introduced epsilon serializability (ESR). This allows asynchronous update propagations while still guaranteeing replica convergence. This means that ESR guarantees 1-copy serializability *eventually*. Four control methods are described that utilise information such as commutativity to ensure eventual consistency.

For single master systems (only one replica is allowed to be updated), one can characterise the amount of replica inconsistency by staleness or freshness [24]. If updates occur with an even load, then freshness can be quantified simply by measuring the time since the replica was last updated.

2.3.2 Ordering constraints

Anyone dealing with distributed systems sooner or later runs in to the problem of ordering. There are two types of ordering constraints: syntactic and semantic.

Syntactic ordering Syntactic ordering is completely independent of the application logics. However it may take into account what data elements were read or written to, or the time point of a certain event.

The causal order relationship introduced by Lamport [60] has proved to be very useful when ordering events in distributed systems. For example, if a multicast service guarantees causal order, then no observer will receive

messages in the wrong order. It is also possible to create consistent snapshots [43] of a system by keeping track of the causal order of messages.

The order error of Yu and Vahdat is a combination of replica consistency and ordering constraints. They define every operation that has been applied locally but not in an ideal history (or applied in the ideal history but in the wrong order) as increasing order error. This means that requiring the order error to be zero is equivalent to requiring causal order and full replica consistency.

Semantic ordering Semantic ordering on the other hand, is an order that is derived from the application requirements. These requirements may be explicit or implicit.

Shapiro et al. [84] have proposed a formalism for partial replication based on two explicit ordering constraints, before and must-have. The before constraints states that if operation α is before operation β then α must be executed before β. The must-have constraint states that if α must-have β then α cannot be executed unless β is also executed. These were used as the basis for the IceCube [77] algorithm, and they allow many ordering properties to be expressed in a concise and clear manner. However, as the authors remark at the end, these constraints cannot capture the full semantics for some applications such as a shared bank account.

Integrity constraints that are discussed below also induce ordering constraints, but implicitly.

2.3.3 Integrity constraints

Integrity constraints have been used in databases for a long time [42]. They are useful for limiting the possible data space to exclude unreasonable data and thereby catching erroneous input or faulty executions. Application programmers use integrity constraints all the time when programming applications, but they are usually written as part of the logic of the application rather than as explicit entities. This might start to change if or when the design-by-contract [64] methodology becomes more popular.

There are three basic types of constraints used in the design-by-contract philosophy:

- *preconditions* are checked before an operation is executed,
- *postconditions* are checked after an operation is executed,
- *invariants* must be satisfied at all times.

Invariants can be maintained using postconditions provided that all data access is performed through operations with appropriate postconditions.

Yu and Vahdat [97] use numerical error as a way to quantify the level of consistency of the data. This should work well for applications with numerical data constraints. However, in most cases constraints are either fulfilled or not.

2.4 Partition tolerance

Now that we have covered the basic ideas relating to fault tolerance and consistency we can focus on the problem of this thesis. What are the possible ways to support partition tolerance in distributed applications with integrity constraints? We start by looking at what can be done to limit the inconsistencies created by providing service in different partitions. For example, imagine that we have a booking system of four nodes and a given performance has 100 tickets. If this system splits in two parts with two nodes in each partition, then each partition can safely book 50 tickets without risking any double bookings. Unfortunately, it may be the case that the majority of bookings occur in one of the partitions. In such a scenario, there will be many unbooked seats and customers having been refused to book a seat.

2.4.1 Limiting Inconsistency

The first step to limiting inconsistency is to be able quantify it, this we discussed in Section 2.3. The second step is to be able to estimate it. This seems to be quite hard. Yu and Vahdat [97] have constructed a prototype system called TACT with which they have succeeded in showing some performance benefits.

Assume that a system designer decides that a given object can optimistically accept N updates for all its replicas. More than this would put the system in a too inconsistent state. Zhang and Zhang [99] call this an update window. Based on this limit they construct an algorithm that divides the number of allowed optimistic updates among the replicas based on latency and update rate. Although this allows for an increase in availability and performance when the network is slow or unreliable it does not cope well with long running network partitions. In a partitioned system the replicas can only accept a limited number of updates before having to start rejecting updates.

Krishnamurthy et al. [59] introduced a quality of service (QoS) model for trading consistency against availability in a real-time context. They have built a framework for supplying a set of active replicas to the querying client. Using a probabilistic model of staleness in replicas they are able to meet the QoS demands without overloading the network.

No special consideration is made regarding failures in the system. The QoS levels are met with a maximum one failed node. However the system is not able to deal with partition failure.

2.4.2 State and operation-based reconciliation

Assume that we have a system that has been partitioned for some time and that update operations have been performed in each of the partitions. When the network heals and the system should come back to one consistent

state we have to perform reconciliation. There are two ways to perform the reconciliation, either by considering the state of each partition, or by considering the operations that have been performed.

Both approaches have benefits and drawbacks. Some of them are related to the problem of just transferring the data. Here parameters such as the size of the data is important as well as the time taken to process operations. Kemme et al, [56] investigates different techniques for transferring data during reconfiguration between replicas in a distributed database. The proposed protocols consider what data must be transferred by using information such as version numbers. Although some of the protocols are targeted at improving recovery and availability the premises are that of a single partition functioning in a partitioned system.

State reconciliation is an attractive option when the state can be merged such as when the state is represented using sets [65]. A directory service application is a good example where set-based reconciliation could be used. However, in most cases it is hard to merge the state without knowing what has been done to change the state. Of course, it is possible to discard the data that has been written in all partitions except one. Alternatively if there is some mechanism to solve arbitrary conflicts (or if there are no conflicts) the state can be constructed by combining parts of the state from each partition. This approach is used for some distributed file systems such as Coda [58] and source management systems such as CVS [16] and Subversion [23]. In these systems, user interaction is required to merge the conflicts. However, anyone who has tried to merge two conflicting CVS checkouts know that this can quickly become tedious and difficult even when the changes seem to be non-conflicting.

There are systems that try to combine state and operation-based reconciliation. The Eternal [70] system is an example of this. For each object there is one replica that is considered as primary. Upon reconciliation the state of these replicas are transferred to the others. Then the operations that have been performed in the non-primary replicas are propagated.

2.4.3 Operation Replay Ordering

An operation-based reconciliation algorithm must have a policy what order to replay the operations in. There are two reasons for having ordering constraints on the operations. First of all, there might be an expected order of replay. Such ordering requirements usually only relate operations that have been performed in the same partition (there are exceptions), and they are usually syntactic requirements such as causal order or time-stamp order.

The second reason for ordering operations is to ensure constraint consistency. This has been widely used in database systems [30]. Davidson [29] presents an optimistic protocol that can handle concurrent updates to the same object in different partitions. Each transaction can be associated with a write-set and a read-set. The write-set contains all data-items that have

been written to and the read-set contains all data-item that have been read. Any transaction that reads a value in one partition should be performed before a transaction that writes to that value in another partition. This constraint together with the normal precedence requirements obtained within a partition gives rise to a graph. It is shown that the graph represents a serializable schedule iff the graph is acyclic. An acyclic graph is achieved by revoking transactions (backout). It is shown that finding the least possible set of backouts needed for creating an acyclic graph is NP-complete. The ordering here is clearly syntactic.

Phatak and Nath [75] follow the same approach. However, instead of requiring serializable final schedules, their algorithm provides snapshot isolation which is weaker. Basically, this means that each transaction gets its own view of the data. This can lead to violation of integrity constraints in some cases.

In the IceCube system [57, 77] the ordering is given by the application programmer as methods that state the preferred order between operations as well other semantic information such as commutativity. Ordering is considered in the sense that some operation histories are considered safe and some unsafe. An example of an unsafe ordering is a delete operation followed by some other operation (read/write). This semantic information is represented as constraints which in turn create dependencies on operations. Due to the complexity of finding a suitable order the authors propose a heuristic for choosing the log schedule.

Martins et al. [63] have designed a fully distributed version of the IceCube algorithm called the Distributed Semantic Reconciliation (DSR) algorithm. The motivation was to include it in a peer-to-peer data management system. In such a system a centralised algorithm would not be feasible. Performance analysis of the DSR algorithm showed that there was not a major improvement in reconciliation speed compared to a centralised algorithm. This is reasonable since there are many interdependencies that needs to maintained in a distributed reconciliation algorithm.

2.4.4 Partition-tolerant Middleware Systems

There are a number of systems that have been designed to provide partition-tolerance. Some are best suited for systems where communication fails only for short periods. None of the systems have proper support for system-wide integrity constraints.

Bayou [91, 74] is a distributed storage system that is adapted for mobile environments. It allows to provisionally (tentatively) accept updates in a partitioned system and to finally accept (commit) them at a later stage. Reconciliation is done on a pair-wise basis where two replicas exchange the tentative writes that have been performed. For each replica there is a primary that is responsible for committing tentative writes.

There is support for integrity constraints through preconditions that are

checked when performing a write. If the precondition fails, a special merge procedure that needs to be associated with every write is performed. Although the system in principle allows constraints to include multiple objects, there is little support for such constraints in the replication and reconciliation protocols. Specifically, the constraints are checked upon performing a tentative write only. Therefore when a primary server commits the operations, they will have been validated on tentative data. Moreover, Bayou cannot deal with critical constraints that are not allowed to be violated.

Bayou will always operate with tentative writes. Even if the connectivity is restored, there is no guarantee that the set of tentative updates and committed updates will converge under continued load. Naturally, if all clients stop performing writes, then the system will converge.

The Eternal system by Moser et al. [68] is a partition-aware CORBA middleware. Eternal relies on Totem for totally ordered multicast and has support both for active and passive replication schemes. The reconciliation scheme is described in [70]. The idea is to keep a sort of primary for each object that is located in only one partition. The state of these primaries are transferred to the secondaries on reunification. In addition, operations which are performed on the secondaries during degraded mode are reapplied during the reconciliation phase. The problem with this approach is that it cannot be combined with constraints. The reason is that one cannot assume that the state which is achieved by combining the primaries for each object will result in a consistent state on which operations can be applied.

Singh et al. [86] describe an integration of load balancing and fault tolerance for CORBA. Specific implementation issues are discussed. They use Eternal for FT and TAO for load balancing. It seems that the replication and load balancing more or less cancel each other out in terms of effect on throughput. Only crash failures are considered.

Jgroups [5] is a system of supporting programming partition aware systems using a method that the authors call enriched view synchrony. Apart from ordinary views of a system the enriched view synchrony model includes sub-views and sets of sub-views. These views can then be used to empower the application to deal with partitions. The authors consider what they term the shared state problem which they characterise as three sub problems. First, there is state transfer which has to be done to propagate changes to stale (non-updated) replicas. Second, there is the state creation problem which arises when no node has an up-to-date state. Finally, there is the state merging problem which occurs when partitions have been servicing updates concurrently.

Holliday et al. [51] propose a framework that allows mobile databases to sign off a portion of the main database for disconnected updates. Their focus is on pessimistic methods because it is made sure that inconsistencies are not allowed to occur. The optimistic approach is briefly mentioned but not investigated in detail.

2.4.5 Databases and File Systems

Pitoura et al. [76] propose a replication strategy for weakly connected systems. In such systems nodes are not disconnected for longer periods as in a network partition, but bandwidth may be low and latency high between clusters of well connected nodes. Such clusters are akin to a partition in the sense that local copies of remote data are kept to achieve high availability. In addition to the normal read and write operations the authors introduce weak writes (which only update locally and do not require system wide consistency) and weak reads (which read from local copies). The effect of weak writes may be revoked as a consequence of a reconciliation process between clusters. Reconciliation is performed in a syntactic way making sure that there is an acceptable ordering of operations according to the defined correctness criteria. However it is assumed that the reconciled operations never conflict with each other requiring the type of reconciliation discussed in this thesis.

Gray et al. [44] address the problem of update anywhere. The authors calculate the number of deadlocks or reconciliations that are needed as the systems scale given syntactic ordering requirements. Particularly mobile nodes suffer from this and the authors suggest a solution. The idea is that a mobile node keeps tentative operations and its own version of the database. When the node connects to the base network the tentative operations are submitted. They are either accepted or rejected. The focus is on serializability but the same reasoning could be applied to constraint checking.

Several replicated file systems exist that deal with reconciliation in some way [80, 58]. Balasubramaniam and Pierce [8] specify a set of formal requirements on a file synchroniser. These are used to construct a simple state-based synchronisation algorithm. Ramsey and Csirmaz [79] present an algebra for operations on file systems. Reconciliation can then be performed on operation level and the possible reorderings of operations can be calculated.

"Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe."

Lewis Carrol

3 Overview and System Model

To explain our work, we need first to explain the basics of our approach. We will give an overview of what kind of system we are considering and how availability can be maintained in such systems. We will then go through the formal terminology that we use in this thesis. This is used when describing the basic timing and fault assumptions that we need as well as explaining what types of ordering and consistency semantics that we assume the system to have.

3.1 Overview

Consider the simple system depicted in Figure 3.1. There are four server nodes hosting three replicated objects. Each object is represented as a polygon with a primary replica marked with bold lines. There are also two clients C1 and C2 that perform invocations to the system.

A link failure as indicated in the picture with a lightning will result in a network partition. At first glance, this is not really a problem. The system is fully replicated so all objects are available at all server nodes. Therefore client requests could in principle continue as if nothing happened. This is true for read operations, where the only problem is the risk of reading stale data. The situation with write operations is more complicated. If client C1 and C2 both try to perform a write operation on the square object then there is a write-write conflict when the network is repaired. Even if clients are only restricted to writing to primary replicas (i.e., C2 is only allowed to update the triangle) there is the problem of integrity constraints. If there is some constraint relating the square and the triangle, C2 cannot safely

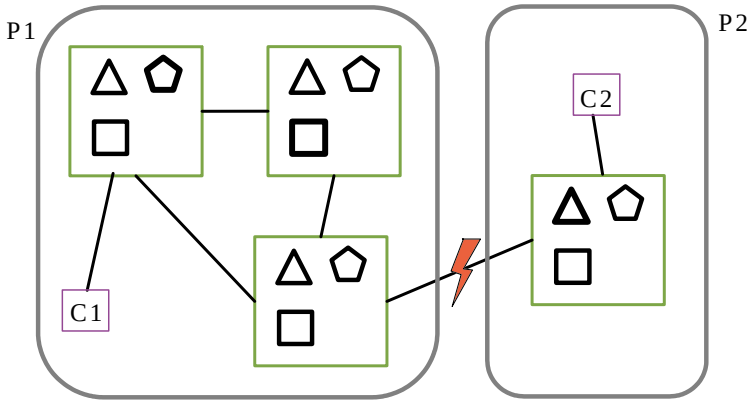

Figure 3.1: System Overview

perform a write operation since the integrity constraint might be violated.

The most common solution to this problem is to deny service to the clients until the system is reunified (shown in Figure 3.2. This is a safe solution without any complications, and the system can continue immediately after the network reunifies. However, it will result in a low availability.

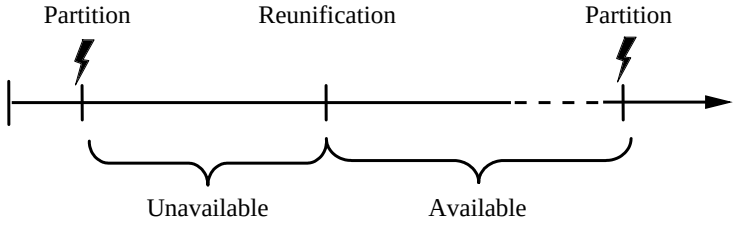

Figure 3.2: Pessimistic approach: No service during partitions

The second approach of letting a majority partition [49] continue operating is better. In our example this would allow at least partial availability in the sense that C1 gets serviced whereas C2 is denied service. However, there are two problems associated with this solution. First, of all there might not be a primary partition. If the nodes in our example were split so that there were two nodes in each partition, then none of the partitions would be allowed to continue. Secondly, it does not allow prioritising between clients or requests. It might be the case that it is critical to service C2 whereas C1's operations are not as important. Such requirements cannot be fulfilled using the majority partition approach.

This leaves us with the optimistic approach where all partitions are allowed to continue accepting update operations. We say that the system operates in *degraded mode*. However, we still have the problems of replica conflicts and integrity constraint violations. This is why we need reconcilia-

tion algorithms to solve the conflicts and the install a consistent state in all nodes. In other words, we temporarily relax the consistency requirements but restore full consistency later.

Unfortunately, there are disadvantages with acting optimistically as well. First of all, there might be side-effects associated with an operation so that the operation cannot be undone. There are two ways to tackle this problem. Either, these operations are not allowed during degraded mode, or the application writer must supply compensating actions for such operations. A typical example of this is billing. Some systems allow users to perform transactions off-line without being able to verify the amount of money on the user's account. A compensating action in this case is to send a bill to the user.

The second problem is that there might be integrity constraints that cannot be allowed to be violated, not even temporarily. This usually means that there is some side effect associated with the operation that cannot be compensated (e.g., missile launch). To deal with this we differentiate between critical and non-critical constraints. During network partitions, only operations that affect non-critical constraints are allowed to continue. Therefore, the system is only partially available.

Since this reconciliation procedure involves revoking or compensating some of the conflicting operations, most constraint-aware reconciliation protocols need to stop incoming requests in order not to create confusion for the clients. We call this type of reconciliation algorithms stop-the-world algorithms since everything needs to be stopped during reconciliation (see Figure 3.3).

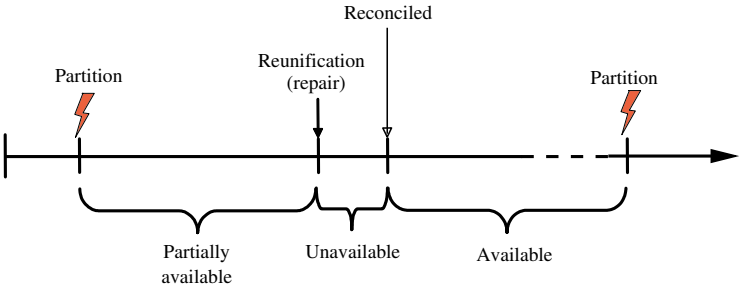

Figure 3.3: Optimistic stop-the-world approach: partial service during partition, unavailable during reconciliation

In addition to the stop-the-world protocols, we have also constructed a reconciliation protocol that we call the Continuous Service (CS) reconciliation protocol. This protocol allows incoming requests during reconciliation (see Figure 3.4). The cost for this is that the reconciliation period gets slightly longer, and that the service guarantees are the same as during the period of network partition.

Although this thesis concentrates on the reconciliation process, one also

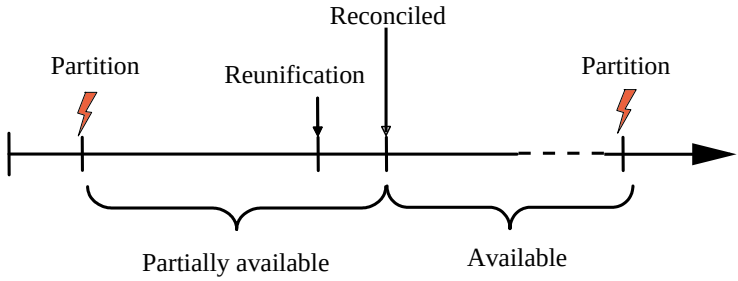

Figure 3.4: CS Optimistic approach: partial service during partitions

needs a replication protocol for the normal and degraded modes of the system. Reconciliation is actually just a part of an optimistic replication scheme. We will assume the existence of a replication protocol that ensures full consistency in normal mode (i.e., no faults) and that provisionally accepts operations during degraded mode (i.e., network partition) provided that no critical constraints are involved. In Chapter 6 we describe the design and implementation of one such replication protocol that is compatible with the CS reconciliation protocol.

We will now proceed with describing the terminology that is used in the rest of this thesis. Then we will continue by describing the system model that we assume.

3.2 Terminology

This section introduces the concepts needed to formally describe the reconciliation protocol and its properties. We will define the necessary terms such as object, partition and replica as well as defining consistency criteria for partitions. These concepts are mainly used in the formal description and analysis of the protocols. Therefore this section can be skimmed on a first reading.

3.2.1 Objects

For the purpose of formalisation we associate data with objects. Implementation-wise, data can be maintained in databases and accessed via database managers.

Definition 1. *An* object *o is a triple $o = (\mathcal{S}, \mathcal{O}, \mathcal{T})$ where $\mathcal{S}$ is the set of possible states, $\mathcal{O}$ is the set of operations that can be applied to the object state and $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{O} \times S$ is a transition relation on states and operations.*

We assume all operation sets to be disjunct so that every operation is associated with one object.

Transitions from a state s to a state s' will be denoted by $s \overset{\alpha}{\rightsquigarrow} s'$ where $\alpha = \langle op, k \rangle$ is an operation instance with $op \in \mathcal{O}$, and $k \in \mathbb{N}$ denotes the unique invocation of operation *op* at some client. Note that this does not imply a total ordering of operations. The identifying numbers are merely used in the formal reasoning to make each operation instance a distinct element.

Definition 2. *An* integrity constraint *c is a predicate over multiple object states. Thus, $c \subseteq \mathcal{S}_1 \times \mathcal{S}_2 \times \ldots \times \mathcal{S}_N$ where N is the number of objects in the system.*

Intuitively, object operations should only be performed if they do not violate integrity constraints.

A distributed system with replication has multiple replicas for every object located on different nodes in the network. As long as no failures occur, the existence of replicas has no effect on the functional behaviour of the system. Therefore, the state of the system in the normal mode can be modelled as a set of replicas, one for each object.

Definition 3. *A* replica *r for object $o = (\mathcal{S}, \mathcal{O}, \mathcal{T})$ is a triple $r = (L, s^0, s^m)$ where the log $L = \langle \alpha_1 \ldots \alpha_m \rangle$ is a sequence of operation instances defined over $\mathcal{O}$. The initial state is $s^0 \in S$ and $s^m \in S$ is a state such that $s^0 \overset{\alpha_1}{\rightsquigarrow} \ldots \overset{\alpha_m}{\rightsquigarrow} s^m$.*

The log can be considered as the record of operations since the last checkpoint that also recorded the (initial) state s^0.

We consider partitions that have been operating independently and we assume the nodes in each partition to acoomodate one primary replica for each object. This will typically be promoted by the middleware. We assume that a replication protocol ensures 1-copy serialisability within a given partition. Moreover, we assume that all objects are replicated across all nodes. For the purpose of reconciliation the important aspect of a partition is not how the actual nodes in the network are connected but the replicas whose states have been updated separately and need to be reconciled. Thus, with the above assumptions, the state of each partition can be modelled as a set of replicas where each object is uniquely represented by the state of the primary replica for that object.

Definition 4. *A* partition state *p is a set of replicas r such that if $r_i, r_j \in p$ are both replicas for object o then $r_i = r_j$.*

The replica states of a partition state $p = \{(L_1, s_1^0, s_1), \ldots, (L_N, s_N^0, s_N)\}$ consists of the state of each of the replicas, that is $\langle s_1, \ldots, s_N \rangle$. Transitions over object states can now be naturally extended to transitions over replica states in a partition.

Definition 5. *Let $\alpha = \langle op, k \rangle$ be an operation instance for some invocation k of operation op. Then $\mathbf{s}^j \overset{\alpha}{\rightsquigarrow} \mathbf{s}^{j+1}$ is a* partition transition *iff there is an*

object o_i *such that* $s_i \overset{\alpha}{\rightsquigarrow} s'_i$ *is a transition for* o_i, $\mathbf{s}^j = \langle s_1, \ldots, s_i, \ldots, s_N \rangle$ *and* $\mathbf{s}^{j+1} = \langle s_1, \ldots, s'_i, \ldots, s_N \rangle$.

3.2.2 Order

So far we have not introduced any concept of order except that a state is always the result of operations performed in some order. When we later will consider the problem of creating new states from operations that have been performed in different partitions we must be able to determine in what (if any) order the operations must be replayed.

At this point we will merely define the existence of a strict partial order relation over operation instances. Later, in Sect. 3.4 we explain the philosophy behind choosing this relation.

Definition 6. *The relation* $\rightarrow$ *is an irreflexive, transitive relation over the operation instances obtained from operations* $\mathcal{O}_1 \cup \ldots \cup \mathcal{O}_N$.

In Definition 8 we will use this ordering to define correctness of a partition state. Note that the ordering relation induces an ordering on states along the time line whereas the consistency constraints relate the states of various objects at a given "time point" (a cut of the distributed system).

3.2.3 Consistency

Our reconciliation protocol will take a set of partition states and produce a new partition state. As there are integrity constraints on the system state and order dependencies on operations, a reconciliation protocol must make sure that the resulting partition state is correct with respect to both of these requirements. This section defines consistency properties for partition states.

Definition 7. *The replica states* $\mathbf{s} = \langle s_1, \ldots, s_N \rangle$ *for partition state* p *where* $p = \{(L_1, s_1^0, s_1), \ldots, (L_N, s_N^0, s_N)\}$ *is* constraint consistent, *denoted* isConsistent(p), *iff for all integrity constraints* c, $\mathbf{s} \in c$.

Next we define a consistency criterion for partition states that also takes into account the order requirements on operations in logs. Intuitively we require that there is some way to construct the current partition state from the initial state using all the operations in the logs. Moreover, all the intermediate states should be constraint consistent and the operation ordering must follow the ordering restrictions. We will use this correctness criterion in evaluation of our reconciliation protocol.

Definition 8. *Let* $p = \{(L_1, s_1^0, s_1), \ldots, (L_N, s_N^0, s_N)\}$ *be a partition state, and let* $\mathbf{s}^k$ *be the k:th version of the replica states. The initial state is* $\mathbf{s}^0 = \langle s_1^0, \ldots s_N^0 \rangle$. *We say that the partition state* p *is* consistent *if there exists a sequence of operation instances* $L = \langle \alpha_1, \ldots, \alpha_m \rangle$ *such that:*

1. $\alpha \in L_i \Rightarrow \alpha \in L$
2. $\mathbf{s}^0 \overset{\alpha_1}{\rightsquigarrow} \ldots \overset{\alpha_m}{\rightsquigarrow} \mathbf{s}^m$
3. *Every* $\mathbf{s}^j \in \{\mathbf{s}^0, \ldots, \mathbf{s}^m\}$ *is constraint consistent*
4. $\alpha_i \rightarrow \alpha_j \Rightarrow i < j$

3.2.4 Utility

Some of the operations will have to be revoked during the reconciliation. In order to be able to choose which operation to revoke it is useful to have a notion of usefulness. For now we will simply define a notion of utility and assume that it is dictated by the application. Later we will discuss how utility can be tied to execution times and what other utility functions could be used.

Definition 9. *Let $\mathcal{O}_i$ be the operations set for object o_i. A* utility function *$U : \bigcup_{i=1}^{N} \mathcal{O}_i \rightarrow \mathbb{R}$ is a mapping from the operations for all objects to a real number.*

The definition can be extended to apply to a partition state by letting the utility of a partition be the sum of the utilities of all the executed operations since the initial state.

3.2.5 Processes

Apart from the data view of a partition which is represented by the states of the replicas in the partition we also need to consider communicating processes. Thus, we can view the system as a set of stateful processes that communicate by message-passing. Later we use timed I/O-automata to describe the behaviour of such processes. With this view, a network partition is characterised by the fact that only a subset of all the nodes is reachable from a given node.

We assume the existence of a group membership service. Remember that the purpose of this service is to keep track of which processes that are reachable at any given point in time. This information is delivered to the processes by so-called views that contain the set of currently reachable processes. This set is referred to as a *group*.

3.3 Fault and Timing Model

We have already stated the need for fault models and timing models when describing algorithms for fault tolerance in distributed systems. We express this as six assumptions on the underlying network behaviour.

We assume that the network may partition multiple times, and that the network repairs completely. That is, no partial network reunifications are

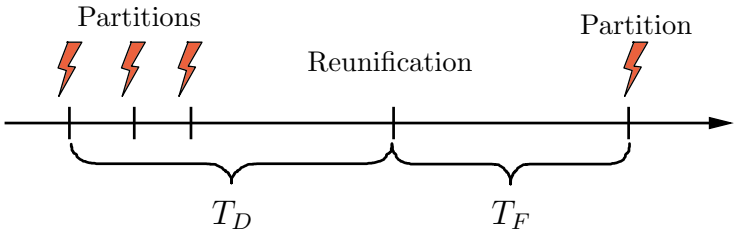

Figure 3.5: Fault model

allowed. The reason for this limitation is that it simplifies the presentation of the algorithms, not that such cases cannot be dealt with. Moreover we assume that nodes never crash. Any pause-crash node (i.e. a node that has a persistence layer) can be seen as a single node that is partitioned away for some time and then rejoins the network.

Moreover, we assume that there are two time bounds on the appearance of faults in the network (see Figure 3.5). T_{D} is the maximal time that the network can be partitioned. T_{F} is needed to capture the minimum time between two faults. The relationship between these bounds is important as operations are piled up during the degraded mode and the reconciliation has to be able to handle them during the time before the next fault occurs.

We will not explicitly describe all the actions of the network but we will give a description of the required actions as well as a list of requirements that the network must meet. The network assumptions are summarised in N1-N6, where N1, N2, and N3 characterise reliable broadcast which can be supplied by a system such as Spread [87]. Assumption N4 relates to partial synchrony which is a basic assumption for fault-tolerant distributed systems. Finally we assume that faults are limited in frequency and duration (N5,N6) which is reasonable, as otherwise the system could never heal itself.

N1 A receive action is preceded by a send (or broadcast) action.

N2 A sent message is not lost unless a network partition occurs.

N3 A sent broadcast message is either received by all processes in the group or a partition occurs.

N4 Messages arrive with a maximum delay of d_{msg} (including broadcast messages).

N5 After a reunification, a partition occurs after an interval of at least T_{F}.

N6 Partitions do not last for more than T_{D}.

3.4 Operation Ordering

In Sect. 3.2.2 we introduced an order relation between operations. In this section we will further elaborate on this concept, and briefly explain why it is important for reconciliation.

We recall that the middleware has to start a reconciliation process when the system recovers from link failures (i.e. when the network is physically reunified). At that point in time there may be several conflicting states for each object since write requests have been serviced in all partitions. In order to merge these states into one common state for the system we can choose to replay the performed operations (that are stored in the logs of each replica). The replay starts from the last common state (i.e. from the state before the partition fault occurred) and iteratively builds up a new state. The question is only in what order should the operations be replayed? It would be tempting to answer this question with "the same order as they were originally applied". But there are two fallacies with this answer. First, it is hard to characterise the order they were originally applied in. Secondly, that order is not necessarily the best one.

In order to deal with this in a proper way, we start by relating a replay order with traditional ordering concepts in distributed systems. As shown in Figure 3.4, each operation invocation is associated with five events: $si(\alpha)$, $ri(\alpha)$, $a(\alpha)$, $sr(\alpha)$, and $rr(\alpha)$. The first and last event, takes place on the client node, and the middle three occur at the server.

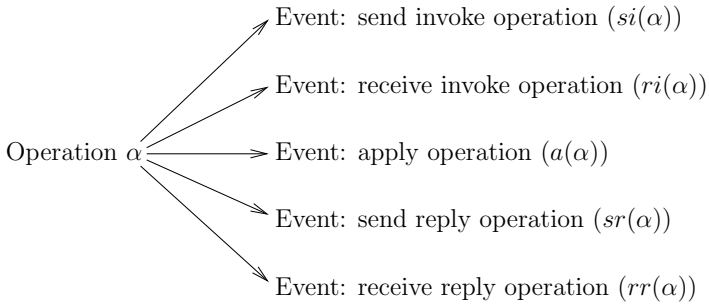

Figure 3.6: Events associated with an operation

We now proceed by first explaining why we need to take client actions into account when determining the replay order. Then we explain why causal order of events induces (in some cases) unnecessary constraints that are expensive to enforce. Finally, we describe what we think is the minimal replay order that is required to meet basic client expectations.

Client excluded order is too weak Lets assume that we have a (asynchronous) system where a number of servers are joined in a logical group that orders event according to some scheme (e.g. causal order). We de-

note this with $e_1 >_A e_2$ if event e_1 precedes event e_2. However, all events taking place outside of this group (i.e., at the clients) are ignored. This is not an unreasonable approach. For crash tolerance it is enough with such a solution, since clients rarely need to be crash tolerant. Now consider the scenario in Figure 3.7.

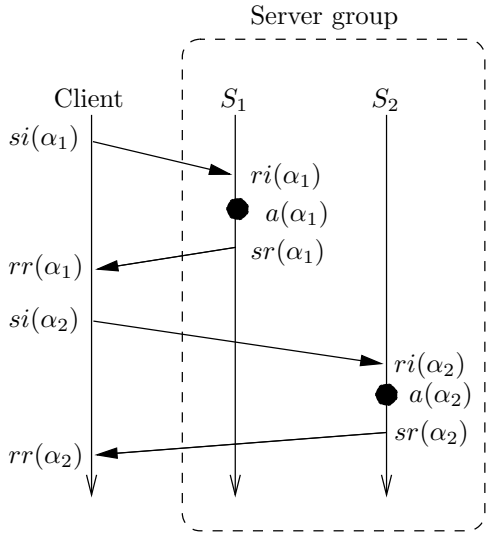

Figure 3.7: Client excluded ordering

Since no messages have passed between S_1 and S_2 in this scenario, there is no way for them to determine the temporal order between events $a(\alpha_1)$ and $a(\alpha_2)$ unless 1) there is a global clock, this is excluded since the system is assumed to be asynchronous 2) the second invocation contains information about the client's logical time.

Now imagine that there is no mechanism to take the client perceived order into account, and that these operations are performed in a degraded system, and should be replayed during reconciliation. In such a case, the system could just as well replay operation α_2 before α_1 as the other way around. Moreover, if there operations that have been performed in another partition that are interleaved with α_1 and α_2, then this change of order will affect the final state.

This completely violates what the client can reasonably expect from the system. The client *knows* that α_1 was executed before α_2 since the reply for α_1 was received even before α_2 was invoked.

Causal order might be too strong Causal ordering of events is a very attractive option. We can define the replay order as follows:

$$\alpha_1 \rightarrow \alpha_2 \text{ if } a(\alpha_1) \succ a(\alpha_2),$$

where $\succ$ is the causal order of the events $a(\alpha_1)$ and $a(\alpha_2)$.

If we adhere to this ordering, there will be no unexpected behaviour. Any observer of system messages will perceive the replayed order to be correct.

So why not use this as the required replay order? The answer is that it is expensive and sometimes even unnecessary. Lets start with the expensive part.

Tracking the causal order of events, requires all messages to be tagged with a logical clock. Although it in principle suffices with Lamport clocks [60] to guarantee causal order, this ordering is stronger than causal and suffers more from the second problem which is discussed below. To exactly capture the causal order, vector clocks are needed where there is one clock for each participating node. This vector with logical clocks needs to be passed along with every message in the network. For a system with hundreds of clients, this is clearly too costly.

The second problem, is that of arbitrary order requirements. Assume that we use Lamport clocks to track time. Now consider the scenario in Figure 3.8. If there have been many internal actions that have occurred in $S1$ the order induced by the Lamport clocks will require that α_2 is replayed before α_1. Although this order is fully acceptable, there is no semantic reason to require this order. It might even be the case that it is impossible to replay the operations in that order due to integrity constraints. Note that this is not a problem of causal ordering per se, but a problem of Lamport's logical clocks.

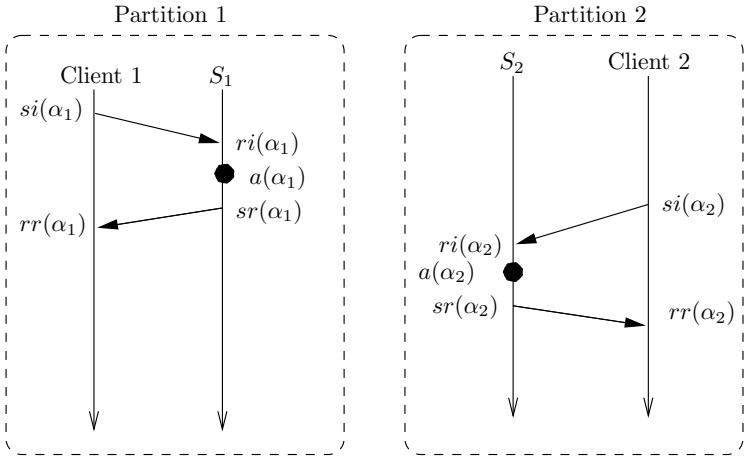

Figure 3.8: Independent operations

In the next chapter we discuss how one can design reconciliation algorithms that try to get a higher utility. By introducing arbitrary order requirements one reduces the possibility of finding good operation orderings.

To summarise the discussion so far, we need some kind of ordering to respect the expectations from clients, but it should be as lightweight as possible to reduce overhead and allow intelligent reconciliation.

Client Expected Order Remember that the client knows that some operations have been executed because of the replies it has received. Therefore we use this expectation to define an ordering relation. When the reconciliation process replays the operations it must make sure that this expected order is respected.

$$\alpha_1 \rightarrow \alpha_2 \text{ iff } rr(\alpha_1) > is(\alpha_2),$$

where $>$ denotes how the events are ordered by the *local* clock at the client node.

This induced order need not be specified at the application level. It can be captured by a client side front end within the middleware, and reflected in a tag for the invoked operations. In Section 6.6 we describe the implementation of such a mechanism.

The main drawback with this ordering is that clients must be independent from each other. If clients communicate, then there might be additional expectations that cannot be guaranteed. In such case, casual ordering would probably be required.

3.5 Integrity Constraints

Since operations will have to be replayed we need to consider the conditions required, so that replaying an operation in a different state than that it was originally executed in does not cause any discrepancies. We assume that such conditions are indeed captured by integrity constraints.

In other words, the middleware expects that an application writer has created the needed integrity constraints such that replaying an operation during reconciliation is harmless as long as the constraint is satisfied, even if the state on which it is replayed is different from the state in which it was earlier executed. That is, there should not be any implicit conditions that are checked by the client at the invocation of the operation. Otherwise, it would not be possible for the middleware to recheck these constraints upon reconciliation.

As an example, consider withdrawal from a credit account. It is acceptable to allow a withdrawal as long as there is coverage for the account in the balance; it is not essential that the balance should be a given value when withdrawal is allowed. Recall that an operation for which a later rejection is not acceptable from an application point of view should be associated with a critical constraint (thereby not applied during a partition at all). An example of such an operation would be the termination of a credit account.

Table 3.1: Notation summary

C	Number of clients.
d_{msg}	Maximal message transmission time.
d_{inv}	Minimal time between two invocations from one client.
d_{han}	Maximal time between two handle actions within reconciliation manager.
d_{act}	Deadline for actions.
d	Upper bound on d_{han} and d_{act}.
f	Number of iterations for STW-Merge and STW-GEU.
L	Sequence of replayed operations.
L_i	Log of operations performed by replica i.
M	Reconciliation manager process.
m	Number of operations performed for a replica.
N	Number of objects.
$\mathcal{O}$	Set of operations for a given object.
o	An object.
P_{viol}	Probability of violation.
p	Partition state, including logs.
q	Number of partitions to be reconciled.
r	An object replica.
s	The state of a replica.
$\mathbf{s}$	State of all replicas in a partition, exluding logs.
$\mathcal{S}$	The set of possible states for an object.
t_j^i	Global time of event i at process j.
$\mathcal{T}$	The set of transitions possible for an object.
$U(\alpha)$	Utility associated with operation α.
T_{F}	Minimal time before a partition fault after a reunify.
T_{D}	Maximal duration of a partition.
α, β, γ	Operation instances.
γ, σ	Sets of admissible timed system traces.

3.6 Notation Summary

We will try to be consistent with use of symbols in this thesis. However, since it might not always be easy to track the meaning of a symbol, we provide a summary in Table 3.1. This is not a complete list, but covers the most frequently used symbols.

"The arrangements were fair to both sides. Since Milo did have freedom of passage everywhere, his planes were able to steal over in a sneak attack without alerting the German aircraft gunners; and since Milo knew about the attack, he was able to alert the German antiaircraft gunners in sufficient time for them to begin firing accurately the moment the planes came into range."

Joseph Heller, Catch 22

4

Reconciliation Algorithms

In this chapter we present four different reconciliation algorithms. Figure 4 categorises them according to whether they are state or operation-based and for the operation-based, whether they require the system to stop invocations during reconciliation, an approach that we denote stop-the-world (STW), or allows continuous service. The Choose1 and STW-Merge algorithms are very simple and should be considered as baselines to which the other two algorithms are compared.

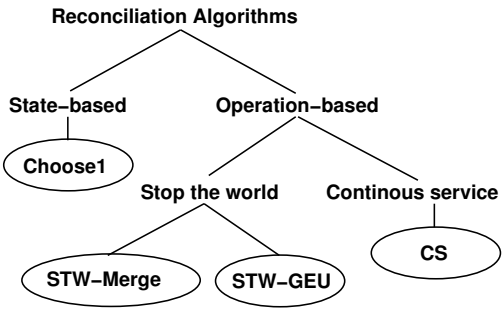

Figure 4.1: Algorithms

The Choose1, STW-Merge, and STW-GEU algorithms are presented as ordinary centralised algorithms. The CS algorithm, on the other hand, requires a more complete description since it is also concerned with what is happening at the respective replicas. Therefore, we have described it using timed I/O-automata of Lynch [61].

4.1 Help Functions

In order to simplify the presentation of the algorithms we start by introducing some help functions. These are fairly straightforward, but by abstracting them in the algorithms we hope to reduce the clutter of letters and symbols. Their formal definitions are only needed for the correctness proofs in Chapter 5, and just as the definitions in the previous chapter they can be skipped in a first reading.

First we introduce a function that applies an operation to the state of the partition. The effect of the function is that one of the replicas in the partition gets its state updated, and the log is appended with the applied operation.

Definition 10. *Let* apply *be a function taking an operation and a partition as arguments and returning an updated partition, formally:*

$$\text{apply}(\alpha, p) =_{def} p \setminus \{r\} \cup r'$$

where $r = (L, s^0, s^m)$ *is a replica for object* $o = (\mathcal{S}, \mathcal{O}, \mathcal{T})$ *and* $\alpha \in \mathcal{O}$. $r' = (L + \langle\alpha\rangle, s^0, s^{m+1})$ *is also a replica for* o *and* $(s^m \overset{\alpha}{\rightsquigarrow} s^{m+1}) \in \mathcal{T}$.

The operation-based algorithms need to collect all the operations that have been performed in a given partition. For the centralised algorithms, this can be formalised with the following function:

Definition 11. *Let* getOps *be a function that takes a partition as argument and returns the set of operations that are present in the logs of the partition.*

$$\text{getOps}(p) =_{def} \{\alpha \mid \alpha \in L \wedge \exists r \in p[r = \langle L, s^0, s\rangle]\}$$

We also need a function that helps to pick out those operations that can be applied from a set of ordered operations. This means choosing the set of operations that have no predecessors.

Definition 12. *Let* minSet *be a function that returns the set of minimal elements in a given subset of all possible operations. Formally:*

$$\text{minSet}(O) =_{def} \{\alpha \mid \alpha \in O \wedge \nexists \beta \in O[\beta \rightarrow \alpha]\}$$

Finally, we present a function called addSuccs that takes two partially ordered sets $A \subset B$ and an element α and adds to A the sole and immediate successors of α that are in B.

Definition 13.

$$\begin{aligned}\text{addSuccs}(A, B, \alpha) =_{def} A \cup \{\beta \in B \mid \alpha \rightarrow \beta \wedge \\ \wedge (\exists \delta \in A.[\delta \rightarrow \beta]) \Rightarrow \delta = \alpha \wedge \\ \wedge \nexists \gamma.[\alpha \rightarrow \gamma \wedge \gamma \rightarrow \beta]\}\end{aligned}$$

This function looks rather esoteric so we will give a brief explanation. First, we add all elements already in A and join it with all elements β in B that fulfil three conditions: (1) β must be a successor of α (2) there is no predecessor to β in A except for α (3) there is no element that is ordered between α and β.

Now, we continue with a baseline algorithm that we use in the evaluation chapter.

4.2 Choose states

The simplest approach for performing state-based reconciliation is to select any of the partitions' states as the final state of the system. An alternative would be to construct the final partition state by combining the object states from several partitions. However, such a constructed state cannot be guaranteed to be consistent. Moreover, it is far from obvious how to efficiently restore consistency in such a system without manual intervention. In Algorithm 1 a partition choosing algorithm is described. Which partition to choose as the resulting partition can depend on the application, but as we will evaluate this algorithm based on the obtained utility we choose the partition resulting in the maximal utility.

Algorithm 1 Choose1

Input: $p_1, \ldots, p_q$ /* q partitions */
Output: $\langle p, rev \rangle$ /* p: A new partition,
rev: revoked operations */

CHOOSE1($p_1, \ldots, p_q$)

1 $p \leftarrow \max_{p' \in \{p_1, \ldots, p_q\}} U(p')$ /* The partition with highest utility */
2 $notChosen \leftarrow \{p_1 \ldots p_q\} \setminus p$ /* The set of partitions not chosen */
3 $rev \leftarrow \bigcup_{p' \in notChosen} \text{getOps}(p')$ /* The set of revoked operations */
4 **return** $\langle p, rev \rangle$

This approach generates a consistent partition. In a way, this approach is very similar to having only one partition (e.g. the majority) operating in the partitioned phase. Although the system proceeds to service in different partitions in degraded mode, there are requests that the clients believe to be accepted and will later be undone.

4.3 Merging operation sequences

The next reconciliation algorithm that we describe works by simply merging the sequences of operations that have been stored in the replicas at each partition and apply them in the required order. During replay, which operation

to replay is chosen arbitrarily as long as the specified strict partial order $\rightarrow$ from Definition 6 is followed.

Algorithm 2 starts by creating a set of candidate elements from which the next operation to execute is chosen. The first candidate contains the elements that have no predecessor in the set of operations. When an operation is executed it is removed from the candidate set and replaced by its immediate and sole successors. If the result is consistent then the new resulting partition is updated. Otherwise the operation is put in the set of revoked operations. This procedure is repeated until all the elements have been considered.

Algorithm 2 STW-Merge

Input: $p_1, \ldots, p_q$ /* q partitions */
$\rightarrow$ /* a partial ordering relation for all the operations in the domain */
Output: $\langle p, rev \rangle$ /* p: A new partition, rev: revoked operations */

STW-Merge$(p_1, \ldots, p_q)$
/* Initialise variables */
1 $p \leftarrow \{(\langle\rangle, s_1^0, s_1^0), \ldots, (\langle\rangle, s_N^0, s_N^0)\}$ /* New system state */
2 $rev \leftarrow \emptyset$ /* Operations to revoke */
3 $ops \leftarrow \bigcup_{p' \in \{p_1, \ldots, p_q\}} \text{getOps}(p')$ /* All operations */
4 $cand \leftarrow \text{minSet}(ops)$ /* Set of candidates to apply */

/* Go through all the operations according to the required order and try to apply them */
5 **while** $cand \neq \emptyset$ **do**
6 $\alpha \leftarrow$ choose arbitrary element from $cand$
7 $cand \leftarrow \text{addSuccs}(cand, ops, \alpha) \setminus \{\alpha\}$
8 **if** isConsistent(apply(α, p)) **then**
9 $p \leftarrow \text{apply}(\alpha, p)$
10 **else**
11 $rev \leftarrow rev \cup \{\alpha\}$
12 **return** $\langle p, rev \rangle$

Note that this algorithm selects the elements nondeterministically. Hence, there are potentially several different results that can be obtained from using this algorithm on the same partition set.

4.4 Greatest Expected Utility

The previous algorithm does not try to order operations to maximise the utility because it has no way of comparing operation sequences and decide on one operation ordering in preference to another. This section presents an algorithm that tries to maximise the expected utility. To explain this concept we will first introduce the concept of probability of violation.

Probability of Violation

A reconciliation process must make decisions about the order that operations are to be performed at each replica. Recall that operation ordering requirements and integrity constraints are orthogonal. Even if the execution of an operation is acceptable with regard to the operation ordering ($\rightarrow$) it can put the partition in an inconsistent state after execution. Moreover, it could be the case that applying a particular operation from another partition may hinder the execution of operations that have taken place within this partition. Trying out the operations to see if an inconsistency is caused can turn out to be very time consuming. So the ideal situation would be to decide whether a given operation should be executed or not without actually executing it. Suppose that for each operation we could determine the probability that we would get an inconsistency if we were to apply it. Then this *probability of violation* could be used to selectively build execution sequences for the resulting partition.

One may wonder how such a parameter would be obtained. In a real application we must be practical and consider if we can calculate this probability from past history or if it is possible to measure. We will explore an approach that is based on system profiling. When the system is functioning in normal mode the consistency violations are immediately detected after applying an operation. By keeping track of how often a particular operation causes an inconsistency we get a rough measure of the probability of violation for that particular operation.

Expected Utility

There are now two ways of measuring the appropriateness of executing a given operation. The utility gives information on how much there is to gain from executing the operation. The probability of violation, on the other hand, tells us what are the chances of getting a result that cannot be used.

The *expected utility* is a measure of what we can expect to achieve given the utility and the violation probability. We quantify the gain of a successful application of the operation α with its utility $U(\alpha)$. The expected utility of operation α is

$$EU(\alpha) = (1 - P_{\text{viol}}(\alpha)) \cdot U(\alpha) - P_{\text{viol}}(\alpha) \cdot U(\alpha) \cdot \theta$$

where $P_{\text{viol}}(\alpha)$ is the probability of violation for operation α. The scaling factor θ reflects the fact that the cost of a violation can be more expensive than just the loss of the utility of the operation. Increasing θ increases the importance of P_{viol}. In all the experiments shown later we equate θ with 10.

The success of the algorithm will depend on how well the expected utility can be estimated. The difficult part of this estimate is to estimate probability of violation. The $U(\alpha)$ part, that is the utility of an operation, can to begin with be uniform for all actions. But profiling will punish those operations whose constraints were often violated in the lifetime of the system.

A system designer may then opt to increase the utility of certain operations that are deemed to be more important than others. It is for example possible to let utilities be set according to the number of nodes that were part of the partition in which the operation was invoked.

The Algorithm

Algorithm 3 STW-GEU

Input: $p_1, \ldots, p_q$ /* q partitions
$\rightarrow$ /* a partial ordering relation for all the operations in the domain */
tryBound /* Maximum number of times an operation is tried */
Output: $\langle p, rev \rangle$ /* p: A new partition
rev: revoked operations */

STW-GEU$(p_1, \ldots, p_q)$
/* Initialise variables */
$p \leftarrow \{(\langle\rangle, s_1^0, s_1^0), \ldots, (\langle\rangle, s_N^0, s_N^0)\}$ /* New system state, starting from pre-fault state */
$rev \leftarrow \emptyset$ /* Operations to revoke */
$ops \leftarrow \bigcup_{p' \in \{p_1, \ldots, p_q\}} \text{getOps}(p')$ /* All operations */
$cand \leftarrow \text{minSet}(ops)$ /* Set of candidates to apply */
$tried \leftarrow \emptyset$ /* Set of tried elements */
$\forall \alpha \in ops : tryCount(\alpha) \leftarrow 0$ /* Number of times that α has been tried */
while $cand \neq \emptyset$ **do**
  if $cand \setminus tried \neq \emptyset$ **then**
    $\alpha \leftarrow \max_{\beta \in (cand \setminus tried)} EU(\beta)$
    if isConsistent(apply(α, p)) **then**
      $p \leftarrow \text{apply}(\alpha, p)$
      $cand \leftarrow \text{addSuccs}(cand, ops, \alpha) \setminus \{\alpha\}$
      $tried \leftarrow \emptyset$
    else /* Couldn't apply the operation */
      $tryCount(\alpha) \leftarrow tryCount(\alpha) + 1$
      if $tryCount(\alpha) > tryBound$ **then**
        $cand \leftarrow \text{addSuccs}(cand, ops, \alpha) \setminus \{\alpha\}$
        $rev \leftarrow rev \cup \{\alpha\}$
      else
        $tried \leftarrow tried \cup \{\alpha\}$
  else /* Tested all ops. in *cand* */
    $\alpha \leftarrow \min_{\beta \in cand} EU(\beta)$ /* Remove op. with lowest EU */
    $tried \leftarrow tried \setminus \{\alpha\}$
    $cand \leftarrow \text{addSuccs}(cand, ops, \alpha) \setminus \{\alpha\}$
    $rev \leftarrow rev \cup \{\alpha\}$
return $\langle p, rev \rangle$

In Algorithm 3 the expected utility is used to choose operation orderings. The algorithm starts by collecting the operations and chooses a set of candidates in the same way as STW-Merge. But instead of choosing arbitrarily among the candidates, the operation with the highest expected utility is chosen.

If an operation is successfully executed (leads to a consistent state) a new set of candidates is created. If, on the other hand, the execution of an operation leads to an inconsistent state then the state is reverted to the previous state and the operation is put in the set of tried operations. The number of times an operation can be unsuccessfully executed due to inconsistencies is bounded by the input parameter *tryBound*. When the number of tries exceeds this bound the operation is abandoned and put in the set of revoked operations.

If the set of tried elements is equal to the set of candidates (meaning that none of the operations could be applied without causing an inconsistency) then the operation with the lowest expected utility is replaced by its successors. This ensures that the algorithm terminates and all operations are tried.

4.5 Continuous Service

Now we continue with the reconciliation protocol that does not require replicas to stop accepting new operations during reconciliation. Before describing the reconciliation protocol in detail we present a short overview of the idea behind the algorithm. Figure 4.2 shows the modes of an optimistic system that uses the CS protocol. The reconciliation protocol is activated upon transition 2.

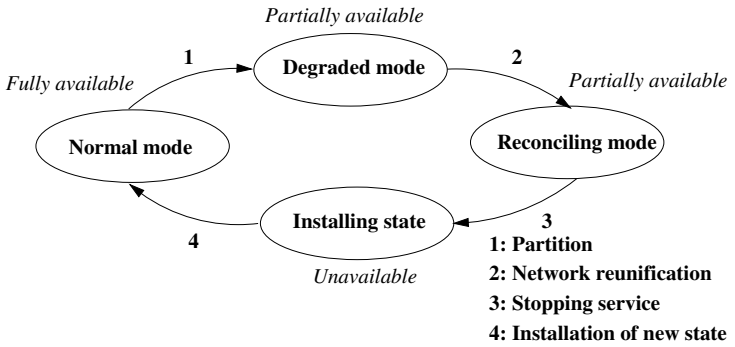

Figure 4.2: System modes

Figure 4.3 shows the two process types present at every node of the system at the middleware layer: the reconciliation manager and the continuous server. Upon launching the reconciliation process (when the network is reunified) the nodes in the new partition elect one reconciliation manager that acts as a coordinator for the given reconciliation duration. Each continuous server has the task of providing the reconciliation manager with logs of operations performed on some object replica at a given node during the degraded mode. It is also responsible for continuously serving new requests

until the reconciliation process ends.

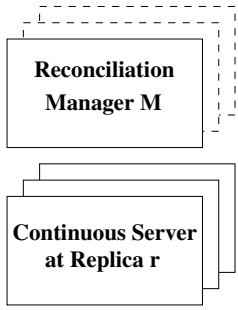

Figure 4.3: Reconciliation Protocol Processes

The reconciliation manager is responsible for merging the operation logs that are sent during reconciling mode. It is activated when the system is reunified and eventually sends an install message with the new partition state to all continuous servers. The new partition state includes empty logs for each continuous server.

The state that is being (re)constructed in the reconciliation manager may not yet reflect all the operations that are before ($\rightarrow$) the new incoming operations. Therefore, the only state in which the incoming operations can be applied to is one of the partition states from the degraded mode. In other words, we need to execute the new operations as if the system was still in degraded mode. In order to do this we will maintain virtual partitions while the reconciliation phase lasts.

4.5.1 Reconciliation Manager

In Algorithm 4 the variable *mode* represents the modes of the reconciliation process and is basically the same as the system modes described in Fig. 4.2 except that the normal and degraded mode are collapsed into an idle mode for the reconciliation manager, which is its initial mode of operation.

When a reunify action is activated the reconciliation manager goes to reconciling mode. Moreover, the variable p, which represents the partition state, is initialised with the pre-partition state, and the variable *ops* that will contain all the operations to replay is set to empty. Now the reconciliation process starts waiting for the continuous servers to send their logs, and the variable *awaitedLogs* is set to contain all servers that have not yet sent their logs. In the formalisation of the algorithm we use the notion from I/O automata [61]. More specifically, $receive(m)_{ij}$ denotes the action that the message m from i has been recieved by process j, and $\leftarrow$ denotes assignment of a state variable.

Next, we consider the action $receive(\langle \text{“log”}, log \rangle)_{iM}$ which will be activated when some continuous server for replica r_i sends its operation log

to M. This action will add logged operations to *ops* and to $acks[i]$ where the latter is used to store acknowledge messages that should be sent back to continuous server for r_i. The acknowledge messages are sent from M to the continuous server for r_i via the action $send(\langle \text{“logAck”}, acks[i] \rangle)_{Mi}$. When logs have been received from all continuous servers (i.e. *awaitedLogs* is empty) then the manager can proceed and start replaying operations. A deadline will be set on when the next handle action must be activated (this is done by setting $last(handle)$).

The action $handle(\alpha)$ is an internal action of the reconciliation process that will replay the operation α (which is minimal according to $\rightarrow$ in *ops*) in the reconciled state that is being constructed. The operation is applied if it results in a constraint-consistent state.

As we will show in Sect. 5.4.2, there will eventually be a time when *ops* is empty at which M will enable $broadcast(\text{“}stop''\text{”})_M$. This will tell all continuous servers to stop accepting new invocations. Moreover, the reconciliation manager will set the mode to *installingState* and wait for all continuous servers to acknowledge the stop message. This is done to guarantee that no messages remain untreated in the reconciliation process. Finally, when the manager has received acknowledgements from all continuous servers it will broadcast an install message with the reconciled partition state and enter idle mode.

4.5.2 The Continuous Server Process

This process (see Algorithm 5) is responsible for receiving invocations to clients and for sending logs to M. We will proceed by describing the states and actions of the continuous server (c-server) process. First note that a c-server process can be in four different modes, normal, degraded, reconciling, and unavailable which correspond to the system modes of Fig. 4.2.

In the formal model we do not explicitly model how updates are replicated from primary replicas to secondary replicas. Instead, we introduce two global shared variables that are accessed by all continuous servers, provided that they are part of the same group. The first shared variable $p[i]$ represents the partition for the group with ID i and it is used by all continuous servers in that group during normal and degraded mode. The group ID is assumed to be delivered by the membership service.

During reconciling mode the group-ID will be 1 for all continuous servers since there is only one partition during reconciling mode. However, as we explained in the beginning of this section the continuous servers must maintain virtual partitions to service requests during reconciliation. The shared variable $vp[j]$ is used to represent the virtual partition for group j which is based on the partition and the membership group that was used during degraded mode.

During normal mode the continuous servers apply operations that are invoked through the $receive(\langle \text{“invoke”}, \alpha \rangle)_{cr}$ action if they result in a constraint-

Algorithm 4 Reconciliation manager M

States

$mode \in \{idle, reconciling, installingState\} \leftarrow idle$
$p \leftarrow \{(\langle\rangle, s_1^0, s_1^0), \ldots, (\langle\rangle, s_N^0, s_N^0)\}$ /* Output of protocol: Constructed state */
$ops \leftarrow \emptyset$ /* Set of operations to reconcile */
$logWait \leftarrow \emptyset$ /* Servers awaited for sending a first log */
$stopAcks \leftarrow \emptyset$ /* Number of received stop "acks"*/
$acks[i] \leftarrow \emptyset$ /* Log items from c-server for replica i to ack.*/
$now \in \mathbb{R}^{0+} \leftarrow 0$
$last(handle) \leftarrow \infty$ /* Deadline for executing handle */
$last(stop) \leftarrow \infty$ /* Deadline for sending stop */
$last(install) \leftarrow \infty$ /* Deadline for sending install */

Actions

Input $reunify(g)_M$
Eff: $mode \leftarrow reconciling$
$p \leftarrow \{(\langle\rangle, s_1^0, s_1^0), \ldots, (\langle\rangle, s_N^0, s_N^0)\}$
$ops \leftarrow \emptyset$
$logWait \leftarrow \{\text{All cont. servers}\}$

Input $receive(\langle \text{"log"}, log\rangle)_{iM}$
Eff: $ops \leftarrow ops \cup log$
$acks[i] \leftarrow acks[i] \cup log$
if $logWait \neq \emptyset$
$logWait \leftarrow logWait \setminus \{i\}$
else
$last(handle) \leftarrow$
$\min(last(handle),$
$now + d_{\text{han}})$

Output $send(\langle \text{"logAck"}, acks[i]\rangle)_{Mi}$
Eff: $acks[i] \leftarrow \emptyset$

Internal $handle(\alpha)$
Pre: $logWait = \emptyset$
$mode = reconciling$
$\alpha \in ops$
$\nexists \beta \in ops \quad \beta \rightarrow \alpha$
Eff: if isConsistent(apply(α, p))
$p \leftarrow \text{apply}(\alpha, p)$
$last(handle) \leftarrow now + d_{\text{han}}$
$ops \leftarrow ops \setminus \{\alpha\}$
if $ops = \emptyset$
$last(stop) = now + d_{\text{act}}$

Output $broadcast(\text{"stop"})_M$
Pre: $ops = \emptyset$
$logWait = \emptyset$
Eff: $stopAcks \leftarrow 0$
$mode \leftarrow installingState$
$last(handle) \leftarrow \infty$
$last(stop) \leftarrow \infty$

Input $receive(\langle \text{"stopAck"}\rangle)_{iM}$
Eff: $stopAcks \leftarrow stopAcks + 1$
if $stopAcks = mn$
$last(install) = now + d_{\text{act}}$

Output $broadcast(\langle \text{"install"}, p\rangle)_M$
Pre: $mode = installingState$
$stopAcks = m \cdot n$
Eff: $mode \leftarrow idle$
$last(install) = \infty$

Timepassage $v(t)$
Pre: $now + t \leq last(handle)$
$now + t \leq last(stop)$
$now + t \leq last(install)$
Eff: $now \leftarrow now + t$

consistent partition; that is, client c invokes and operation α on replica r. A set *toReply* is increased with every applied operation that should be replied to by the action $send(\langle \text{“}reply'', \alpha \rangle)_{rc}$.

A continuous server leaves normal mode and enters degraded mode when the group membership service sends a partition message with a new group-ID. The c-server will then copy the contents of the previous partition representation to one that will be used during degraded mode. Implicit in this assignment is the determination of one primary per partition for each object in the system[1]. The continuous server will continue accepting invocations and replying to them during degraded mode.

When a c-server receives a reunify message it will take the log of operations served during degraded mode (the set *log*) and sends it to the reconciliation manager M by the action $send(\langle \text{“}log'', log \rangle)_{rM}$. In addition, the c-server will enter reconciling mode and copy the partition representation to a virtual partition representation. The latter will be indexed using virtual group-ID vg which will be the same as the group-ID used during degraded mode. Finally, a deadline will be set for sending the logs to M.

The continuous server will continue to accept invocations during reconciliation mode with some differences in handling. First of all, the operations are applied to a virtual partition state. Secondly, a log message containing an applied operation is immediately scheduled to be sent to M. Finally, the c-server will not immediately reply to the operations. Instead it will wait until the log message has been acknowledged by the reconciliation manager and $receive(\langle \text{“logAck”}, log \rangle)_{Mr}$ is activated. The reason for this is to ensure that no operations gets replayed in the wrong order. Now any operation whose reply was pending and for whom a “logAck” has been received can be replied to (added to the set *toReply*).

At some point the manager M will send a stop message which will make the continuous server to go into unavailable mode and send a stopAck message. During this mode no invocations will be accepted until an install message is received. Upon receiving such a message the c-server will install the new partition representation and once again go into normal mode.

4.6 Additional Notes

In this section we describe how to adapt the proposed algorithms to handle some of the problems that occur in real applications. We have chosen not to include them in the formalisation since they are easily understood and would only clutter up the descriptions. The more far-reaching changes that results in completely different algorithms are presented as future work in Section 8.2.

[1] This is handled by the replication protocol in conjunction with the name service and group membership service and thus not modelled explicitly.

Algorithm 5 Continuous server (c-server) for replica r

Shared vars

$p[i] \leftarrow \{(\langle\rangle, s_1^0, s_1^0), \ldots, (\langle\rangle, s_N^0, s_N^0)\}$, for $i = 1 \ldots q$ /* Representation for partition i, before reunification */

$vp[i]$, for $i = 1 \ldots q$ /* Representation for virtual partition i, after reunification */

States

$mode \in \{idle, normal, degraded, reconciling, unavailable\} \leftarrow idle$

$g \in \{1 \ldots q\} \leftarrow 1$ /* Group identity (supplied by group membership service) */

$vg \in \{1 \ldots q\} \leftarrow 1$ /* Virtual group identity, used between reunification and install */

$log \leftarrow \emptyset$ /* Set of log messages to send to reconciliation manager M*/

$toReply \leftarrow \emptyset$ /* Set of operations to reply to */

$pending \leftarrow \emptyset$ /* Set of operations to reply to when logged */

$enableStopAck$ /* Boolean to signal that a "stopAck" should be sent */

$last(log) \leftarrow \infty$ /* Deadline for next $send(\langle$"log'', $\ldots\rangle)$ action */

$last(stopAck) \leftarrow \infty$ /* Deadline for next $send(\langle$"stopAck", $\ldots\rangle)$ action */

$now \in \mathbb{R}^{0+} \leftarrow 0$

Actions

Input $partition(g')_r$
Eff: $mode \leftarrow degraded$
$p[g'] \leftarrow p[g]$
$g \leftarrow g'$

Input $reunify(g')_r$
Eff: $log \leftarrow L_r$ where $\langle L_r, s_r^0, s_r\rangle \in p$
$mode \leftarrow reconciling$
$vg \leftarrow g$
$vp[vg] \leftarrow p[g]$
$g \leftarrow g'$
$last(log) \leftarrow now + d_{\text{act}}$

Input $receive(\langle$"invoke", $\alpha\rangle)_{cr}$
Eff: $switch(mode)$
$normal \mid degraded \Rightarrow$
if isConsistent(apply($\alpha, p[g]$))
$p[g] \leftarrow$ apply($\alpha, p[g]$)
$toReply \leftarrow toReply \cup \{\langle\alpha, c\rangle\}$
$reconciling \Rightarrow$
if isConsistent(apply($\alpha, vp[vg]$))
$vp[vg] \leftarrow$ apply($\alpha, vp[vg]$)
$log \leftarrow log \cup \{\alpha\}$
$last(log) \leftarrow$
min($last(log), now + d_{\text{act}}$)
$pending \leftarrow pending \cup \{\langle\alpha, c\rangle\}$

Output $send(\langle$"log", $log\rangle)_{rM}$
Pre: $mode \in \{reconciling, unavailable\}$
$log \neq \emptyset$
Eff: $log \leftarrow \emptyset$
$last(log) \leftarrow \infty$

Input $receive(\langle$"logAck", $log\rangle)_{Mr}$
Eff: $replies \leftarrow \{\langle\alpha, c\rangle \in pending \mid \alpha \in log\}$
$toReply \leftarrow toReply \cup replies$
$pending \leftarrow pending \setminus replies$

Output $send(\langle$"reply", $\alpha\rangle)_{rc}$
Pre: $\langle\alpha, c\rangle \in toReply$
Eff: $toReply \leftarrow toReply \setminus \{\langle\alpha, c\rangle\}$

Input $receive($"stop"$)_{Mr}$
Eff: $mode \leftarrow unavailable$
$enableStopAck \leftarrow true$
$last(stopAck) \leftarrow now + d_{\text{act}}$

Output $send(\langle$"stopAck"$\rangle)_{rM}$
Pre: $enableStopAck = true$
$log = \emptyset$
Eff: $enableStopAck = false$
$last(stopAck) \leftarrow \infty$

Input $receive(\langle$"$install''$, $p'\rangle)_{Mr}$
Eff: $p[g] \leftarrow p'$ /* g = 1 */
$mode \leftarrow normal$

Timepassage $v(t)$
Pre: $now + t \leq last(log)$
$now + t \leq last(stopAck)$
Eff: $now \leftarrow now + t$

Nested invocations We have not dealt explicitly with nested invocations in our formalisations. The best way to do this which is also easily implemented is to log only top-level operations. This means that if the invocation α causes the sub-invocations β and γ then only α is logged. During the reconciliation process, when α is replayed, so are β and γ since they are invoked automatically by the object where α is replayed. Such behaviour is supported by the sandbox construction described in Chapter 6

State transfer State transfer can be very expensive in systems with large state. Therefore the install message which ends the CS algorithm with the replica states might be unreasonable. The obvious fix to this is to replace the full installation state with information on what operations to replay in each replica. This log would need to include not only top-level operations (as discussed above) but all operations that are to be performed on a given replica. Moreover, the replica needs to be able to apply this operation log without side-effects. This mechanism is no different from that required to support operation-based replication. Naturally, a system that cannot afford state-transfer during reconciliation would also require operation-based replication.

Compensation We have chosen to revoke operations that cannot be replayed due to violations of integrity constraints. As we have already noted, this is not possible for operations with side-effects. Therefore revocations should be replaced with compensations. However, the effect of applying a compensating action should be the same as revoking the operation, namely keeping the state consistent.

Full replication Our system model assumes full replication. That is, we assume that every node hosts a replica for every object. For a real system such a scheme would be very expensive. However, this assumption was only made to simplify the presentation. In the implementation of the CORBA middleware, we allowed any kind of replication strategy. This is accomplished by letting the continuous servers, at the beginning of the reconciliation phase, report to the reconciliation manager which objects they host replicas for.

"$-1 = i \cdot i = \sqrt{-1} \cdot \sqrt{-1} = \sqrt{-1 \cdot -1} = \sqrt{1} = 1$"

Unknown

5 Correctness

The goal of the reconciliation protocols is to restore consistency in the system. This is achieved by merging the results from several different partitions into one partition state. The clients have no control over the reconciliation process and in order to guarantee that the final result does not violate the expectations of the clients we need to assert correctness properties of the protocol. In this chapter we analyse the properties of the algorithms described in Chapter 4. In particular we are interested in showing termination and correctness. By correctness we mean that they produce a final state that is consistent according to Definition 7.

Choose1 is trivially correct in the sense that if the replication protocol maintains consistency in *each* of the partitions, then by choosing the state from one of these partitions as the final state will also give a final state. Moreover termination is trivial as there are no loops in the algorithm.

As for the other algorithms they have been presented in two different ways, and therefore two different proof techniques will be used. The centralised algorithms (STW-Merge, STW-GEU) will be analysed by asserting certain properties of the state variables used in the algorithms. The CS protocol, which is a distributed protocol, will be analysed by considering sequences of event traces for the entire system.

5.1 StopTheWorld-Merge

We start with the STW-Merge algorithm, since it is fairly straightforward, and it will help us to show correctness for the more intricate STW-GEU algorithm.

5.2 Assumptions

In order to prove the desired properties we need to make two basic assumptions. First, in order to prove that the reconciliation phase ends with the installment of a consistent partition state, we need to assume that the state from which the reconciliation started is consistent. This is a reasonable assumption since normal and degraded mode operations always respect integrity constraints. Second, we assume that the logs that have been collected during degraded mode are finite.

A1 The initial partition state $\mathbf{s}^0$ is constraint consistent (see Definition 7).

A2 All partition logs are finite sets.

5.2.1 Notation

The proof will assert certain relationships between the state variables for different stages of the algorithm. To make time explicit, we will with $A^{i:j}$ denote the value of variable A in the i:th iteration of the loop, after the execution of line j. For example $A^{1:4}$ denotes the value of the set A after line 4 has executed.

5.2.2 Some basic properties

To show termination and correctness we first need to establish some basic facts. The first one being that the candidate set *cand* is always a subset of *ops* (i.e., the set of operations to replay).

Lemma 1. $cand^{i:j} \subseteq ops^{k,l},\ \forall i, j, k, l$

Proof. This is immediate, since all operations that are added to *cand* (lines 4 and 7) already belong to *ops*, and elements are never removed from *ops*. □

Next, we want to assert that the set *cand* will never contain an operation that precedes previous members of *cand*.

Lemma 2. $\forall \alpha_i \in cand^{i:5}, \alpha_j \in cand^{j:5} : \alpha_i \rightarrow \alpha_j \Rightarrow i < j$

Proof. We show this by induction over i.

Base case, $i = 1$: Since $j \geq 1$, we need to show that if $\alpha_i \rightarrow \alpha_j$ then $j \neq 1$. From line 4 we have that $cand^{1:5} = \text{minSet}(ops^{1:3})$. According to the definition of minSet, there can be no elements in $cand^{1:5}$ with a predecessor in $ops^{1:3} \supseteq cand^{1:5}$. Therefore j cannot be zero if $\alpha_i \rightarrow \alpha_j$.

Inductive step: Assume that $\forall \alpha_i \in cand^{i:5}, \alpha_j \in cand^{j:5} : \alpha_i \rightarrow \alpha_j \Rightarrow i < j$ holds for all $i \leq n$. We will show that it also holds for $i = n + 1$, that is, $\forall \alpha_j \in cand^{j:5} : \alpha_{n+1} \rightarrow \alpha_j \Rightarrow n + 1 < j$.

Consider the set $cand^{n+1:5}$. According to line 7 this is equal to addSuccs($cand^{n:5}, ops^{1:3}, \alpha_n) \setminus \{\alpha_n\}$ for some operation $\alpha_n \in cand^{n:5}$. Now consider an arbitrary element α_{n+1} in $cand^{n+1:5}$. There are two cases to consider:

(1) α_{n+1} was also a member of $cand^{n:5}$. According to the definition of addSuccs, there can be no successors of α_{n+1} added to $cand^{n+1:5}$, since only successors of the removed element α_n are added. This fact, together with the inductive assumption gives that $\forall \alpha_j \in cand^{j:5} : \alpha_{n+1} \rightarrow \alpha_j \Rightarrow n+1 < j$.

(2) α_{n+1} was not a member of $cand^{n:5}$, which means that it was added by the addSuccs function. This means that $\alpha_n \rightarrow \alpha_{n+1}$, so any successor to α_{n+1} is also a successor to α_n. Thus there can be no successors to α_{n+1} in any $cand^{i:5}, i \leq n$. Moreover, there can be no successors to α_{n+1} in $cand^{n+1:5}$, since such elements are excluded by the addSuccs function. Therefore $\forall \alpha_j \in cand^{j:5} : \alpha_{n+1} \rightarrow \alpha_j \Rightarrow n+1 < j$. This concludes the proof.

□

The following corollary to Lemma 2 comes from the fact that in order to apply an operation in iteration i, the operation must be a member of $cand^{i:5}$.

Corollary 1. *If α_i and α_j were applied in iteration i and j, respectively, and $\alpha_i \rightarrow \alpha_j$, then $i < j$.*

Finally, before proceeding with the termination theorem, we need to assert that elements are only added to *cand* once.

Lemma 3. *No element from ops is added to cand twice.*

Proof. Assume for contradiction that α is added twice to *cand*, say iterations i and j, $i < j$. The second time, this must be by the addSuccs function on line 7. By the definition of addSuccs there must be an element $\beta \in cand^{j-1:5}$ s.t. $\beta \rightarrow \alpha$. But according to Lemma 2 this means that $j - 1 < i$, even though we assumed $i < j$, a contradiction. □

5.2.3 Termination

Now we have all that we need to show the termination property for STW-Merge.

Theorem 1. *The STW-Merge algorithm terminates*

Proof. On line 5 we can see that the algorithm terminates when *cand* is empty. Line 7 is the only line where *cand* is changed (after getting the initial value on line 3). The initial value of *cand* is finite, since it is a subset of the finite set *ops* (see Lemma 1 and Assumption A2). One element is

removed every iteration (see line 7). Moreover, all the elements that are added are added from the finite set *ops*. Finally, by Lemma 3 no element from *ops* is added twice. Therefore *cand* will eventually be empty. □

5.2.4 Correctness

It is intuitively rather clear that STW-Merge will result in a consistent state when it finishes. However, for completeness we state this formally and prove this by induction.

Theorem 2. *The resulting partition $p^{f:12}$ from STW-Merge, where f is the final iteration of the loop is consistent according to Definition 8.*

Proof. Let L^i be the sequence of all operations that have been applied until the end of iteration i. We will show that L^f is a sequence that satisfies the requirements for $p^{f:5} = p^{f:12}$ to be consistent by using induction over i.

Base Case, $i = 1$: The partition $p^{1:5} = \{(\langle\rangle, s_1^0, s_1^0), \ldots, (\langle\rangle, s_N^0, s_N^0)\}$ is consistent, since requirements 1,2 and 4 of Definition 8 are vacuously true and 3 is given by A1.

Inductive step: Assume that L^i satisfies the requirements for $p^{i:5}$ to be consistent.

We will consider two cases based on whether applying α results in an inconsistent state or not (line 8).

(1) If apply$(\alpha, p^{i+1:8})$ is not constraint consistent then the if-statement on line 8 is false and the partition state will remain unchanged ($p^{i+1:5} = p^{i:5}$), and thus $p^{i+1:5}$ is consistent according to the inductive assumption.

(2) If apply$(\alpha, p^{i+1:8})$ is constraint consistent then the partition state $p^{i+1:5}$ will be set to apply$(\alpha, p^{i:5})$. By the inductive assumption we know that L^i leads to $p^{i:5}$. We will show that the sequence $L^{i+1} = L^i + \langle\alpha\rangle$ satisfies the requirements for $p^{i+1:5}$ to be consistent.

Consider the conditions 1-4 in the definition of consistent partition (Def. 8).

1. By the definition of apply we know that all replicas in p remain unchanged except one which we denote r. So for all replicas $\langle L_j, s_j^0, s_j\rangle \neq r$ we know that $\beta \in L_j \Rightarrow \beta \in L^i \Rightarrow \beta \in L^{i+1}$. Moreover, the new log of replica r will be the same as the old log with the addition of operation α. And since all elements of the old log for r are in L^i, they are also in L^{i+1}. Finally, since α is in L^{i+1} then all operations for the log of r leading to $p^{i+1:5}$ are in L^{i+1}.

2. Consider the last state $\mathbf{s}^k = \langle s_1, \ldots, s_j, \ldots s_N \rangle$ where s_j is the state of the replica that will be changed by applying α. Let s'_j be the state of this replica in p^{i+1} which is the result of the transition $s_j \overset{\alpha}{\rightsquigarrow} s'_j$. By the inductive assumption we have that $\mathbf{s}^0 \overset{\alpha_1}{\rightsquigarrow} \ldots \overset{\alpha_k}{\rightsquigarrow} \mathbf{s}^k$. Then $\mathbf{s}^0 \overset{\alpha_1}{\rightsquigarrow} \ldots \overset{\alpha_k}{\rightsquigarrow} \mathbf{s}^k \overset{\alpha}{\rightsquigarrow} \mathbf{s}^{k+1}$ where $\mathbf{s}^{k+1} = \langle s_1, \ldots, s'_j, \ldots s_N \rangle$ is a partition transition according to Definition 5.
3. By the inductive assumption we know that $p^{i:5}$ is consistent and therefore $\forall j \leq k$ $\mathbf{s}^j$ is constraint consistent. Further since $\text{apply}(\alpha, p^{i:5})$ is constraint consistent according to (2), $\mathbf{s}^{k+1}$ is constraint consistent.
4. This follows directly from Corollary 1.

□

5.3 StopTheWorld-GEU

The STW-GEU algorithm is based on STW-Merge, and proving their correctness is done in a very similar way. Therefore, we will merely provide the proofs in this section were they differ from the proofs for STW-Merge. Moreover, we use the same assumptions and notation as in the previous section. Next we will revisit the basic properties that were established for STW-Merge and see that they hold for STW-GEU as well. We will also add one lemma that is specific for STW-GEU.

5.3.1 Basic properties

First of all, Lemma 1 and Lemma 3 holds for STW-GEU for the same reasons as for STW-Merge.

We also need the ordering lemma for STW-GEU that states that if two operations are ordered, then the candidate sets which they belong to follow the same order:

Lemma 4. $\forall \alpha_i \in cand^{i:7}, \alpha_j \in cand^{j:7} : \alpha_i \rightarrow \alpha_j \Rightarrow i < j$

The proof for this lemma is analogous to the proof of Lemma 2. The only difference is that for some iterations i, it is true that $cand^{i:7} = cand^{i+1:7}$ since addSuccs may not be executed in all iterations. However, it is easy to see that it is never called twice in one iteration.

And with the ordering lemma we have the corollary that constrains the order of applied operations:

Corollary 2. *If α_i and α_j were applied in iteration i and j, respectively, and $\alpha_i \rightarrow \alpha_j$, then $i < j$.*

The only new property that we need to assert for STW-GEU is the one that states that the set of tried elements is always a subset of the set of operations to replay:

Lemma 5. $tried^{i:25} \subseteq ops^{1:3}, \forall i$

Proof. This is realised since *tried* is only added elements on line 23, with an operation that is a member of *cand* (line 9) which in turn is a subset of $ops^{1:3}$ (Lemma 1). □

5.3.2 Termination

The fact that the STW-GEU algorithm terminates is not at all as easy to see as it is for STW-Merge since operations are tried several times. However, as we will show, each iteration progresses either by removing an element from the candidate set or by trying a previously untried (for that particular configuration) operation.

Theorem 3. *The STW-GEU algorithm terminates*

Proof. We will prove this by introducing a function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that $f(i)$ returns a natural number for iteration i. We will show that f is monotonically increasing and that it has an upper bound. Let $f(i) = (|ops^{1:3}| + 2) * r(i) + |tried^{i:25}|$, where the function $r(i)$ returns the number of operations that have been removed from *cand* until the end of iteration i.

There are four cases to consider depending on what lines are executed for each iteration as shown in parentheses for each case. Note that only one case is applicable for each iteration, and that the denoted lines are the only lines that are executed during the iteration for each particular case.

(7-13): $|tried^{i:25}|$ will be zero after these lines, but $r(i)$ will be increased with one. Since $|tried^{i:25}| \leq |ops^{1:3}|$ (Lemma 5), $f(i)$ will increase.

(7-9,14-18): $r(i)$ will increase with one, and thus $f(i)$ will increase.

(7-9,14-16,19-20): From line 9 we know that α was not in *tried* so $|tried|$ will increase with one while $r(i)$ stays constant, and thus $f(i)$ will increase.

(7,21-25): $|tried|$ is reduced at most by one, and $r(i)$ is increased by one so $f(i)$ will increase.

So for all four cases $f(i)$ will increase, and it is therefore monotonically increasing. It follows directly from Lemma 3 and the fact that *cand* is initially empty that no element can be removed twice from *cand*. Therefore $r(i) \leq ops^{1:3}$. So the upper bound on $f(i)$ is $(|ops^{1:3}|+2)\cdot|ops^{1:3}|+|ops^{1:3}| = |ops^{1:3}|\cdot|ops^{1:3}|+3\cdot|ops^{1:3}|$. This concludes the termination proof for STW-GEU. □

5.3.3 Correctness

The correctness proof for STW-GEU is completely analogous to the proof of Theorem 2. Therefore we will merely state the theorem here.

Theorem 4. *The resulting partition $p^{f:26}$ from STW-GEU, where f is the final iteration of the loop is consistent according to Definition 8.*

Now that we have analysed the two centralised reconciliation algorithms, we turn the attention to the distributed CS reconciliation protocol.

5.4 Continuous Service Protocol

The CS protocol provides more of a challenge to prove correct, as there is a growing set of unreconciled operations that is accumulated while the reconciliation occurs. So we need to show that the protocol does not get stuck in reconciliation mode for ever despite incoming operations. In this section we will show that (1) the CS protocol terminates in the sense that the reconciliation mode eventually ends and the system proceeds to normal mode (2) the resulting partition state which is installed in the system is consistent in the sense of Definition 8.

5.4.1 Assumptions

The results rely on a number of assumptions on the system. We assume that network to fulfil the fault and timing assumptions N1-N6 described in Section 3.3. Moreover, we need to assume some restrictions on the behaviour of the clients such as the speed at which invocations are done and the expected order of operations. The rest of the section describes these assumptions in more detail.

Client Assumptions.

In order to prove termination and correctness of the reconciliation protocol we need some restrictions on the behaviour of clients. First of all the time between two client invocations is not allowed to be too short. Basically, this restricts the total load of the system and allows the reconciliation to finish. Secondly, we assume that the required $\rightarrow$ order fulfils the client-expected-order semantics that we described in Section 3.4.

C1 The minimum time between two invoke actions from one client is d_{inv}.

C2 If there is an application-specific ordering between two operations, then the first operation must have been replied to before the second was invoked. Formally, admissible timed system traces must be a subset of $ttraces(C2)$. $ttraces(C2)$ is defined as the set of sequences such that for all sequences σ in $ttraces(C2)$:

$\alpha \rightarrow \beta$ and $(send(\langle \text{“invoke”}, \beta \rangle)_{cr}, t_1) \in \sigma \Rightarrow$
$\exists (receive(\langle \text{“reply”}, \alpha \rangle)_{r'c}, t_0) \in \sigma$ for some r' and $t_0 < t_1$.

Server Assumptions.

As we are concerned with reconciliation and do not want go into detail on other responsibilities of the servers or middleware (such as checkpointing), we will make two assumptions on the system behaviour that we do not explicitly model. First we assume that the state from which reconciliation starts is consistent. This is the same assumption that was made for STW-Merge and STW-GEU. Second, we assume that the replica logs are empty at the time when a partition occurs. This is required to limit the length of the reconciliation as we do not want to consider logs from the whole life time of a system. In Section 6.4.1 we discuss how this can be enforced in an implementation.

A1 The initial state $\mathbf{s}^0$ is constraint consistent (see Definition 7).

A2 All replica logs are empty when a partition occurs.

We will now proceed to prove correctness of the protocol. First we give a termination proof and then a partial correctness proof.

5.4.2 Termination

In this section we will prove that the reconciliation protocol will terminate in the sense that after the network is physically healed (reunified) the reconciliation protocol eventually activates an install message to the c-servers with the reconciled state. As stated in the theorem it is necessary that the system is able to replay operations at a higher rate than new operations arrive (reflected in the ratio q).

We start by defining the ratio q as the minimum handling rate $\frac{1}{d_{\text{han}}}$ divided by the maximum inter-arrival rate for client invocations $C \cdot \frac{1}{d_{\text{inv}}}$, where C is the maximum number of clients.

Definition 14. *Let* $q =_{def} \frac{d_{\text{inv}}}{C \cdot d_{\text{han}}}$.

In order to show the termination we need to characterise all the timed traces in which every reunification action is followed by an install action.

Definition 15. *Let the set* $ttraces(Installing)$ *of action sequences be such that for every* $(reunify(g)_M, t)$ *there is a* $(broadcast(\langle \text{“install”}, p \rangle)_M, t')$ *in the sequence, with* $t < t'$.

Now we can state the theorem that states the conditions required for termination:

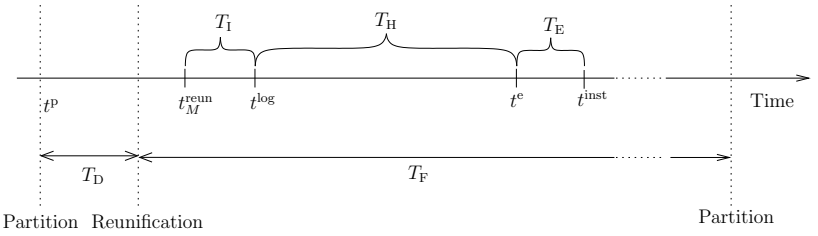

Figure 5.1: Reconciliation time line

Theorem 5. *All admissible system traces are in the set ttraces(Installing), provided that* $q > 1$ *and that* $T_{\mathrm{F}} > \frac{T_{\mathrm{D}}+7d}{q-1} + 9d$*, where* d *exceeds* d_{msg} *and* d_{act}*.*

Proof. Consider an arbitrary admissible timed trace γ such that $(\mathit{reunify}(g)_M, t_M^{\mathrm{reun}})$ appears in γ. Let all time points t^i below refer to points in γ. The goal of the proof is to show that there exists a point t^{inst} after t_M^{reun}, at which there is an install message appearing in γ.

The timing relation between two partitions and the time line for manager M can be visualised in Fig. 5.1 (see N5 and N6). Let t_i^{reun} denote the time point at which the reunification message arrives at process i. The reconciliation activity is performed over three intervals: initialising (T_{I}), handling (T_{H}), and ending (T_{E}). The proof strategy is to show that the reconciliation activity ends before the next partition occurs, considering that it takes one message transmission for the manager to learn about reunification. That is, $d_{\mathrm{msg}} + T_{\mathrm{I}} + T_{\mathrm{H}} + T_{\mathrm{E}} < T_{\mathrm{F}}$.

Let t^{log} be the last time point at which a log message containing a pre-reunification log is received from some c-server. This is the time point at which handling (replaying) operations can begin. The handling interval (T_{H}) ends when the set of operations to replay (*ops*) is empty. Let this time point be denoted by t^e.

Initialising:

$$T_{\mathrm{I}} = t^{\mathrm{log}} - t_M^{\mathrm{reun}}$$

The latest estimate for t^{log} is obtained from the latest time point at which a c-server may receive this reunification message (t_r^{reun}) plus the maximum time for it to react (d_{act}) plus the maximum transmission time (d_{msg}).

$$T_{\mathrm{I}} \leq \max_r(t_r^{\mathrm{reun}}) + d_{\mathrm{act}} + d_{\mathrm{msg}} - t_M^{\mathrm{reun}}$$

By N3 and N4 all reunification messages are received within d_{msg}.

$$T_{\mathrm{I}} \leq t_M^{\mathrm{reun}} + d_{\mathrm{msg}} + d_{\mathrm{act}} + d_{\mathrm{msg}} - t_M^{\mathrm{reun}} \leq 2d_{\mathrm{msg}} + d_{\mathrm{act}} \qquad (5.1)$$

Handling: The maximum handling time is characterised by the maximum number of invoked client requests times the maximum handling time for each operation (d_{han}, see Algorithm 4), times the maximum number of clients C. We divide client invocations in two categories, those that arrive at the reconciliation manager before $t^{\log}$ and those that arrive after $t^{\log}$.

$$T_{\text{H}} \leq \big([\text{pre-}t^{\log}\text{ messages}] + [\text{post-}t^{\log}\text{ messages}]\big) \cdot C \cdot d_{\text{han}}$$

The maximum time that it takes for a client invocation to be logged at M is equal to $2d_{\text{msg}} + d_{\text{act}}$, consisting of the transmission time from client to the c-server and the transmission time from c-server to manager as well as the reaction time for the c-server. The worst case estimate of the number of post-$t^{\log}$ messages includes all invocations that were initiated at a client prior to $t^{\log}$ and logged at M after $t^{\log}$. Thus the interval of $2d_{\text{msg}} + d_{\text{act}}$ must be added to the interval over which client invocations are counted.

$$T_{\text{H}} \leq \left(\frac{T_{\text{D}} + d_{\text{msg}} + T_{\text{I}}}{d_{\text{inv}}} + \frac{T_{\text{H}} + 2d_{\text{msg}} + d_{\text{act}}}{d_{\text{inv}}} \right) \cdot C \cdot d_{\text{han}} \tag{5.2}$$

using earlier constraint for T_{I} in (5.1). Finally, together with the assumption in the theorem we can simplify the expression as follows:

$$T_{\text{H}} \leq \frac{T_{\text{D}} + 5d_{\text{msg}} + 2d_{\text{act}}}{q - 1} \tag{5.3}$$

Ending: According to the model of reconciliation manager M an empty *ops* results in the sending of a stop message within d_{act}. Upon receiving the message at every c-server (within d_{msg}), the c-server acknowledges the stop message within d_{act}. The new partition can be installed as soon as all acknowledge messages are received (within d_{msg}) but at the latest within d_{act}. Hence T_{E} can be constrained as follows:

$$T_{\text{E}} = t^{\text{inst}} - t^{e} \leq 3d_{\text{act}} + 2d_{\text{msg}} \tag{5.4}$$

Final step: Now we need to show that $d_{\text{msg}} + T_{\text{I}} + T_{\text{H}} + T_{\text{E}}$ is less than T_{F} (time to next partition according to N5). From (5.1), (5.3), and (5.4) we have that:

$$T_{\text{I}} + T_{\text{H}} + T_{\text{E}} \leq 2d_{\text{msg}} + d_{\text{act}} + \frac{T_{\text{D}} + 5d_{\text{msg}} + 2d_{\text{act}}}{q - 1} + 3d_{\text{act}} + 2d_{\text{msg}}$$

Given a bound d on delays d_{act} and d_{msg} we have:

$$d_{\text{msg}} + T_{\text{I}} + T_{\text{H}} + T_{\text{E}} \leq \frac{T_{\text{D}} + 7d}{q - 1} + 9d$$

Which concludes the proof according to theorem assumptions. □

5.4.3 Correctness

The main correctness requirement on the reconciliation protocol is to preserve consistency. The model of the c-servers obviously keeps the partition state consistent (see the action under $receive(\langle \text{“invoke”}, \alpha \rangle)_{cr}$. The proof of correctness is therefore about the manager M withholding this consistency during reconciliation, and specially when replaying actions. Before we go on to the main theorem on correctness we present a theorem that shows the ordering requirements of the application (induced by client actions) are respected by our models. Again we first define the subset of possible traces.

Definition 16. *Define the set $ttraces(Order)$ as the set of all action sequences with monotonically increasing times with the following property: for any sequence $\sigma \in ttraces(Order)$, if $(handle((\alpha), t)$ and $(handle((\beta), t')$ is in σ, $\alpha \rightarrow \beta$, and there is no $(partition(g), t'')$ between the two handle actions, then $t < t'$.*

Lemma 6. *All admissible timed traces of the system are in the set $ttraces(Order)$.*

Proof. We assume $\alpha \rightarrow \beta$, and take an arbitrary timed trace γ belonging to admissible timed traces of the system such that $(handle(\alpha), t)$ and $(handle(\beta), t')$ appear in γ and no partition occurs in between them. We are going to show that $t < t'$, thus γ belongs to $ttraces(Order)$. The proof strategy is to assume $t' < t$ and prove contradiction.

By the precondition of $(handle(\beta), t')$ we know that α cannot be in *ops* at time t' (see the **Internal** action in M). Moreover, we know that α must be in *ops* at time t because $(handle(\alpha), t)$ requires it. Thus, α must be added to *ops* between these two time points and the only action that can add operations to this set is $receive(\langle \text{“log”}, \ldots \rangle)_{rM}$. Hence there is a time point t^l at which $(receive(\langle \text{“log”}, \langle \ldots, \alpha, \ldots \rangle \rangle)_{rM}, t^l)$ appears in γ and

$$t' < t^l < t \tag{5.5}$$

Next, consider a sequence of actions that must all be in γ with $t^0 < t^1 < \ldots < t_8 < t'$.

1. $(handle((\beta), t')$
2. $(receive(\langle \text{“log”}, \langle \ldots, \beta, \ldots \rangle \rangle, t_8)_{r_1 M}$ for some r_1
3. $(send(\langle \text{“log”}, \langle \ldots, \beta, \ldots \rangle \rangle, t_7)_{r_1 M}$
4. $(receive(\langle \text{“invoke”}, \beta \rangle, t_6)_{cr_1}$ for some c
5. $(send(\langle \text{“invoke”}, \beta \rangle, t_5)_{cr_1}$
6. $(receive(\langle \text{“reply”}, \alpha \rangle, t_4)_{cr_2}$ for some r_2
7. $(send(\langle \text{“reply”}, \alpha \rangle, t_3)_{r_2 c}$

8. $(receive(\langle \text{“logAck”}, \langle \ldots, \alpha, \ldots \rangle \rangle, t_2)_{Mr_2}$
9. $(send(\langle \text{“logAck”}, \langle \ldots, \alpha, \ldots \rangle \rangle, t_1)_{Mr_2}$
10. $(receive(\langle \text{“log”}, \langle \ldots, \alpha, \ldots \rangle \rangle, t^0)_{r_2M}$

We show that the presence of each of these actions requires the presence of the next action in the list above (which is preceding in time).

- (1⇒2) is given by the fact that β must be in *ops* and that $(receive(\langle \text{“log”}, \langle \ldots, \beta, \ldots \rangle \rangle, t_8)_{r_1M}$ is the only action that adds operations to *ops*.
- (2⇒3), (4⇒5), (6⇒7) and (8⇒9) are guaranteed by the network (N1).
- (3⇒4) is guaranteed since β being in $L = \langle \ldots, \beta, \ldots \rangle$ at r_1 implies that some earlier action has added β to L and $(receive(\langle \text{“invoke”}, \beta \rangle, t_6)_{cr_1}$ is the only action that adds elements to L at r_1.
- (5⇒6) is guaranteed by C2 together with the fact that $\alpha \rightarrow \beta$.
- (7⇒8) Due to 7 α must be in *toReply* at r_2 at time t_3. There are two actions that set *toReply*: one under the normal/degraded mode, and one upon receiving a “logAck” message from the manager M.

  First, we show that r_2 cannot be added to *toReply* as a result of $receive(\langle \text{“invoke”}, \alpha \rangle)_{cr_2}$ in normal mode. Since α is being replayed by the manager ($(handle(\alpha), t)$ appears in γ) then there must be a partition between applying α and replaying α. However, no operation that is applied in normal mode will reach the reconciliation process M as we have assumed (A2) that the replica logs are empty at the time of a partition. And since α belongs to *ops* in M at time t, it cannot have been applied during normal mode.

  Second, we show that r_2 cannot be added to *toReply* as a result of $receive(\langle \text{“invoke”}, \alpha \rangle)_{cr_2}$ in degraded mode. If α was added to *toReply* in degraded mode then the log in the partition to which r_2 belongs would be received by M shortly after reunification (that precedes handle operations). But we have earlier established that $\alpha \notin ops$ at t', and hence α cannot have been applied in degraded mode. Thus α is added to *toReply* as a result of a receive “logAck” action and (7⇒8).
- (9⇒10) is guaranteed since α must be in $ackset[r_2]$ and it can only be put there by $(receive(\langle \text{“log”}, \langle \ldots, \alpha, \ldots \rangle \rangle, t^0)_{r_2M}$

We have in (5.5) established that the received log message that includes α appeared in γ at time point t^l, $t' < t^l$. This contradicts that $t^0 = t^l < t'$, and concludes the proof. □

Now we are ready to state and prove the partial correctness theorem for the CS protocol.

Definition 17. *Let the set ttraces(Correct) be the set of action sequences with monotonically increasing times such that if* $(broadcast(\langle \text{“install”}, p\rangle)_M, t^{\text{inst}})$ *is in the sequence, then p is consistent according to Definition 8.*

Theorem 6. *All admissible timed executions of the system are in the set ttraces(Correct).*

Proof. Consider an arbitrary element σ in the set of admissible timed system traces. We will show that σ is a member of the set *ttraces*$(Correct)$. The strategy of the proof is to analyse the subtraces of σ that correspond to actions of each component of the system. In particular, the sequence corresponding to actions in the reconciliation manager M will be of interest.

Let γ be the sequence that contains all actions of σ that are also actions of the reconciliation manager M ($\gamma = \sigma|M$). It is trivial that for all processes $i \neq M$ it holds that $\sigma|i \in ttraces(Correct)$ as there are no install messages broadcasted by any other process. Therefore, if we show that γ is a member of *ttraces*$(Correct)$ then σ will also be a member of *ttraces*$(Correct)$.

We will proceed to show that γ is a member of *ttraces*$(Correct)$ by performing induction on the number of actions in γ.

Base case: Let p be the partition state before the first action in γ. The model of the reconciliation manager M initialises p to $\{(\langle\rangle, s_1^0, s_1^0), \ldots, (\langle\rangle, s_N^0, s_N^0)\}$. Therefore, requirements 1,2 and 4 of Definition 8 are vacuously true and 3 is given by A1.

Inductive step: Assume that the partition state resulting from action i in γ is consistent. We will then show that the partition state resulting from action $i+1$ in γ is consistent. It is clear that the model of the reconciliation manager M does not affect the partition state except when actions $reunify(g)_M$ and $handle(\alpha)$ are taken. Thus, no other actions need to be considered. We show that *reunify* and *handle* preserve consistency of the partition state.

The action $(reunify(g)_M, t)$ sets p to the initial value of p which has been shown to be consistent in the base case.

The action $(handle(\alpha), t)$ is the interesting action in terms of consistency for p. We will consider two cases based on whether applying α results in an inconsistent state or not. Let p^i be the partition state after action i has been taken.

(1) If $\text{apply}(\alpha, p^i)$ is not constraint consistent then the if-statement in the action handle is false and the partition state will remain unchanged, and thus consistent after action $i+1$ according to the inductive assumption.

(2) If $\text{apply}(\alpha, p^i)$ is constraint consistent then the partition state p^{i+1} will be set to $\text{apply}(\alpha, p^i)$. By the inductive assumption there exists a

sequence L leading to p^i. We will show that the sequence $L' = L + \langle\alpha\rangle$ satisfies the requirements for p^{i+1} to be consistent.

Consider the conditions 1-4 in the definition of consistent partition (Def. 8).

1. By the definition of apply we know that all replicas in p remain unchanged except one which we denote r. So for all replicas $\langle L_j, s_j^0, s_j\rangle \neq r$ we know that $\beta \in L_j \Rightarrow \beta \in L \Rightarrow \beta \in L'$. Moreover the new log of replica r will be the same as the old log with the addition of operation α. And since all elements of the old log for r are in L, they are also in L'. Finally, since α is in L' then all operations for the log of r leading to p^{i+1} are in L'.
2. Consider the last state $\mathbf{s}^k = \langle s_1, \ldots, s_j, \ldots s_N\rangle$ where s_j is the state of the replica that will be changed by applying α. Let s'_j be the state of this replica in p^{i+1} which is the result of the transition $s_j \overset{\alpha}{\rightsquigarrow} s'_j$. By the inductive assumption we have that $\mathbf{s}^0 \overset{\alpha_1}{\rightsquigarrow} \ldots \overset{\alpha_k}{\rightsquigarrow} \mathbf{s}^k$. Then $\mathbf{s}^0 \overset{\alpha_1}{\rightsquigarrow} \ldots \overset{\alpha_k}{\rightsquigarrow} \mathbf{s}^k \overset{\alpha}{\rightsquigarrow} \mathbf{s}^{k+1}$ where $\mathbf{s}^{k+1} = \langle s_1, \ldots, s'_i, \ldots s_N\rangle$ is a partition transition according to Definition 5.
3. By the inductive assumption we know that p^i is consistent and therefore $\forall j \leq k\ \mathbf{s}^j$ is constraint consistent. Further since $\text{apply}(\alpha, p^i)$ is constraint consistent according to (2), $\mathbf{s}^{k+1}$ is constraint consistent.
4. The order holds for L according to the inductive assumption. Let t be the point for $handle(\beta)$ in γ. For the order to hold for L' we need to show that $\alpha \nrightarrow \beta$ for all operations β in L. Since β appears in L there must exist a $handle(\beta)$ at some time point t' in γ. Then according to Lemma 6 $\alpha \nrightarrow \beta$ (since if $\alpha \rightarrow \beta$ then $t < t'$ and obviously $t < t'$).

□

Having formally proved the correctness and termination of all the presented algorithms, in the next chapters we treat implementation and performance issues in more detail.

"The proverbial German phenomenon of the verb-at-the-end about which droll tales of absentminded professors who would begin a sentence, ramble on for an entire lecture, and then finish up by rattling off a string of verbs by which their audience, for whom the stack had long since lost its coherence, would be totally nonplussed, are told, is an excellent example of linguistic recursion."

Douglas Hofstadter

6 CS Implementation

In Chapter 4 we gave a formal description of the CS reconciliation protocol. This allows the protocol to be conceptually understood and its properties formally shown. However, it does not provide much information on how such a protocol could be implemented in a real middleware. Therefore, we will in this chapter describe two implementations that we have done: a simulated environment based on J-Sim [53], and a component in a CORBA-based middleware for fault tolerance called DeDiSys [31].

The reason that we chose to make two implementations is that they provide different means of evaluation. The simulation environment allows important system parameters to be changed, so that the behaviour of the protocol can be investigated under different conditions. In the CORBA implementation the protocol is tested on a real system, and provides more realistic measurements results.

We chose J-Sim as the simulator platform since it is a component-based simulation environment that provides event-based simulation. Moreover, it is built with Java and uses TCL as glue code to control simulations. The same Java code could therefore be used in both the simulation and later in the CORBA middleware.

6.1 DeDiSys

The DeDiSys project aims at designing and implementing dependable middlewares. Although the project contained a track where fault tolerance for service-centric applications is investigated, its main focus is on data-centric

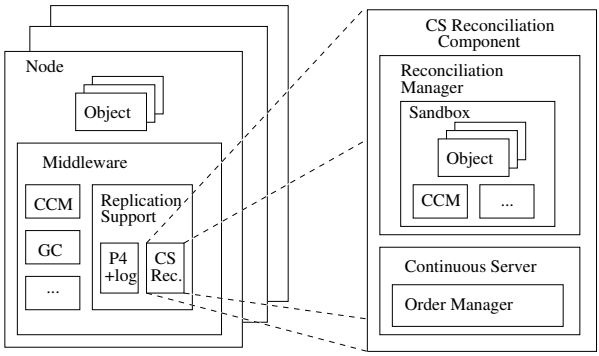

Figure 6.1: CS overview

systems. In particular, the goal is to construct a partition-tolerant middleware that could deal properly with application defined integrity constraints.

Three different middleware technologies were used as a basis for creating the fault-tolerant middlewares: CORBA, .Net, and EJB. The same basic design was used on all three platforms, but they differed slightly in focus. The CORBA and .Net platforms were mainly developed as prototypes to evaluate replication and reconciliation protocols, whereas the EJB platform was mainly used as a basis for evaluating constraint handling approaches.

Since there are several nodes involved in the development of each platform, there was considerable effort in integrating the different parts. The end result is three working implementations of the DeDiSys middleware, where the CORBA platform allows the protocols in this thesis to be evaluated. However, most of the design decisions in the common design were not made specifically for the protocols in the thesis. For example, the constraint consistency manager is designed to deal with other types of reconciliation approaches such as application-driven (or manual) reconciliation. This fact is reflected in the following sections, even if we try to focus the presentation to the parts that are relevant for our current discussion.

6.2 Overview

In the description of the protocol it was divided in two parts, the manager and the continuous servers. Here, we give a more detailed description of how the components can be built from smaller components and how they interact with the rest of the system. For a longer discussion on the DeDiSys architecture and with a slightly different focus see [73].

In Figure 6.1 an overview of the middleware architecture is shown. Each node contains the middleware and a number of application objects. The middleware is composed of a number of services, of which the replication support is the focus in this presentation. This component is, in turn, composed of a

replication protocol, and a reconciliation protocol, i.e., the continuous service (CS) protocol. These protocols rely on additional middleware services such as Group Communication (GC) and Constraint Consistency Manager (CCM). The CCM is used to check consistency of integrity constraints. The box with "..." is an abstraction of other services in the middleware not relevant for this presentation.

6.3 Group Membership and Group Communication

Since we rely on group membership and group communication services both in the specification of our protocol and in the assumptions on the replication services, we need a component providing this in our implementation.

In the simulated platform, we simply integrated group membership service in the location service. Thus, network partition faults could be injected by simply telling the location service to divide the network on a logical level. All communication between nodes were performed using the J-Sim ports that are asynchronous message passing primitives. The ports were interconnected via a simulated network component. This component added network delays and provided multicasting.

In the CORBA platform, a real group membership service is required. We chose to use Spread [87] for this since it has a Java binding and provides the needed group membership and group communication services. All communication between the reconciliation manager and continuous servers as discussed in the previous chapter are performed using spread messages.

6.4 Replication Support

As we have already noted, the CS reconciliation protocol is really part of an optimistic replication approach. So far, we have not described any replication mechanism in detail. We proceed by describing two of the replication protocols that have been developed in the DeDiSys project: P4, and P4Log that builds upon P4 but is compatible with the CS reconciliation protocol. In the following subsection we will describe the implementation of various components for replication support both in the simulated environment and the CORBA version.

6.4.1 Replication Protocols

P4 The Primary Per Partition Protocol (P4) has been designed by Beyer et al. [12]. The protocol allows consistency to be temporarily relaxed during degraded mode, by allowing operations with non-critical integrity constraints. Moreover, the protocol also allows operations with critical con-

straints given that no involved object has been changed in a way that might be reversed.

During normal mode, the protocol ensures one-copy-serializability by redirecting all update operations to the primary replica. If the update succeeds, then the new state of the primary is propagated to the secondary replicas.

The P4 protocol was originally designed with two in-built reconciliation protocols, one automatic, and one manual. The automatic reconciliation protocol requires all intermediate states to be saved. When the system reunifies, the protocol chooses a set of primaries from which to construct the new state. Since these primaries may come from different partitions, the resulting state is not guaranteed to be consistent. Thus, the protocol reverts to an older version for some object and checks again if the state is consistent. This reconciliation process is not guaranteed to end until all changes have been reversed. Moreover, since it requires that all intermediate states are saved, it is not a feasible to implement.

The manual protocol uses application callbacks to allow the application writer to construct rules that decide the new state. The drawback with this approach is that the application writer needs to figure out how the application is to reconcile a state that it has no knowledge of how it was constructed.

P4Log The P4 protocol is nearly compatible with the CS reconciliation protocol, but there are two things that make them incompatible. Therefore, we have designed a replication protocol P4Log that builds upon P4, but is compatible with CS. First of all, the CS protocol requires a log of all the update operations that have occurred during the degraded mode. Thus, P4Log includes a logging mechanism described below. Secondly, P4 allows operations with critical constraints during degraded mode for some special circumstances. In order to guarantee that these operations are not revoked during reconciliation one can add an ordering requirement: all critical operations and operations that are ordered before a critical operation must be replayed before any non-critical operation. This is similar to the way Bayou treats committed operations at a primary server. The idea is illustrated in Figure 6.2, where all the operations within the dashed lines must be replayed before the other operations. By using this extension one would achieve slightly higher operation-based availability during degraded mode. However, since accepting critical constraints first would affect the revocations of non-critical constraints, it is not clear if the overall number of accepted operations would increase with this approach. In our implementation we have chosen to disallow all critical operation during degraded mode, and thereby side-stepping this problem.

In order to perform the reconciliation, the CS protocol needs to know what the state was before the partition fault occurred. This can be accomplished by checkpointing the state of any object in degraded mode, be-

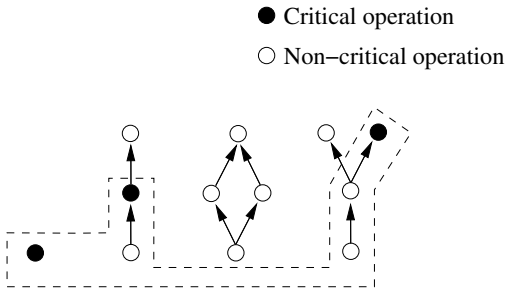

Figure 6.2: Replay order required for P4

fore performing an operation that changes its state. Moreover, the logs of each replica since the last checkpoint are needed to merge the partitions. Therefore, a logging and checkpointing service is needed. The interface in Figure 6.3 describes the logging service that is required. Each time an operation is successfully invoked, the logOperation method is called to store the operation. All objects for which logging is performed must be registered using registerObject in order to facilitate checkpointing. This method also saves the state of the object.

In order to perform the reconciliation, the CS protocol needs to fetch the last consistent state of the system. This is done using the method getLastCheckpoint. Finally, when the reconciliation manager requests all logs that have been constructed during the degraded mode the method getLogSinceCheckpoint is called to fetch a list of applied operations.

```
interface LoggingServiceInterface {
   void logOperation(Request r, ObjectRef logRef);
   void registerObject(ObjectRef logRef, byte[] state);
   byte[] getLastCheckpoint(ObjectRef o);
   List<Request> getLogSinceCheckpoint(ObjectRef o);
}
```

Figure 6.3: Logging Service

6.4.2 Replication Protocol Implementation

Simulation Since the goal of the simulation environment was to investigate the properties of the reconciliation activity, we chose to use a very simple variant of the P4Log protocol. It was integrated in the invocation service component together with the continuous server. Logging and constraint checking was performed as usual, but the state changes were never replicated to the secondary replicas.

CORBA The replication service in the CORBA implementation is composed of a replication manager, a replication protocol (P4Log) and a reconciliation protocol (CS) as shown in Figure 6.4. The figure also shows the control flow of a client invocation. The invocation is intercepted in a CORBA interceptor that passes the control to the replication manager. The replication manager just forwards the request to the replication protocol. The replication protocol can then decide if the request should be forwarded to the primary replica or to some replica that temporarily acts as primary during a network partition. Moreover, the object is registered as a transactional resource and the preconditions are checked with the CCM. The request is also passed on to the CS protocol that may perform a checkpoint on the target object if the operation is the first to be performed on that object in the degraded mode.

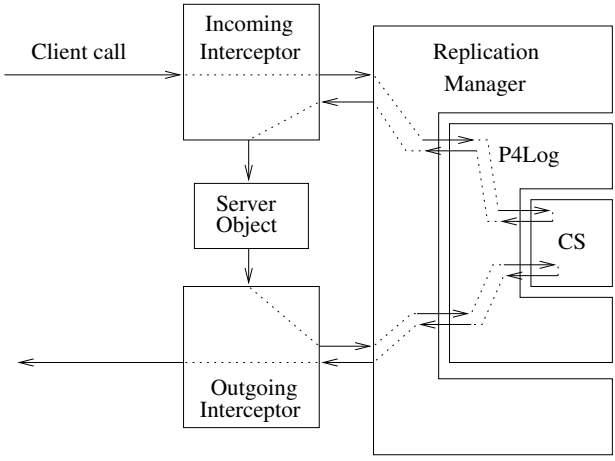

Figure 6.4: CORBA invocation

When the operation has been applied by the server object, it is once again passed through all the replication components. This time to perform logging (and possible interaction with the reconciliation manager) and update propagation. Updates are propagated via spread messages in a synchronous fashion.

The replication manager is responsible for keeping track of the system modes that were illustrated in Figure 4.2 The manager is registered as a listener to events from Spread. When the changes in the network occur, the replication manager determines if the replication protocol should be notified that it should enter degraded mode, or in the case of a join, it should commence the reconciliation process.

6.5 Constraint Consistency Manager

The Constraint Consistency Manager (CCM) is responsible for checking constraint consistency during operation invocations. Moreover, the reconciliation protocol needs to use the CCM while restoring consistency. The constraints managed by the CCM can be classified in different ways. Remember that there are critical and non-critical constraints, but one can also categorise the constraints according to when they are checked as. That is, , pre- and post-conditions, and invariants. Depending on the type of constraint and whether the constraint is associated with a given object, or a method or a class, the CCM should correctly evaluate the appropriate constraints.

Simulation In the simulation, only postcondition constraints were used, and all constraints were checked for all operations. The constraint consistency manager was implemented a store of constraints that were called upon after each operation invocation.

CORBA The CORBA implementation of the CCM provides a more general support for constraint handling than is used by the P4Log/CS protocols. The basic components are shown in Figure 6.5.

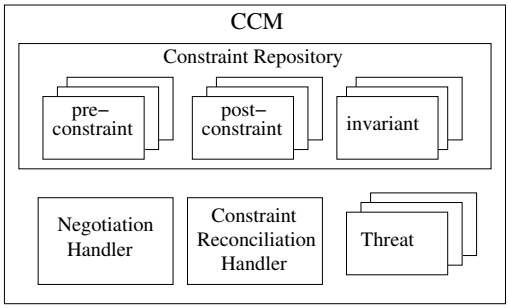

Figure 6.5: CORBA CCM

The component contains a constraint repository in which all constraints are stored. Since there is one CCM for each node, these constraints must be propagated during the system initialisation. The three basic types of constraints (i.e., pre- and post-conditions and invariants) are handled appropriately. Moreover, for each constraint additional information such as affected objects and methods are stored. This way, the CCM only needs to check a small subset of the constraint for every method invocations, thereby reducing unnecessary computations. If a constraint is violated, or it is accepted but the constraint is critical, then the CCM throws a CORBA exception. This is propagated back to the replication protocol that aborts the current

transaction, and to the client informing that the operation could not be applied.

When an operation with a non-critical operation is invoked during degraded mode, it is not necessarily a good idea to accept it. Imagine, a booking system where there is a non-critical constraint saying that the number of bookings should not exceed the number of seats. If there are many free seats, then it is probably safe to make a booking. But if the number of booked seats are close to the limit, then the risk of violation is high. Therefore, it might not worth the risk to accept any more bookings for this flight. This type of reasoning can be done as part of the constraint management and is localised in the Negotiation Handler subcomponent. The implementation allows the application to register an application specific negotiation handler.

The CORBA CCM implementation also supports manual reconciliation by allowing the application to register a constraint reconciliation handler. This is needed by the P4 protocol when performing manual reconciliation. Finally the CCM can store *threats*. A threat is generated every time an operation is provisionally accepted. The term threat refers to that this action is a potential threat to consistency. A state-based reconciliation algorithm may then choose to take these threats into consideration when restoring consistency as it provides similar information as storing operation logs.

6.6 Ordering

Since the CS reconciliation protocol needs to keep track of the order in which operations are to be replayed, a mechanism is needed that makes the induced ordering explicit.

In the simulation implementation, operations were simply ordered by their id numbers. This order can be seen as an approximation of the client expected order described in Section 3.4, which is also used in the correctness proof. In the CORBA implementation we have implemented the Client Expected Order as described below.

This ordering can easily be implemented by tagging each operation at the client side with the operations that must be executed before it. Since the before operations are those that the client has received a reply for, we must keep track of operation invocations as well as replies. We achieve this by introducing an Order Manager that is called from the interceptors at the client side. That is, there is an order manager at each client responsible for creating the operation tags for each operation. It must also be called each time a reply is received to compute the tags correctly.

Algorithm 6 shows the pseudo code for the Order manager. When a request is intercepted on its way to the server the OrderManager is called with TAGOPERATION to get a tag to append to the request. The tag contains a unique id (composed of the client id *cid* and a sequence number *seq*) for the operation (CORBA ids are only unique for requests in progress), and information of order dependency. When the request has been processed by

Algorithm 6 Order Manager

States: $seq \leftarrow 0$ /* Sequence number */
$replies \leftarrow \emptyset$ /* Recently received replies */
cid /* Globally unique client id */

tagOperation($\langle \alpha \rangle$)
1 $seq \leftarrow seq + 1$
2 $id \leftarrow \langle cid, seq \rangle$
3 **return** $\langle \alpha, id, replies \rangle$

registerReply($\langle \alpha, id, beforeSet \rangle$)
1 $replies \leftarrow replies \cup \{id\} \setminus beforeSet$

the server and a reply is intercepted then the OrderManager is informed with the registerReply operation. Note that the OrderManager only needs to maintain a local state for each client. Therefore, no communication is required between different managers. The state contains information on sent and received messages for one particular client.

6.7 Sandbox

In order to successfully replay operations that have been invoked in a middleware environment, we need to mimic this environment in a sandbox. For the application that runs inside the sandbox it appears as the real environment but no changes in the sandbox are committed until the end of the reconciliation phase. In this section we describe the design of the sandbox environment. Its principal interface is shown in Figure 6.6.

```
interface SandboxInvocationServiceInterface {
   void registerObj(byte[] state, String ref, String class);
   void invoke(Request request);
   byte[] getObjState(String reference);
}
```

Figure 6.6: Sandbox Invocation Service

All objects that are to be involved in the reconciliation are registered using the registerObject method. The arguments passed to this method is the state of the object, the object reference and the class name. The latter is needed to create new instances of the object inside the sandbox environment.

During the reconciliation phase the reconciliation manager repeatedly invokes a request in its operation log using the invoke method. The sand-

box is responsible for checking consistency constraints and an operation is applied to the internal state only if all constraints are satisfied. When the reconciliation manager has finished invoking requests it will fetch the resulting states from the sandbox using the getObjectState call with the reference for each object to be fetched.

Simulation In the simulation environment, the sandbox environment was implemented with a single class that held all the application objects in a hash table. The invocations were dispatched to the correct object using Java reflection mechanisms.

CORBA In the CORBA implementation, the sandbox is initialised with its own ORB and interceptors were used to check integrity constraints in the same way as in the normal system. However, the mechanisms in the sandbox interceptors are much simpler since everything is performed on one node and information does not need to be propagated throughout the system. A simple transaction mechanism was used so that operations could be aborted if its integrity constraints were violated.

6.8 Test application

To test the algorithms and the middleware implementation we needed a test application. We chose to construct a synthetic application that was well suited to perform trade-off studies. The application is composed of a set of real or integer number objects. Possible operations are addition, subtraction, multiplication, division, and *setValue*. An operation is applied to the current value with a random constant. There are also integrity constraints in the system expressed as: $n_1 + c < n_2$ where n_1 and n_2 are object values and c is a constant. In a sense this application resembles a distributed sensor network where the numbers correspond to sensors values.

Although the application is very simple, it is complex enough to give an indication of how the algorithms perform. One of the properties is that even if the current state is known, it is hard to predict how the execution of a certain operation will affect the consistency two or three operations in the future. This makes the reconciliation realistic in the sense that it is not trivial to construct a consistent state while keeping as many performed operations as possible. Moreover, the application allows key system parameters (e.g. ratio of critical constraints and load) to be changed and thus provides means to experimentally investigate their effect on availability.

The deployment of the application were slightly different in the different experiments. The details are provided in the next chapter. In all of the experiments the application starts with initialising a number of servers with some default value. A set of constraints are created and added to the constraint store. The constraint parameters are chosen randomly with a

uniform distribution. However, all constraints are created so that the initial state of the system is consistent.

"A computer once beat me at chess, but it was no match for me at kick boxing."

Emo Philips

7 Evaluation

We have proposed a number of algorithms for performing reconciliation after network partitions and we have showed them to be correct. However, to really show their usefulness we have also tried to quantify the costs and benefits that come with our approach. In this chapter we present the result of several experimental evaluations that have been performed on the reconciliation algorithms. First, we propose a number of metrics that are suitable for evaluation of such algorithms and explain why the traditional availability metrics are not enough by themselves. Then we relate the results of some initial simulation studies that were done to compare the performance of the Choose1, STW-Merge and STW-GEU algorithms (Section 7.2). These studies pointed out the benefit of operation-based reconciliation, which then led to the development and evaluation of the CS algorithm. This algorithm was evaluated using both a simulation setup based on J-Sim (Section 7.3) and the CORBA-based platform (Section 7.4).

7.1 Performance metrics

Before presenting the results of the experiments we need to explain the metrics that we have used to measure the performance. The reason that we need to introduce new metrics is because of the fact that our approach requires some operations to be revoked. Whereas the definitions in Section 2.1.1 use the term *operational* to characterise system availability, we will introduce two concepts namely *partially operational* and *apparently operational* that applies to optimistic systems.

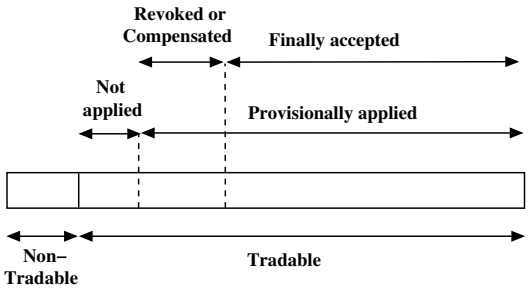

Figure 7.1: The set of operations during a system partition and subsets thereof

Partially operational Figure 7.1 shows a categorisation of the operations that are accepted during degraded mode in a system with integrity constraints that act optimistically. We see that the operations are first of all divided in tradable and non-tradable operations. Non-tradable operations are those operations with critical constraints (i.e., constraints that cannot be violated under any circumstances), whereas the tradable operations are those that we can apply optimistically. In the absence of network partitions, the system can guarantee full consistency and therefore non-tradable operations can be applied in normal mode but not in degraded mode. So it is reasonable to claim that the system is only partially operational during degraded mode.

Note that this term should be used also for the availability achieved by the primary partition approach and quorum consensus. In such systems it is only a subset of the client requests that can be served depending on what is the type of operation (read or write) and to what node the request is directed. Here, it is the type of integrity constraints that defines whether the operation is tried or not.

Apparently operational Of the tradable operations, there are some operations that are not applied due to violation of integrity constraints (as can be seen in Figure 7.1). These operations would have been rejected also during normal mode as part of the normal operation of the system. However, of the operations that are accepted not all of them will be *finally accepted*, which leads us to the concept of apparently operational. It appears to the clients during degraded mode as if the system is operational. However, due to the integrity constraints, there will be operations that must be revoked or compensated in some way during the reconciliation process.

We now proceed with describing the metrics that we have used in our evaluations. The metrics can be divided in two categories; time-based and operation-based metrics. The first category is typically based on the time taken in some segment of the system life time. The second category is based on the counting of the operations that pass through the system and are

treated in one way or the other (subsets from Figure 7.1).

7.1.1 Time-based metrics

We consider the following metrics:

- Apparent availability: Probability of partial/apparent operation at a time point; that is, the average interval that the network is in partial/fully available mode divided by the length of the experiments.
- Average latency: computed over all operations (during normal respectively degraded modes).
- Average time spent in revoking (compensating) one operation.
- Average duration of reconciliation operation.

From the above list, we are going to use the availability metric as a measure for improved performance. However, we need to complement this metric with other measurements in order to identify the substance of improvement (i.e., excluding the apparent availability).

The average latency is interesting if we run the application on top of a standard middleware; in particular, as an input to the computation of the overhead for the middleware. Therefore this has only been measured in the CORBA implementation of the CS protocol.

The time taken to revoke or compensate one operation depends on what the application requirements are. It is interesting since it very much affects the reconciliation time. In our experimental setting, we choose to compute this time based on an estimate of an undo-time per operation, thus turning it into a parameter.

Finally, the time taken for reconciliation is interesting for two reasons. First of all, for the stop-the-world protocols, this is a period in which the system is unavailable, and thus can be seen as an indication of availability. Secondly, even for the continuous service protocol it is interesting to see how long it takes before the system can go back to normal mode. Because even if service is maintained during the reconciliation period by the CS protocol, the system is only partially operational since non-tradable operations are not accepted. Moreover operations accepted during the reconciliation phase can also be revoked in the same way as operation performed in the degraded mode.

7.1.2 Operation-based metrics

As mentioned above measurements of apparent availability are only meaningful if they are presented together with an indication of the "loss" from revoked operations. To be specific about the level of service delivered to clients we propose a number of operation-based metrics:

- The number of finally accepted operations during the whole experimental interval.
- The proportion of revocations over provisionally accepted operations.
- The accrued utility

The number of finally accepted operations is not an indication of availability by itself. There is little information in saying that a given system has accepted X many operations during a certain time period. However, when compared with another approach such as the pessimistic approach the *increase* in number of accepted operations is meaningful. This tells us if the system has increased its availability. Unfortunately there is no easy way of characterising the maximum number of accepted operations. One way would be to compare with the total number of submitted operations. However, as we explained earlier, all operations are not accepted even when there are no faults.

A very interesting metric is the proportion of revocations over provisionally accepted operations. This depends on the number operations that the client thinks have been performed but which must be revoked/compensated. It is heavily dependent on the application semantics (i.e., the type of constraints). For an application with a high proportion of revocations, it should not pay off to act optimistically during network partitions. However, the revocation ratio can be reduced by employing algorithms such as the STW-GEU. Moreover, this metric is related to, but should not be confused with, the collision probability calculated by Grey et al. [44] to be proportional to the square of the number of operations. Wang et al. [95] have investigated the conflict rate for file systems. Common for these two metrics is that they consider two replicas to be in conflict if they have been updated concurrently. In our model, on the other hand, a conflict occurs only as the result of the violation of some integrity constraint. Such violations can be caused by concurrent updates, but not necessarily.

We have introduced utility as a metric to compare the usefulness for each operation and we can therefore talk about the accrued utility during a reconciliation period as the accumulated utility from the applied operations minus the utility of the revoked operations. The utility of an operation can be dependent on a number of factors, the lost benefit if it is not applied, or the economic gain for a business that gets paid by its customers for delivered service. Also, the time taken to execute an operation is important if we are considering undoing/redoing the operation; thus, the higher the execution time the lower utility. However, we will in our experiments let the utility be a random number, which is assigned to each operation.

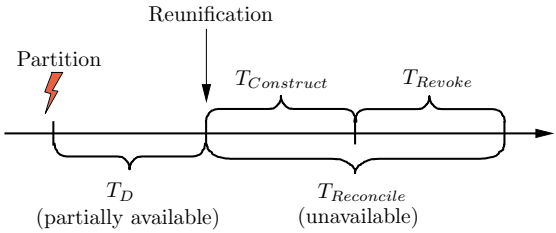

Figure 7.2: Time for reconciliation

7.2 Evaluation of Reconciliation Approaches

This section contains an experimental evaluation of the three centralised algorithms, Choose1, STW-Merge and STW-GEU. These all require that the system does not accept new updates during the reconciliation process. So all the time spent performing reconciliation will reduce the availability. The time that the system is completely unavailable is time taken to reconcile the system, $T_{Unavailable} = T_{Reconcile}$. Note that the reconciliation time $T_{Reconcile}$ can also be divided in two parts, $T_{Construct}$, which is the time to construct a new partition state and T_{Revoke}, which is the time taken to perform all the necessary actions for the operations that have been revoked. This revocation or compensation mechanism potentially involves sending a notification to the client, which could then try to redo the operation. So $T_{Reconcile} = T_{Construct} + T_{Revoke}$. Figure 7.2 illustrates these time periods. If no notification of the clients are necessary or revoking an operation for some other reason takes little time to perform then there is little use in spending time constructing an elaborate partition state. If, on the other hand, the undo operation is expensive in time then minimising T_{Revoke} is the main goal.

With the experimental evaluation we aim to demonstrate that for short time to repair and a short revocation time for operations the Choose1 algorithm is good enough. Since the system does not stay long in degraded mode there are not many operations lining up and therefore the loss of utility is outweighed by the gain of a short construction time $T_{Construct}$. However, we expect the STW-Merge and STW-GEU algorithms to achieve much better utility for longer repair times and thus also shorter reconciliation times. Moreover, we will demonstrate that the key to choosing between STW-Merge and STW-GEU is the time needed to revoke operations. If undoing an operation is time consuming then the STW-GEU algorithm should perform better than STW-Merge as it aims at minimising the number of revoked operations.

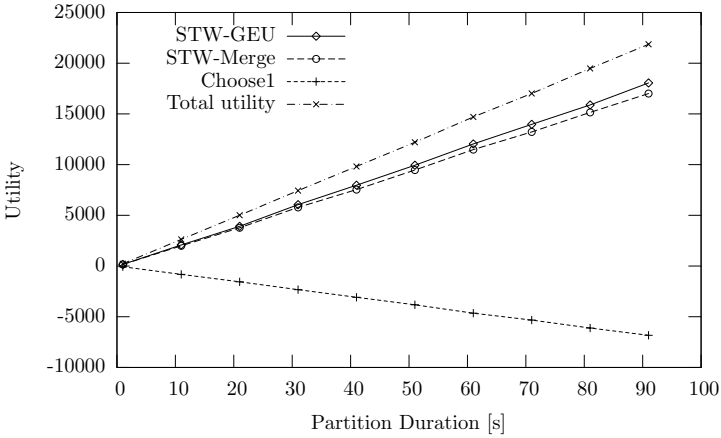

Figure 7.3: Utility vs. Partition Duration [s]

Experimental setup

The application described in Section 6.8 was implemented in Python together with the surrounding framework needed to perform the simulations. Eight objects were initialised with random numbers, uniformly distributed between -100 and 100. The results in this section are based on five integrity constraints created between two randomly chosen objects so that the initial partition state was constraint consistent. [1]

Each application instance was profiled during a phase where 10000 operations were scheduled and statistics could be gathered about whether the operations succeeded or not. This data was then used to calculate P_{viol} for each operation. The system was then split up into three partitions in which operations were scheduled for execution with a load of 10 operations per second. Each object received the same load of requests. For each measurement point 100 samples were obtained and averaged.

Each operation was assigned a utility with a normal distribution of $N(1.0, 0.1)$. The execution times of the algorithms were measured by assigning execution times for different steps of the algorithm, more specifically a step that does not require disk access takes $100ns$, disk operations take $1\mu s$ and the task of revoking an operation takes $1ms$. Constraint checking is a step that is assumed not to require disk access. Revoking an element is assumed to require notifying clients over the network and is therefore far more time consuming than other types of operations.

Results

In Figure 7.3 the utility of the reconciliation process is plotted against the time to repair (that is, the time spent in degraded mode) which in turn decides the number of operations that are queued up during the degraded mode.

It is clear that just choosing the state from one partition will result in a low utility. The operations of all the other partitions will be undone and this will result in a negative utility given that the partitions are approximately of the same size. If the utility of one partition would heavily outweigh the utility of the other partitions (or there would be only two partitions) then the Choose1 algorithm performs significantly better. However, there would have to be a very large imbalance for it to outperform the other algorithms. Further experiments indicate that the gap remains even if the time to revoke for an operation is reduced by a factor of 10. As the STW-Merge algorithm is nondeterministic we have also studied how it behaves over 100 runs with the same input parameters. For a partition duration of 31s the resulting mean utility is 5881.1 and the standard deviation 31.5.

The upper curve shows the total utility which is the sum $\sum_{\alpha \in op} U(\alpha)$ where *op* is the set of all operations appearing in the partition logs. The optimal utility is less than or equal to this sum. For the optimal utility to be equal to the total utility it must be possible to execute the operations in such an order that inconsistencies do not occur. Since operations are chosen randomly this seems very unlikely. The STW-GEU algorithm performs slightly better than the STW-Merge and the difference increases somewhat as the partition duration grows (meaning more operations to reconcile).

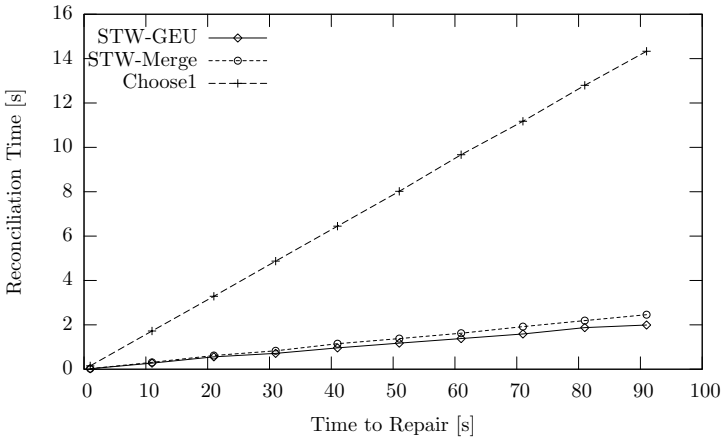

Figure 7.4: Reconciliation Time [s] vs. Partition Duration[s]

[1]Simulations have also been performed with more objects and constraints with similar results.

In Figure 7.4 the reconciliation times of the algorithms (including the time needed to deal with the revoked operations) are compared. The graphs illustrate that the three algorithms execute in linear time over the repair time (the longer the duration of partition the higher the number of performed operations). Essentially the results follow the same pattern as before. The STW-GEU algorithm performs slightly better that the STW-Merge and both are significantly better than the Choose1 algorithm[2]. As the revocation of operations is the most time consuming task this is not surprising. However, since the STW-Merge and STW-GEU show very similar results we will study in more detail how the cost of revocation affects the performance comparison between the two.

In Figure 7.5 the reconciliation times of STW-Merge and STW-GEU are plotted against the revocation time for one operation. As the STW-Merge algorithm has a shorter construction time $T_{Construct}$ than STW-GEU it will result in a shorter reconciliation time when it is inexpensive to undo operations. When the time to revoke an operation increases the effect of the longer construction time for STW-GEU diminishes and the shorter total revocation time T_{Revoke} results in a total reconciliation time that is shorter.

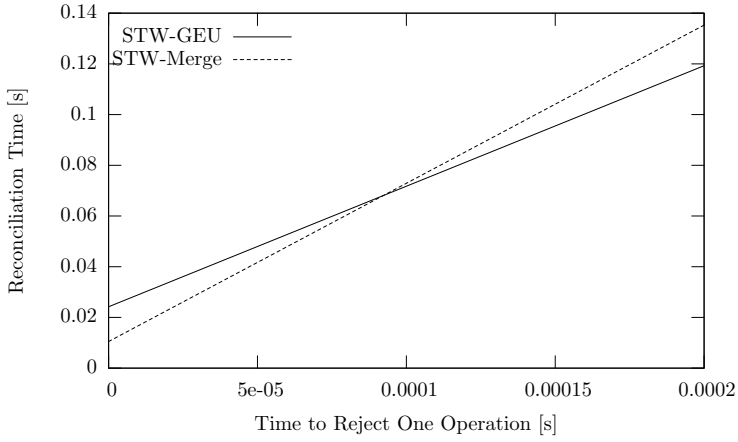

Figure 7.5: Reconciliation Time [s] vs. Time to Revoke One Operation [s]

7.3 Simulation-based Evaluation of CS

In this section we present a simulation-based experimental evaluation of the continuous service reconciliation protocols. The goal of these studies were to determine whether continuous service reconciliation pays off in terms of

[2]The Choose1 algorithm could benefit from parallelising the task of undoing operations.

Table 7.1: Simulation parameters

Number of runs	100
Number of objects	100
Number of constraints	30
Number of critical constraints	10
Simulation time	70 [s]
Number of nodes	50
Number of clients	30
Mean network delay	0.1 [s]
Normal system load	120 [ops/s]

time and operation-based availability, and to study the effects of varying reconciliation rate and load.

As a baseline, we consider two alternatives. First, a system that does not trade availability for consistency (using a pessimistic protocol), and the STW-Merge algorithm that does not accept new operations during reconciliation. The pessimistic protocol only accepts operations during normal mode, and rejects operations if there is a network partition. Note that the pessimistic protocol does not need to have a reconciliation phase, since the state is always kept consistent.

7.3.1 Simulation setup

The simulations were performed with J-Sim [96] using the event-based simulation engine. A simple middleware (see the "Simulation" paragraphs in Chapter 6) was constructed and the test application described in Section 6.8 was implemented on top of it. However, as the main goal of the implementation was to evaluate the reconciliation protocol some parts of the system have been simplified. The group communication component is for example simulated using a network component that also provides a group membership service. This allows fault injection and network delays to be controlled. Faults are injected by configuring the location service component to resolve object lookups in the same way as if there had been a network partition. This is done in the beginning of each simulation run.

The simulation parameters that were constant in all experiments are shown in Table 7.1. We base these figures on data provided by industry partners, with real applications that can benefit from partition tolerance, in the DeDiSys project.

7.3.2 Results

In all of the following curves we compare three different protocol behaviours. All three measurements are performed using the same application and random seeds. Moreover, the middleware implementations only differ in the

places where the replication and reconciliation differ. The first curve ("continuous") in each graph shows a middleware that acts optimistically during the partition fault, and then uses the continuous service reconciliation (CS) protocol to merge the results. The first baseline that does not accept invocations during reconciliation is denoted as "stop-the-world". Finally, the last ("pessimistic") shows the results for a pessimistic middleware which does not accept invocations during a partitioned state.

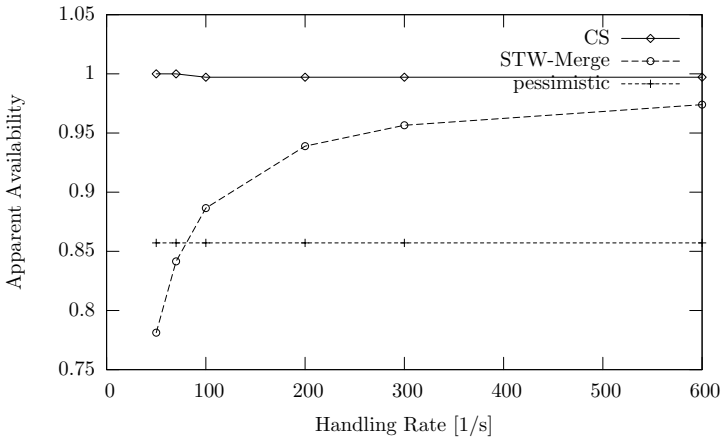

Figure 7.6: Apparent Availability vs. Handling rate

The effect of handling rate In Figure 7.6 the apparent availability is plotted against the handling rate of the reconciliation manager. This rate is an estimate of the average time taken to reconsider a provisionally accepted operation, replay it, and potentially undo it. The partition lasted for 10 seconds in each run. The 95% confidence intervals are within 0.35% for all measurement points. The pessimistic approach gives just over 85% independently of the nature of operations that are potentially revocable (since they are never run). This is natural since no operations are performed during partitions. The continuous service protocol manages to supply nearly full availability except for the small effect given by the time spent installing the new state. However, the availability of "stop-the-world" depends very much on the length of the reconciliation phase, which in turn is decided by how fast the handling rate of the reconciliation manager is.

There is an anomaly for the CS protocol for small handling rates. There is no period of unavailability for these rates. The reason is that the protocol will never reach the stop state during the simulation time, and thus never become unavailable. The termination proof from Chapter 5 gives that a

condition for the termination is

$$H > \left(\frac{T_D + 7d}{T_F - 9d}\right) C \cdot I$$

where H is the (worst case) handling rate, T_D the partition duration, d a bound on message and service time, T_F the time until next fault (in these runs the end of the simulation), C the number of clients, and I the (worst case) invocation rate for each client. If we put the (average) numbers from our simulations in this inequality we find that the handling rate must be at least 137 to *guarantee* termination. In the figure we see that termination actually occurs for rates over 100 (indicated by the fact that the CS protocol drops from full availability to just under 100%).

As the results in Figure 7.6 only give the apparent availability (as discussed in Section 7.1) we need also to compare the second availability metric, which is how many operations we have finally accepted.

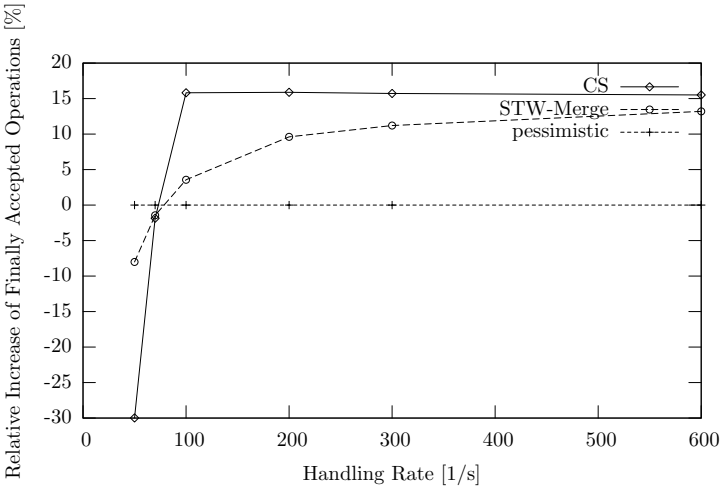

Figure 7.7: Relative Increase of Finally Accepted Operations vs. Handling rate

In Figure 7.7 the relative increase of finally accepted operations compared to the pessimistic approach is plotted against the handling rate. This graph is based on the same experiment as Figure 7.6. The 95% confidence intervals are within 1% for all measurement points. The optimistic approaches achieve better as handling rate increases. For large enough handling rates they give significantly better results compared to the pessimistic approach. The CS reconciliation protocol only gives distinctly better results than "stop the world" for handling between 100 and 300 operations per second. However, as the handling rate increases further the difference becomes marginal.

This plot indicates that an estimate of the average handling rate, based

on profiling the application, is appropriate as a guideline before selecting the CS protocol in a reconfigurable middleware.

The effect of partition duration There are applications, like telecommunication, where partitions do occur but a lot of effort is spent to make them as short as possible so that acting pessimistically will not cause a big decrease in availability. In Figure 7.8 we see the effect that the partition duration has on the apparent availability. For long enough partitions the only approach that gives acceptable results is the continuous service reconciliation. The confidence intervals for this graph are within 0.1% for all measurement points.

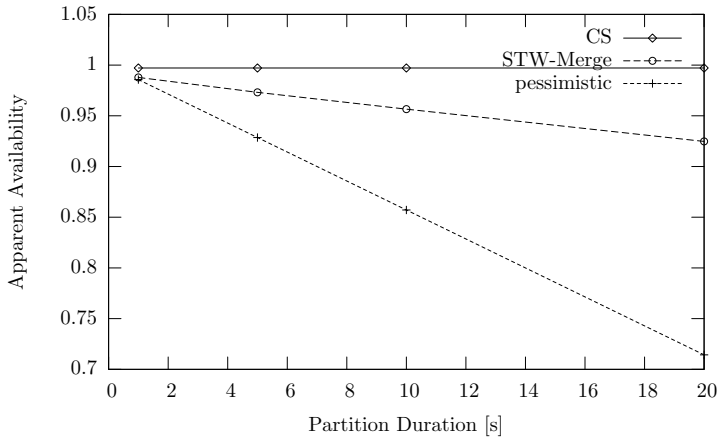

Figure 7.8: Apparent Availability vs. Partition Duration

Both of the optimistic reconciliation protocols considered here are operation-based. That is, they use a log of operations that were performed in the degraded mode. One can also perform state-based reconciliation where only the current state of the partitions is used to construct the new state. A state-based reconciliation scheme might give equally high apparent availability as the continuous service protocol but instead it might suffer in terms of finally accepted operations as shown in Section 7.2 by measuring the accrued utility.

In Figure 7.9 we see that as the partition duration increases the ratio of revoked operations decreases. This is a bit counter-intuitive, one would expect the opposite. However, there is an explanation to this phenomenon. The cause lies in the fact that in our synthetic application two partitions perform similar kinds of client calls. This means that an operation which has been successfully applied in one partition is likely to be compatible with changes that have occurred in the other as well. The longer the partition lasts, the more operations are performed and the risk of different types of

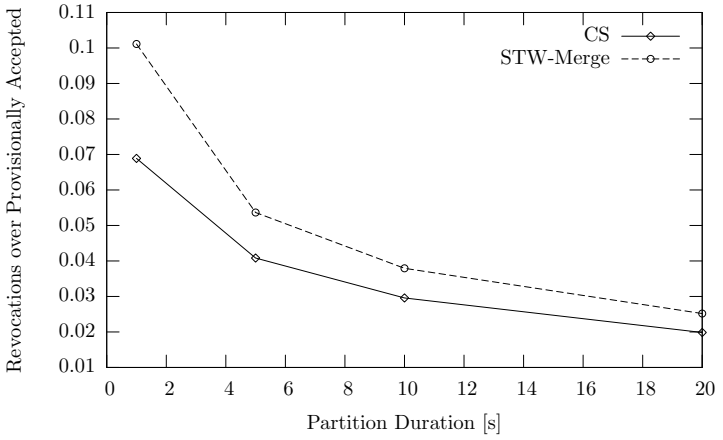

Figure 7.9: Revocations over Provisionally Accepted vs. Partition Duration

operations in the two partitions decreases. Naturally, this behaviour depends on the nature of the integrity constraints and thus on the application. The confidence intervals are within 6.9% for all measurement points.

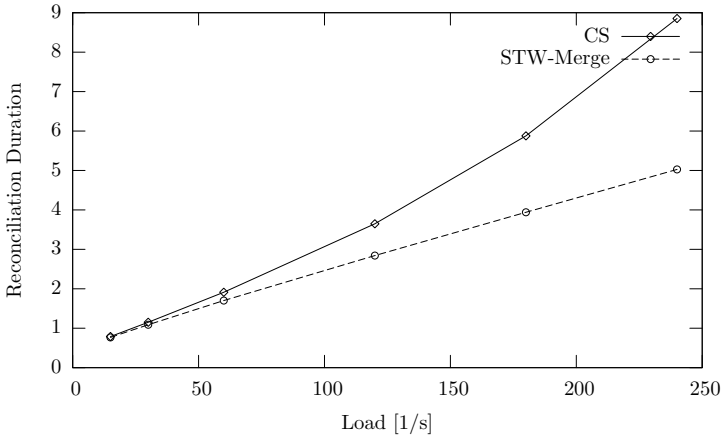

Figure 7.10: Reconciliation Duration vs. Load

The effect of load So far the experiments have been performed with a constant arrival rate of 120 operations per second. To see the effect of load we have plotted the reconciliation duration against load in Figure 7.10. Here, the 95% confidence intervals are within 1.6% for all measurement points. The handling rate for this experiment was 300 actions per second. This figure might seem high compared to the load. However, the reconciliation

process is performed at a single node which means that no network communication is needed. As can be seen in the figure the continuous service protocol suffers more than the other protocols under heavy load; especially, as it approaches the maximum load. However, this does not translate to less apparent availability as in the case of stop-the-world. The only period of unavailability for the CS protocol is during the time between the continuous servers receive a stop message from the reconciliation manager and the time to receive the installed state. The length of this period is not affected by the length of the reconciliation phase. Thus, the apparent availability of CS is not decreased (as was shown in Figure 7.6).

7.4 CORBA-based Evaluation of CS

Having checked the positive potentials of the CS protocol in a simulated setting we then constructed an evaluation environment in order to quantify the overheads (i.e., costs) associated with a fault-tolerant middleware that might utilise this protocol.

7.4.1 Experimental Setup

The CORBA-based partition tolerant middleware was tested and evaluated using two hardware platforms: one with several virtual machines running on one physical node and one composed of a real network of physical machines. The reason for using the setup with virtual machines was that it was easier to setup the experiments and to collect data in case of partial executions. However, for the throughput measurements the virtual machines would not provide reliable results, and therefore the physical machines had to be used. The same middleware from Chapter 6 were used for both platforms. The CORBA ORB implementation was JacORB 2.2.3.

Figure 7.11 shows the framework that was used for performing measurements. The same setup was used in both the setup with the virtual machines as with the physical machines. Four nodes were setup with an host-control CORBA object. This could be remotely controlled from a fifth server that managed the experiments. The host-control object on each machine could start up the DeDiSys middleware, initialise server objects and start up client objects. Each node was running Linux and by using the Iptables firewall (FW) capability in Linux, the host-control object could be used to regulate communication. This allowed network partition fault to be injected. In the figure we illustrate the links that were blocked with dashed lines.

Virtual machines The availability measurements were performed on four virtual VMWare machines running *Ubuntu Linux 6.10*. The host machine was an *Acer TravelMate 8204* with a *Intel DualCore T2500* processor running at 2Gz and 2GB RAM. *Sun Microsystem*'s Java™virtual machine was of version 1.5.0_09.

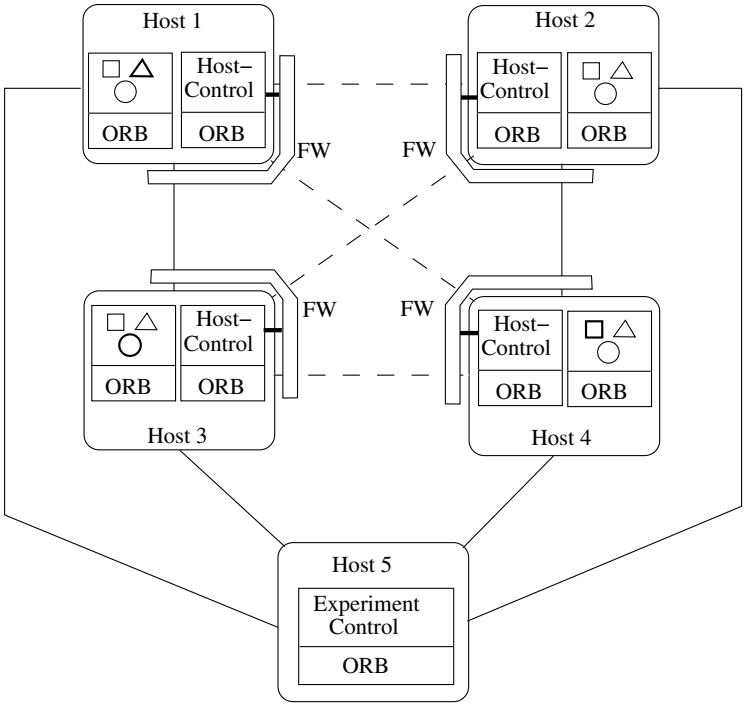

Figure 7.11: Deployment of the test environment.

Physical machines For acquisition of time-related metrics, such as maximum throughput of operations and middleware-induced latency, the environment consisting of virtual machines is inadequate, as virtual machines compete for the same physical resources in an undeterministic fashion, resulting in an uncontrollable queueing delay.

Therefore, four dedicated workstation computers with the same hardware configuration were set up[3] . Each computer had an *AMD Athlon*^{TM}*64 3000+* processor running at 1.8GHz (3620 BogoMIPS) and 1 GB of RAM. The computers were interconnected with a 1 Gbps Ethernet network.

The installed operating system was *Scientific Linux 4* (Linux kernel version 2.6.9-34.0.1.EL). *Sun Microsystem*'s Java^{TM}virtual machine was of version 1.5.0_05.

7.4.2 Results

In this section we present the results of the performance measurements. First, we show how much is gained from acting optimistically in presence of

[3]This setup was provided by Klemen Zagar at Cosylab in Ljubljana, Slovenia, who also ran the overhead measurements

Table 7.2: Parameter summary

Number of runs	20
Number of objects	50
Number of constraints	50
Critical constraints	10 %
Number of nodes	4
Number of clients	2
Number of operations	200
Experiment duration	68 [s]
Load	6 [ops/s]

partitions using the CS reconciliation protocol, both in terms of apparent availability as well as number of accepted operations. Secondly, we have measured the overhead incurred by our solution by measuring the throughput of middleware operations. Finally, we investigate the effect of load on the reconciliation protocol.

Apparent Availability The availability measurements were performed by repeating the following scenario 20 times. First, a number of operations were performed in the normal mode. Then a partition fault was injected followed by further operations being applied in both partitions. After a specific time interval the injected fault was removed. At that point, the reconciliation process started. Invocations were continued during the whole reconciliation stage and a while into the normal mode. Table 7.2 summarises the parameters used. The duration of the experiment depends on the duration of the fault.

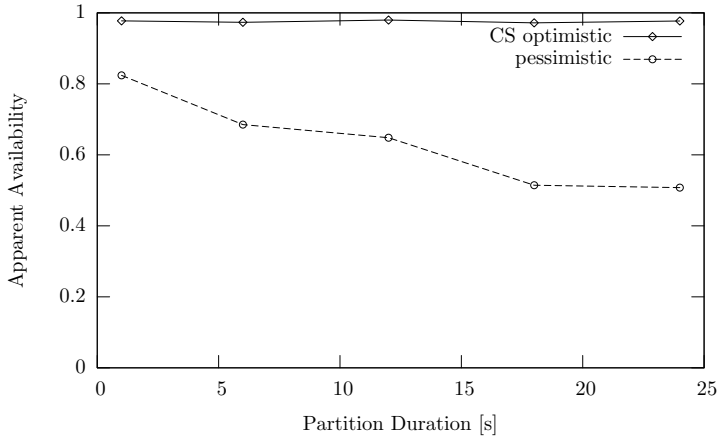

Figure 7.12: Apparent Availability vs. Partition Duration

In Figure 7.12 the apparent availability is plotted against the partition duration. As expected, the optimistic protocol maintains a constant high availability of approximately 98%. The unavailability is caused by the short period in which the reconciliation protocol awaits the final operations before installing a new state.

The availability of the pessimistic protocol is a bit more interesting. Since the duration of the experiment is 68[s] we would expect the availability to start at 100% and then drop off to 60%. However the performance is worse than that. The reason for this is that Spread takes some time before registering that the network has reunified. The timeout for lookups in the Spread implementation that we used was set to 10 seconds. This timeout is also the cause of the stair-like shape of the graph. The availability does not drop continuously as the fault duration increases. Instead there is a drop every 10-12 seconds that corresponds to the lookup interval of Spread.

Operation-based availability Apart from the apparent availability which reflects when the clients perceive the system to be apparently operational, it is equally important that the work performed during the period of degradation is kept when the system is later reunified. Therefore, we have investigated how many operations are successfully performed by the two approaches.

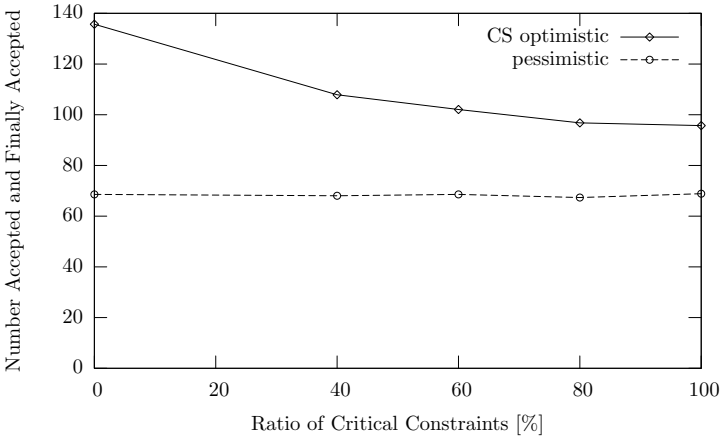

Figure 7.13: Accepted Operations vs. Ratio of Critical Constraints

Figure 7.13 shows the number of operations accepted in normal mode and the number of finally accepted operations put together. The parameters are the same as in the previous experiment except that the fault duration is kept constant at 10[s]. The ratio on the x-axis tells the proportion of the constraints in the system that are critical. Remember that if a critical constraint needs to be checked for a given operation, then that operation cannot

be performed optimistically. Thus the higher the ratio is, the less likely is it that acting optimistically pays off. This fact is reflected in the dropping of total number of accepted operations for the optimistic protocol. However, even if all constraints are critical there are a number of unconstrained objects. Therefore, there are always more operations performed with the optimistic approach compared to the pessimistic. Moreover, Spread is able to detect that the network has reunified faster when more spread messages are transmitted (as with the CS protocol). The pessimistic protocol on the other hand does not send any spread messages during the degraded mode. This causes the degradation period to be shorter for the optimistic scenario and thus more operations can be accepted in normal mode.

As can be seen in Table 7.2 there was a total of 50 constraints in the system. This means that in average 6-7 objects are unconstrained. If there would be even more constraints, then the performance of the CS protocol would decrease further. In the worst case, all objects are associated with a number of critical constraints. In such a scenario there would be no benefit in using an optimistic approach.

Overhead We have now established that we really do gain in increased availability with the optimistic middleware running the CS reconciliation protocol. However, most mechanisms for added fault tolerance require additional bandwidth and computational resources. To understand the cost associated with the added fault tolerance, we have performed single-client throughput measurements. These are equivalent to round-trip time measurements since throughput is the inverse of round-trip time.

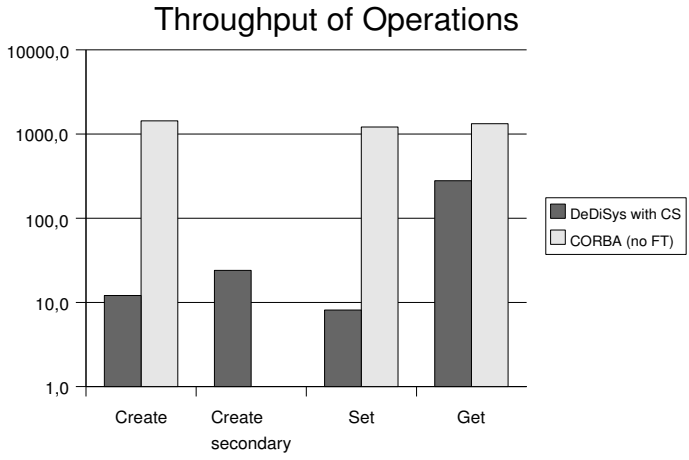

Figure 7.14: Throughput of operations.

These measurements are not meant to give the overhead of the CS reconciliation protocol but rather the overhead associated with the entire DeDiSys middleware in which the CS reconciliation protocol is a component. To get the overhead we compare the throughput with standard CORBA with no fault tolerance (Fig. 7.14). With such a platform, even though the objects are distributed over nodes, a node crash renders some objects unreachable for integrity checks. Moreover, there is no partition tolerance.

Obviously, there is no point in comparing the performance during degraded mode since the standard CORBA would not operate under such conditions. So the measurements are performed during normal mode. The CS reconciliation protocol is not active in this mode and no logging is required.

The experiments were done with batches of 50 invocations in each batch. The time taken to run one batch was measured using the `currentTimeMillis` method in Java. Execution of the entire batch was repeated five times to compute the variance. The variance was an indicator of the quality of the measurement. For batches with a large variance the experiment was repeated with a larger batch size (from 50 up to 10000 runs).

These results are a first attempt to characterise the overhead associated with our solution. For the standard CORBA operations there were no consistency checking and no time spent in transaction manager or replication protocols. That is, this seemingly extra overhead is not the cost of adopting automatic reconciliation; but includes the heavy costs of state-based replication [13] and consistency checking (which is part of application requirements). Since the baseline offers very high levels of performance, the impact on throughput is significant: a degradation of two orders of magnitude. However, considering that this is a non-optimised middleware, in absolute terms the throughput is still acceptable. In both normal and degraded modes, the system is still capable of handling roughly 10 mutable and several hundred read-only operations per second.

Effect of Load We have also investigated the effect of load on the reconciliation protocol. The termination proof of Section 5.4.2 is conditional on the minimum invocation inter-arrival time being greater than the reconciliation handling time. However, even if we achieve termination, the reconciliation time might still be very long. Therefore, we are interested in measuring the actual reconciliation duration as a function of system load.

Figure 7.15 shows that the duration increases as the load increases. The same parameters were used as in the availability measurements, but with a varying load. The experiment was run for higher loads than 10 as well, but in those cases the reconciliation did not finish during the length of the experiment. The rate at which operations were replayed was in the range 10-15. So it is consistent with the termination proof that the with the higher loads, termination may not happen. Again it is important to realize that the sandbox replay service has not been optimised with regards to performance.

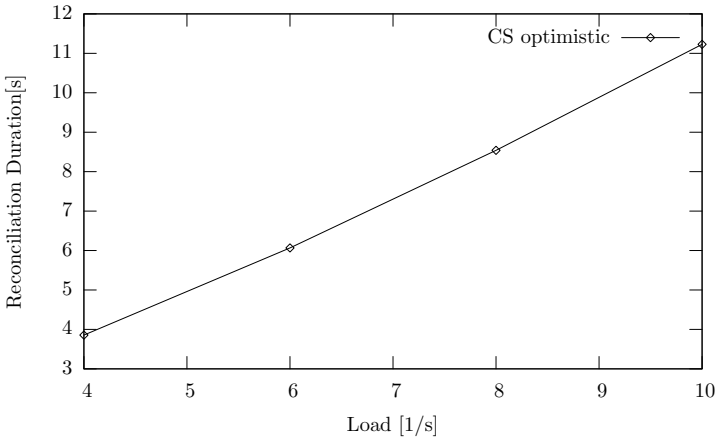

Figure 7.15: Reconciliation Duration vs. Load

"When 900 years you reach, look as good, you will not."

Yoda

8 Conclusions and Future Work

8.1 Conclusions

We have investigated different approaches for increasing availability during network partitions for systems with integrity constraints. Several protocols have been proposed and analysed both theoretically and experimentally.

We believe that the results provide strong support for our hypothesis that data-centric applications can be made partition-tolerant by acting optimistically and letting a continuous service operation-based reconciliation protocol take care of sorting out inconsistencies. We have showed that with the help of such a protocol one can achieve nearly 100% availability while still being able to restore full consistency. Finally, we have demonstrated that this can be done using a component in a general purpose middleware.

8.1.1 Optimistic Replication with Integrity Constraints

The fact that optimistic replication provides higher availability is fairly clear per definition. It is not equally obvious for systems with integrity constraints. Our work shows that if reconciliation can be done at a high enough rate, then the statement still holds.

Unfortunately, the availability gained through optimistic replication is only an apparent availability. Some of the operations that were optimistically accepted must be revoked later on. Therefore, in addition to the classic availability metric, the number of finally accepted operations must be considered when judging the benefit of optimistic replication. Also with this metric, the optimistic replication approach wins given a fast enough recon-

ciliation rate.

Moreover, as we have shown in our experimental studies, the penalty of acting pessimistically in presence of network partitions is much higher than seems necessary at first. Even a short period of disconnection can lead to long periods of degradation due to group membership services taking a long time to discover the new topology.

8.1.2 State vs Operation-based Reconciliation

There are several ways to perform reconciliation either by transferring state or by transferring operations. One state-based reconciliation scheme is to compose the final state by combining the states of primary objects (possibly from different partitions). This can be combined with reapplying operations that have been performed at secondary replicas. Unfortunately this approach is not compatible with system wide integrity constraints. If the primaries for objects A and B were in different partitions, then a constraint stating that $A > B$ could be violated. The only thing to do in such a scenario is to manually restore constraint consistency. However, for complex systems, this is an unfeasible approach.

The next variant of state-based reconciliation is to simply choose one of the partition states as the resulting state (Choose1). Although this seems simplistic, it is not as bad as it sounds, there are several benefits. First of all it allows for very quick reconciliation. Secondly, if the degraded mode lasted for only a short while, and the load were unevenly distributed amongst the partitions, there might not be that many operations that are lost. One can also compare this with the primary partition approach. The Choose1 algorithm would allow the same operations to be finally accepted as if there was a primary partition. Moreover, in situations were there were no primary partition, the Choose1 algorithm still allows the operations from one partition to remain.

However, as we showed in the simulation studies, the Choose1 algorithm performs poorly compared to the operation-based algorithms. The reason of course being the large number of operations that potentially need to be revoked by Choose1. The drawback with operation-based reconciliation is that one needs to be able to deal with side effects. However this problem already has to be dealt with in a system that uses operation-based replication.

We believe that for systems that can explicitly express integrity constraints, operation-based reconciliation is the best option for merging the partition states.

8.1.3 Optimising Reconciliation

In the ideal case, all operations that were accepted during a network partition could be kept in the reconciled state. However, in that case it can hardly be called optimistic replication anymore. For systems with integrity

constraints there will inevitably be conflicts. The idea that these conflicts can be compensated using some kind of merge procedure as in Bayou[91] is nice, but it doesn't really solve the problem. Any such logic that is incorporated in the merge procedure, could just as well be encoded in the operation in itself. It is more likely that a conflict is hard to solve automatically. Therefore the user that performed the action will probably need to be informed that the operation did not succeed. So there will be a cost associated with every conflict. Therefore, one of the goals of a reconciliation algorithm should be to minimise the number of revoked operations.

A big step in this direction is to actually let the application semantics decide what constitutes a conflict rather that relying on syntactic approaches such as read and write sets. Moreover, we believe that utilities can be a powerful way to further reduce the costs of the revocations. Our STW-GEU algorithm managed to increase the accrued utility compared to the STW-Merge algorithm that did not differentiate between operations. The cost was an increased time to build the resulting state. The usefulness of such a utility-based algorithm is dependent on the existence of relevant utility metrics. Our approach lets the application designer decide what are the most important operations to save, and which operations can be revoked easier.

8.1.4 Continuous Service

Since reconciliation can be rather time consuming we constructed a reconciliation protocol, called CS, that is able to maintain (apparent) availability of the system while the system reconciles. There are some issues that must be dealt with such as ordering, but we have showed that the CS protocol can guarantee client expected ordering of operations as well as maintain constraint consistency.

Moreover, it is clear that the CS protocol can be used to achieve very high availability since the even long running network partitions only cause a very short period of complete unavailability. Unless the rate at which reconciliation can be performed is very high, the CS protocol outperforms the STW-Merge algorithm both in terms of apparent availability, as well as the number of finally accepted operations. We have not compared the performance of th CS protocol with the STW-GEU. The reason was that CS was built upon the same principle as STW-Merge and we wanted to see the effect of a continued service protocol. However there is no problem in using the mechanism from STW-GEU in CS to obtain even better results.

We have demonstrated that the protocol can be implemented in a standard middleware (i.e., CORBA). This middleware does not provide completely transparent support for partition tolerance, since this is probably not possible for systems with strong consistency requirements. However, the application writer does only have to provide explicit integrity constraints, and figure out how to deal with revoked operations to utilise this approach.

The overhead costs of the middleware add-ons are quite high. However, they are not caused by the reconciliation protocol, but rather by the costs of synchronous state-based replication. The CS protocol could be combined with a more efficient replication and transaction scheme too.

8.2 Future work

There are of course many ways to improve our work. We will here describe what we think are interesting directions for future research.

8.2.1 Algorithm Improvements

The algorithms that we have proposed have to some extent been simplified to allow them to be explained, specified and analysed in a reasonable fashion. However, there are ideas for improvement that we have not been able to test yet.

Distribution Although the CS reconciliation protocol is a distributed algorithm that does not rely on any single node, it has a centralised idea. For every reconciliation round one reconciliation manager is elected to perform the reconciliation process. There are a couple of disadvantages to this. First, one node might not have enough resources to reconcile operations in a high enough rate (handling rate). Secondly, this requires the transfer of state between nodes. If the state is very large then this would be infeasible. The latter problem can be easily solved as discussed below.

The problem of resource limitations in one node is interesting. One approach would be to let the original primary for each object take care of reconciling operations for that particular object. However, since the reconciliation algorithm must make sure to check integrity constraints for each operation, the whole reconciliation procedure would have to be synchronised over all participating nodes. Therefore, it is not certain that there would be any improved performance at all. Given the fact that integrity constraints then would have to be checked on remote nodes instead of locally an improvement seems unlikely. However, the work could be split for objects that are independent of each other with regards to integrity constraints. This is illustrated in Figure 8.1.

Objects 1 and 2 are independent of objects 3 and 4. Therefore one could divide the work between two reconciliation managers that worked independently of each other.

Improved Ordering The STW-GEU algorithm is the only one in this thesis that tries to optimise the replay order of operations to reduce the number of revocations. As we discussed in Chapter 2, there are several approaches that try to use graph algorithms to reduce the conflict rate

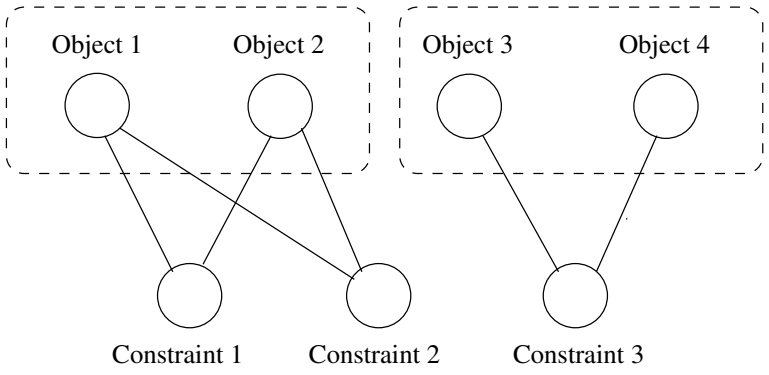

Figure 8.1: Independent objects

[29, 77]. These ideas could be combined with the semantics that are encoded in integrity constraints.

One such approach that would be interesting to evaluate is to estimate the probability of future violations for an operation using the following estimation. First, at any given time point during the reconciliation, count the number of operations that have yet to be reapplied and that are associated with a constraint that depends on a given object. In the example of Figure 8.2 the count for object 1 is 2 since operations 1 and 2 are conditional on the value of object 1. The same holds for Object 2, whereas the count is 1 for objects 3 and 4.

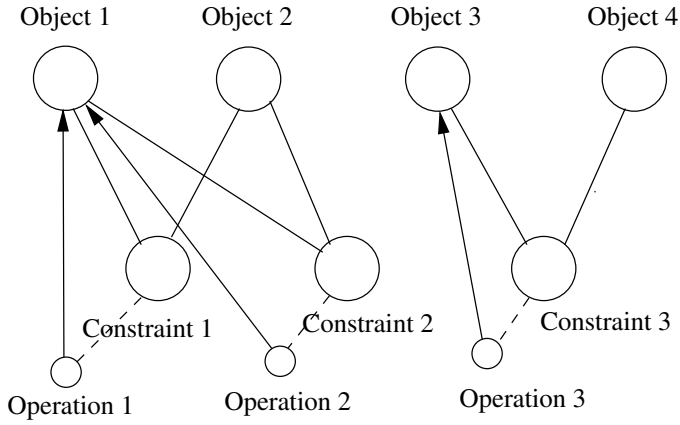

Figure 8.2: Relationship between objects and operations

Thus there is less risk in performing operations that update objects 3 or 4 compared to operations that update objects 1 or 2.

8.2.2 Mobility and Scale

We have assumed scenarios where all nodes and links are known beforehand and no nodes move during run-time. However, network partitions are actually much more likely in mobile ad-hoc networks (MANET). We have yet to see such network being used in a big scale in the real world. Therefore we do not yet know what types of applications that will emerge and what the requirements will be. However, we can reasonably expect that shared data will be a problem also in such scenarios.

Obviously, the group membership and group communication services that we rely on in this work is hard to achieve in MANETs. Moreover, if changes occur more often than is required for reconciliation to finish, then our approach would not work even in the static setting. Probably, one cannot expect to maintain the same degree of eventual consistency that we are able to guarantee. In such a scenario the approach of Bayou is more reasonable.

If one increases the scale of the network, similar problems as in a mobile scenario arise. This stems from the fact that one cannot know the entire state of the system. The result is that maintaining a consistent system-wide state is hard even without failures. A number of gossip algorithms have been proposed [93, 17, 47] to propagate state changes in an efficient manner for large scale or mobile networks.

8.2.3 Overloads

We needed to put an upper bound on load when proving the termination of the CS reconciliation protocol. In many networks (including those described above) this is not possible. In fact, system overload is a fault model that has not been very well researched outside of the real-time community. In soft real-time systems, there is usually an admission controller that limits the system load to assure schedulability. For mobile or large scale networks without a central admission control, this is not possible to achieve. Instead, one will probably have to deal with transient overloads much in the same way as if the network has partitioned.

Bibliography

[1] T. Abdelzaher, S. Dawson, W.-C. Feng, F. Jahanian, S. Johnson, A. Mehra, T. Mitton, A. Shaikh, K. Shin, Z. Wang, H. Zou, M. Bjorkland, and P. Marron. ARMADA middleware and communication services. *Real-Time Systems*, 16(2):127–153, May 1999.

[2] M. K. Aguilera and S. Toueg. Randomization and failure detection: A hybrid approach to solve consensus. Technical report, Cornell University, Ithaca, NY, USA, 1996.

[3] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: a communication subsystem for high availability. In *Twenty-Second International Symposium on Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers.*, pages 76–84, Boston, Massachusetts, USA, July 1992. IEEE Computer Society.

[4] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Dependable and Secure Computing*, 1(1):11–33, 2004.

[5] Ö. Babaoglu, A. Bartoli, and G. Dini. Enriched view synchrony: A programming paradigm for partitionable asynchronous distributed systems. *IEEE Transactions on Computers*, 46(6):642–658, 1997.

[6] O. Babaoglu, R. Davoli, L.-A. Giachini, and M. Gray Baker. RELACS: A communications infrastructure for constructing reliable applications in large-scale distributed systems. In *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences (HICSS'95)*, volume 2, pages 612–621, Los Alamitos, CA, USA, Jan. 1995. IEEE Computer Society.

[7] Ö. Babaoglu, R. Davoli, and A. Montresor. Group communication in partitionable systems: Specification and algorithms. *IEEE Transactions on Software Engineering*, 27(4):308–336, 2001.

[8] S. Balasubramaniam and B. C. Pierce. What is a file synchronizer? In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking (MobiCom'98)*, pages 98–108, New York, NY, USA, 1998. ACM Press.

[9] BBC News, Aug. 2004.

[10] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing (PODC'83)*, pages 27–30, New York, NY, USA, 1983. ACM Press.

[11] P. A. Bernstein and N. Goodman. Multiversion concurrency control-theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[12] S. Beyer, M. Bañuls, P. Galdámez, J. Osrael, and F. D. Muñoz-Escoí. Increasing availability in a replicated partitionable distributed object system. In *Proceedings of the Fourth International Symposium on Parallel and Distributed Processing and Applications (ISPA'06)*, volume 4330 of *Lecture Notes in Computer Science*, pages 682–695, Heidelberg, Germany, 2006. Springer-Verlag.

[13] S. Beyer, F. D. Muñoz-Escoí, and P. Galdámez. Implementing network partition-aware fault-tolerant CORBA systems. In *Proceedings of the 2nd International Conference. on Availability, Reliability, and Security (ARES'07)*, Los Alamitos, CA, USA, 2007. IEEE Computer Society Press.

[14] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the eleventh ACM Symposium on Operating systems principles (SOSP'87)*, pages 123–138, New York, NY, USA, 1987. ACM Press.

[15] K. P. Birman. Replication and fault-tolerance in the ISIS system. *SIGOPS Operating Systems Review*, 19(5):79–86, 1985.

[16] P. Cederqvist. *Version management with CVS*, 2005.

[17] R. Chandra, V. Ramasubramanian, and K. Birman. Anonymous gossip: improving multicast reliability in mobile ad-hoc networks. In *21st International Conference on Distributed Computing Systems (ICDCS'01)*, pages 275–283, Los Alamitos, CA, USA, Apr. 2001. IEEE Computer Society.

[18] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[19] B. Charron-Bost and A. Schiper. Harmful dogmas in fault tolerant distributed computing. *SIGACT News*, 38(1):53–61, 2007.

[20] W. Chen, S. Toueg, and M. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(1):13–32, Jan. 2002.

[21] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

[22] B. A. Coan, B. M. Oki, and E. K. Kolodner. Limitations on database availability when networks partition. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing (PODC'86)*, pages 187–194, New York, NY, USA, 1986. ACM Press.

[23] B. Collins-Sussmann, B. Fitzpatrick, and C. Pilato. *Version Control with Subversion*. O'Reilly, 2004.

[24] C. Coulon, E. Pacitti, and P. Valduriez. Consistency management for partial replication in a high performance database cluster. In *Proceedings of the 11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, volume 1, pages 809–815 Vol.1, Los Alamitos, CA, USA, July 2005. IEEE Computer Society.

[25] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[26] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, June 1999.

[27] F. Cristian and F. Schmuck. Agreeing on processor group membership in timed asynchronous distributed systems, UCSD technical report CSE95-428. Technical report, Department of Computer Science and Engineering, University of California, 1995.

[28] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. Sanders, D. Bakken, M. Berman, D. Karr, and R. Schantz. AQuA: an adaptive architecture that provides dependable distributed objects. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98)*, pages 245–253, Los Alamitos, CA, USA, Oct. 1998. IEEE Computer Society.

[29] S. B. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM Transactions on Database Systems*, 9(3):456–481, 1984.

[30] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys*, 17(3):341–370, 1985.

[31] DeDiSys. European IST FP6 DeDiSys Project. http://www.dedisys.org, 2006.

[32] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.

[33] D. Dolev, R. Friedman, I. Keidar, and D. Malkhi. Failure detectors in omission failure environments. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing (PODC'97)*, page 286, New York, NY, USA, 1997. ACM Press.

[34] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[35] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: a fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 296–306, Los Alamitos, CA, USA, June 1995. IEEE Computer Society.

[36] P. Felber and P. Narasimhan. Experiences, strategies, and challenges in building fault-tolerant CORBA systems. *IEEE Transactions on Computers*, 53(5):497–511, 2004.

[37] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. In *Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems (PODS'83)*, pages 1–7. ACM Press, 1983.

[38] A. Fox and E. Brewer. Harvest, yield, and scalable tolerant systems. In *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on*, pages 174–178, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[39] R. Friedman and R. van Renesse. Strong and weak virtual synchrony in horus. In *Proceedings of the 15th Symposium on Reliable Distributed Systems (SRDS '96)*, page 140, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[40] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, 1999.

[41] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[42] P. Godfrey, J. Grant, J. Gryz, and J. Minker. Integrity constraints: semantics and applications. In J. Chomicki and G. Saake, editors, *Logics for databases and information systems*, pages 265–306. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[43] A. P. Goldberg, A. Gopal, A. Lowry, and R. Strom. Restoring consistent global states of distributed computations. In *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging (PADD'91)*, pages 144–154, New York, NY, USA, 1991. ACM Press.

[44] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data (SIGMOD'96)*, pages 173–182, New York, NY, USA, 1996. ACM Press.

[45] R. Guerraoui, M. Larrea, and A. Schiper. Non blocking atomic commitment with an unreliable failure detector. In *Reliable Distributed Systems, 1995. Proceedings., 14th Symposium on*, pages 41–50, Sept. 1995.

[46] I. Gupta, T. D. Chandra, and G. S. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing (PODC'01)*, pages 170–179, New York, NY, USA, 2001. ACM Press.

[47] Z. Haas, J. Halpern, and L. Li. Gossip-based ad hoc routing. *IEEE/ACM Transactions on Networking*, 14(3):479–491, June 2006.

[48] N. Hayashibara, A. Cherif, and T. Katayama. Failure detectors for large-scale distributed systems. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, page 404, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[49] A. A. Helal, B. K. Bhargava, and A. A. Heddaya. *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[50] A. S. Helal. Modeling database system availability under network partitioning. *Information Sciences*, 83(1-2):23–35, Mar. 1995.

[51] J. Holliday, D. Agrawal, and A. E. Abbadi. Disconnection modes for mobile databases. *Wireless Networks*, 8(4):391–402, 2002.

[52] H.-M. Huang and C. Gill. Design and performance of a fault-tolerant real-time CORBA event service. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS'06)*, pages 33–42, Los Alamitos, CA, USA, July 2006. IEEE Computer Society.

[53] J-sim. www.j-sim.org.

[54] W. Jia, J. Kaiser, and E. Nett. Rmp: fault-tolerant group communication. *IEEE Micro*, 16(2):59–67, Apr. 1996.

[55] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. Moshe: A group membership service for wans. *ACM Transactions on Computer Systems*, 20(3):191–238, 2002.

[56] B. Kemme, A. Bartoli, and Ö. Babaoglu. Online reconfiguration in replicated databases based on group communication. In *Proceedings*

of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS) (DSN'01), pages 117–130. IEEE Computer Society, 2001.

[57] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The icecube approach to the reconciliation of divergent replicas. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing (PODC'01)*, pages 210–218, New York, NY, USA, 2001. ACM Press.

[58] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

[59] S. Krishnamurthy, W. H. Sanders, and M. Cukier. An adaptive framework for tunable consistency and timeliness using replication. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN'02)*, pages 17–26. IEEE Computer Society, 2002.

[60] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[61] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[62] S. Maffeis. Adding group communication and fault-tolerance to corba. In *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)*, Berkeley, CA, USA, June 1995. USENIX Association.

[63] V. Martins, R. Akbarinia, E. Pacitti, and P. Valduriez. Reconciliation in the APPA P2P system. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems (ICPADS'2006)*, volume 1, pages 401–410, Los Alamitos, CA, USA, July 2006. IEEE Computer Society.

[64] B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, Oct. 1992.

[65] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory*, 49(9):2213–2218, Sept. 2003.

[66] L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems (ICDCS'94)*, pages 56–65, Los Alamitos, CA, USA, June 1994. IEEE Computer Society.

[67] L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, C. Lingley-Papadopoulos, and T. Archambault. The totem system. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS-25). Digest of Papers.*, pages 61–66, Los Alamitos, CA, USA, June 1995. IEEE Computer Society.

[68] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. Consistent object replication in the eternal system. *Theory and Prectice of Object Systems*, 4(2):81–92, 1998.

[69] P. Narasimhan, T. A. Dumitraş, A. M. Paulos, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava. MEAD: support for real-time fault-tolerant CORBA. *Concurrency and Computation: Practice and Experience*, 17(12):1527–1545, 2005.

[70] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Replica consistency of CORBA objects in partitionable distributed systems. *Distributed Systems Engineering*, 4(3):139–150, 1997.

[71] B. Natarajan, A. Gokhale, S. Yajnik, and D. Schmidt. Doors: towards high-performance fault tolerant CORBA. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'00)*, pages 39–48, Los Alamitos, CA, USA, Sept. 2000. IEEE Computer Society.

[72] Object Management Group (OMG). Corba fault tolerant chapter, v3.03, formal/04-03-21, 2004.

[73] J. Osrael, L. Froihofer, and K. Goeschka. A system architecture for enhanced availability of tightly coupled distributed systems. In *The First International Conference on Availability, Reliability and Security (ARES 2006)*, page 8pp., Los Alamitos, CA, USA, Apr. 2006. IEEE Computer Society.

[74] K. Petersen, M. Spreitzer, D. Terry, and M. Theimer. Bayou: replicated database services for world-wide applications. In *Proceedings of the 7th workshop on ACM SIGOPS European workshop (EW 7)*, pages 275–280, New York, NY, USA, 1996. ACM Press.

[75] S. H. Phatak and B. Nath. Transaction-centric reconciliation in disconnected client-server databases. *Mobile Networks and Applications*, 9(5):459–471, 2004.

[76] E. Pitoura and B. Bhargava. Data consistency in intermittently connected distributed systems. *IEEE Transactions on Knowledge and Data Engineering*, 11(6):896–915, 1999.

[77] N. Preguica, M. Shapiro, and C. Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *On The Move*

to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE, volume 2888 of *Lecture Notes in Computer Science*, pages 38–55, Heidelberg, Germany, Oct. 2003. Springer.

[78] C. Pu and A. Leff. Replica control in distributed systems: as asynchronous approach. In *Proceedings of the 1991 ACM SIGMOD international conference on Management of data (SIGMOD'91)*, pages 377–386. ACM Press, 1991.

[79] N. Ramsey and E. od Csirmaz. An algebraic approach to file synchronization. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-9)*, pages 175–185, New York, NY, USA, 2001. ACM Press.

[80] P. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek. Resolving file conflicts in the Ficus file system. In *USENIX Conference Proceedings*, pages 183–195, Boston, MA, June 1994. USENIX.

[81] Säkerhetscentret.se, Mar. 2007.

[82] Säkerhetscentret.se, Feb. 2007.

[83] F. B. Schneider. *What good are models and what models are good?*, pages 17–26. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[84] M. Shapiro, K. Bhargavan, and N. Krishna. A constraint-based formalism for consistency in replicated systems. In *Principles of Distributed Systems*, pages 331–345. Springer, 2005.

[85] D. P. Siewiorek and R. S. Swarz. *Reliable computer systems (3rd ed.): design and evaluation*. A. K. Peters, Ltd., Natick, MA, USA, 1998.

[86] A. V. Singh, L. E. Moser, and P. M. Melliar-Smith. Integrating fault tolerance and load balancing in distributed systems based on CORBA. In *Proceedings of the 5th European Dependable Computing Conference (EDCC'05)*, volume 3463 of *Lecture Notes in Computer Science*, pages 154–166, Heidelberg, Germany, 2005. Springer.

[87] Spread. The spread toolkit. http://www.spread.org, 2006.

[88] D. Szentivanyi. *Performance Studies of Fault-Tolerant Middleware*. PhD thesis, Linköpings universitet, 2005.

[89] D. Szentivanyi and S. Nadjm-Tehrani. Middleware support for fault tolerance. In Q. Mahmoud, editor, *Middleware for Communications*. John Wiley & Sons, 2004.

[90] G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, New York, NY, USA, 2001.

[91] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles (SOSP'95)*, pages 172–182, New York, NY, USA, 1995. ACM Press.

[92] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: a flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

[93] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. Technical report, Cornell University, Ithaca, NY, USA, 1998.

[94] A. Vaysburd and K. Birman. The maestro approach to building reliable interoperable distributed applications with multiple execution styles. *Theory and Prectice of Object Systems*, 4(2):71–80, 1998.

[95] A.-I. Wang, P. L. Reiher, R. Bagrodia, and G. H. Kuenning. Understanding the behavior of the conflict-rate metric in optimistic peer replication. In *Proceedings of the 13th International Workshop on Database and Expert Systems Applications (DEXA'02)*, pages 757–764, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[96] H. ying Tyan. *Design, realization and evaluation of a component-based software architecture for network simulation.* PhD thesis, Department of Electrical Engineering, Ohio State University, 2001.

[97] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems*, 20(3):239–282, 2002.

[98] H. Yu and A. Vahdat. Minimal replication cost for availability. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing (PODC'02)*, pages 98–107, New York, NY, USA, 2002. ACM Press.

[99] C. Zhang and Z. Zhang. Trading replication consistency for performance and availability: an adaptive approach. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)*, page 687. IEEE Computer Society, 2003.

Avdelning, Institution
Division, Department

Institutionen för datavetenskap,
Dept. of Computer and Information Science
581 83 Linköping

Datum
Date

2007-10-16

Språk
Language
☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category
☒ Licentiatavhandling
☐ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN
978-91-85895-89-2

ISRN
LiU-Tek-Lic–2007:40

Serietitel och serienummer **ISSN**
Title of series, numbering
0280–7971

Linköping Studies in Science and Technology

Thesis No. 1331

URL för elektronisk version

Titel
Title

Restoring Consistency after Network Partitions

Författare
Author

Mikael Asplund

Sammanfattning
Abstract

The software industry is facing a great challenge. While systems get more complex and distributed across the world, users are becoming more dependent on their availability. As systems increase in size and complexity so does the risk that some part will fail. Unfortunately, it has proven hard to tackle faults in distributed systems without a rigorous approach. Therefore, it is crucial that the scientific community can provide answers to how distributed computer systems can continue functioning despite faults.

Our contribution in this thesis is regarding a special class of faults which occurs when network links fail in such a way that parts of the network become isolated, such faults are termed network partitions. We consider the problem of how systems that have integrity constraints on data can continue operating in presence of a network partition. Such a system must act optimistically while the network is split and then perform a some kind of reconciliation to restore consistency afterwards.

We have formally described four reconciliation algorithms and proven them correct. The novelty of these algorithms lies in the fact that they can restore consistency after network partitions in a system with integrity constraints and that one of the protocols allows the system to provide service *during* the reconciliation. We have implemented and evaluated the algorithms using simulation and as part of a partition-tolerant CORBA middleware. The results indicate that it pays off to act optimistically and that it is worthwhile to provide service during reconciliation.

Nyckelord
Keywords
distributed systems, fault tolerance, network partitions, dependability, integrity constraints

Department of Computer and Information Science
Linköpings universitet

Linköping Studies in Science and Technology
Faculty of Arts and Sciences - Licentiate Theses

No 17 **Vojin Plavsic:** Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)

No 28 **Arne Jönsson, Mikael Patel:** An Interactive Flowcharting Technique for Communicating and Realizing Algorithms, 1984.

No 29 **Johnny Eckerland:** Retargeting of an Incremental Code Generator, 1984.

No 48 **Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching, 1985.

No 52 **Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems, 1985.

No 60 **Johan Fagerström:** Simulation and Evaluation of Architecture based on Asynchronous Processes, 1985.

No 71 **Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System, 1987.

No 72 **Tony Larsson:** On the Specification and Verification of VLSI Systems, 1986.

No 73 **Ola Strömfors:** A Structure Editor for Documents and Programs, 1986.

No 74 **Christos Levcopoulos:** New Results about the Approximation Behavior of the Greedy Triangulation, 1986.

No 104 **Shamsul I. Chowdhury: S**tatistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology, 1987.

No 108 **Rober Bilos:** Incremental Scanning and Token-Based Editing, 1987.

No 111 **Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments, 1987.

No 113 **Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge, 1987.

No 118 **Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs, 1987.

No 126 **Dan Strömberg:** Transfer and Distribution of Application Programs, 1987.

No 127 **Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems, 1987.

No 139 **Christer Bäckström:** Reasoning about Interdependent Actions, 1988.

No 140 **Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing, 1988.

No 146 **Johan Hultman:** A Software System for Defining and Controlling Actions in a Mechanical System, 1988.

No 150 **Tim Hansen:** Diagnosing Faults using Knowledge about Malfunctioning Behavior, 1988.

No 165 **Jonas Löwgren:** Supporting Design and Management of Expert System User Interfaces, 1989.

No 166 **Ola Petersson:** On Adaptive Sorting in Sequential and Parallel Models, 1989.

No 174 **Yngve Larsson:** Dynamic Configuration in a Distributed Environment, 1989.

No 177 **Peter Åberg:** Design of a Multiple View Presentation and Interaction Manager, 1989.

No 181 **Henrik Eriksson:** A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.

No 184 **Ivan Rankin:** The Deep Generation of Text in Expert Critiquing Systems, 1989.

No 187 **Simin Nadjm-Tehrani:** Contributions to the Declarative Approach to Debugging Prolog Programs, 1989.

No 189 **Magnus Merkel:** Temporal Information in Natural Language, 1989.

No 196 **Ulf Nilsson:** A Systematic Approach to Abstract Interpretation of Logic Programs, 1989.

No 197 **Staffan Bonnier:** Horn Clause Logic with External Procedures: Towards a Theoretical Framework, 1989.

No 203 **Christer Hansson:** A Prototype System for Logical Reasoning about Time and Action, 1990.

No 212 **Björn Fjellborg:** An Approach to Extraction of Pipeline Structures for VLSI High-Level Synthesis, 1990.

No 230 **Patrick Doherty:** A Three-Valued Approach to Non-Monotonic Reasoning, 1990.

No 237 **Tomas Sokolnicki:** Coaching Partial Plans: An Approach to Knowledge-Based Tutoring, 1990.

No 250 **Lars Strömberg:** Postmortem Debugging of Distributed Systems, 1990.

No 253 **Torbjörn Näslund:** SLDFA-Resolution - Computing Answers for Negative Queries, 1990.

No 260 **Peter D. Holmes:** Using Connectivity Graphs to Support Map-Related Reasoning, 1991.

No 283 **Olof Johansson:** Improving Implementation of Graphical User Interfaces for Object-Oriented Knowledge-Bases, 1991.

No 298 **Rolf G Larsson:** Aktivitetsbaserad kalkylering i ett nytt ekonomisystem, 1991.

No 318 **Lena Strömbäck:** Studies in Extended Unification-Based Formalism for Linguistic Description: An Algorithm for Feature Structures with Disjunction and a Proposal for Flexible Systems, 1992.

No 319 **Mikael Pettersson:** DML-A Language and System for the Generation of Efficient Compilers from Denotational Specification, 1992.

No 326 **Andreas Kågedal:** Logic Programming with External Procedures: an Implementation, 1992.

No 328 **Patrick Lambrix:** Aspects of Version Management of Composite Objects, 1992.

No 333 **Xinli Gu:** Testability Analysis and Improvement in High-Level Synthesis Systems, 1992.

No 335 **Torbjörn Näslund:** On the Role of Evaluations in Iterative Development of Managerial Support Sytems, 1992.

No 348 **Ulf Cederling:** Industrial Software Development - a Case Study, 1992.

No 352 **Magnus Morin:** Predictable Cyclic Computations in Autonomous Systems: A Computational Model and Implementation, 1992.

No 371 **Mehran Noghabai:** Evaluation of Strategic Investments in Information Technology, 1993.

No 378 **Mats Larsson:** A Transformational Approach to Formal Digital System Design, 1993.

No 380 **Johan Ringström:** Compiler Generation for Parallel Languages from Denotational Specifications, 1993.

No 381 **Michael Jansson:** Propagation of Change in an Intelligent Information System, 1993.

No 383 **Jonni Harrius:** An Architecture and a Knowledge Representation Model for Expert Critiquing Systems, 1993.

No 386 **Per Österling:** Symbolic Modelling of the Dynamic Environments of Autonomous Agents, 1993.

No 398 **Johan Boye:** Dependency-based Groudness Analysis of Functional Logic Programs, 1993.

No 402 **Lars Degerstedt:** Tabulated Resolution for Well Founded Semantics, 1993.
No 406 **Anna Moberg:** Satellitkontor - en studie av kommunikationsmönster vid arbete på distans, 1993.
No 414 **Peter Carlsson:** Separation av företagsledning och finansiering - fallstudier av företagsledarutköp ur ett agentteoretiskt perspektiv, 1994.
No 417 **Camilla Sjöström:** Revision och lagreglering - ett historiskt perspektiv, 1994.
No 436 **Cecilia Sjöberg:** Voices in Design: Argumentation in Participatory Development, 1994.
No 437 **Lars Viklund:** Contributions to a High-level Programming Environment for a Scientific Computing, 1994.
No 440 **Peter Loborg:** Error Recovery Support in Manufacturing Control Systems, 1994.
FHS 3/94 **Owen Eriksson:** Informationssystem med verksamhetskvalitet - utvärdering baserat på ett verksamhetsinriktat och samskapande perspektiv, 1994.
FHS 4/94 **Karin Pettersson:** Informationssystemstrukturering, ansvarsfördelning och användarinflytande - En komparativ studie med utgångspunkt i två informationssystemstrategier, 1994.
No 441 **Lars Poignant:** Informationsteknologi och företagsetablering - Effekter på produktivitet och region, 1994.
No 446 **Gustav Fahl:** Object Views of Relational Data in Multidatabase Systems, 1994.
No 450 **Henrik Nilsson:** A Declarative Approach to Debugging for Lazy Functional Languages, 1994.
No 451 **Jonas Lind:** Creditor - Firm Relations: an Interdisciplinary Analysis, 1994.
No 452 **Martin Sköld:** Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques, 1994.
No 455 **Pär Carlshamre:** A Collaborative Approach to Usability Engineering: Technical Communicators and System Developers in Usability-Oriented Systems Development, 1994.
FHS 5/94 **Stefan Cronholm:** Varför CASE-verktyg i systemutveckling? - En motiv- och konsekvensstudie avseende arbetssätt och arbetsformer, 1994.
No 462 **Mikael Lindvall:** A Study of Traceability in Object-Oriented Systems Development, 1994.
No 463 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av Sandviks förvärv av Bahco Verktyg, 1994.
No 464 **Hans Olsén:** Collage Induction: Proving Properties of Logic Programs by Program Synthesis, 1994.
No 469 **Lars Karlsson:** Specification and Synthesis of Plans Using the Features and Fluents Framework, 1995.
No 473 **Ulf Söderman:** On Conceptual Modelling of Mode Switching Systems, 1995.
No 475 **Choong-ho Yi:** Reasoning about Concurrent Actions in the Trajectory Semantics, 1995.
No 476 **Bo Lagerström:** Successiv resultatavräkning av pågående arbeten. - Fallstudier i tre byggföretag, 1995.
No 478 **Peter Jonsson:** Complexity of State-Variable Planning under Structural Restrictions, 1995.
FHS 7/95 **Anders Avdic:** Arbetsintegrerad systemutveckling med kalkylkprogram, 1995.
No 482 **Eva L Ragnemalm:** Towards Student Modelling through Collaborative Dialogue with a Learning Companion, 1995.
No 488 **Eva Toller:** Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
No 489 **Erik Stoy:** A Petri Net Based Unified Representation for Hardware/Software Co-Design, 1995.
No 497 **Johan Herber:** Environment Support for Building Structured Mathematical Models, 1995.
No 498 **Stefan Svenberg:** Structure-Driven Derivation of Inter-Lingual Functor-Argument Trees for Multi-Lingual Generation, 1995.
No 503 **Hee-Cheol Kim:** Prediction and Postdiction under Uncertainty, 1995.
FHS 8/95 **Dan Fristedt:** Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.
FHS 9/95 **Malin Bergvall:** Systemförvaltning i praktiken - en kvalitativ studie avseende centrala begrepp, aktiviteter och ansvarsroller, 1995.
No 513 **Joachim Karlsson:** Towards a Strategy for Software Requirements Selection, 1995.
No 517 **Jakob Axelsson:** Schedulability-Driven Partitioning of Heterogeneous Real-Time Systems, 1995.
No 518 **Göran Forslund:** Toward Cooperative Advice-Giving Systems: The Expert Systems Experience, 1995.
No 522 **Jörgen Andersson:** Bilder av småföretagares ekonomistyrning, 1995.
No 538 **Staffan Flodin:** Efficient Management of Object-Oriented Queries with Late Binding, 1996.
No 545 **Vadim Engelson:** An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing, 1996.
No 546 **Magnus Werner :** Multidatabase Integration using Polymorphic Queries and Views, 1996.
FiF-a 1/96 **Mikael Lind:** Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
No 549 **Jonas Hallberg:** High-Level Synthesis under Local Timing Constraints, 1996.
No 550 **Kristina Larsen:** Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag. 1996.
No 557 **Mikael Johansson:** Quality Functions for Requirements Engineering Methods, 1996.
No 558 **Patrik Nordling:** The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.
No 561 **Anders Ekman:** Exploration of Polygonal Environments, 1996.
No 563 **Niclas Andersson:** Compilation of Mathematical Models to Parallel Code, 1996.
No 567 **Johan Jenvald:** Simulation and Data Collection in Battle Training, 1996.
No 575 **Niclas Ohlsson:** Software Quality Engineering by Early Identification of Fault-Prone Modules, 1996.
No 576 **Mikael Ericsson:** Commenting Systems as Design Support—A Wizard-of-Oz Study, 1996.
No 587 **Jörgen Lindström:** Chefers användning av kommunikationsteknik, 1996.
No 589 **Esa Falkenroth:** Data Management in Control Applications - A Proposal Based on Active Database Systems, 1996.
No 591 **Niclas Wahllöf:** A Default Extension to Description Logics and its Applications, 1996.
No 595 **Annika Larsson:** Ekonomisk Styrning och Organisatorisk Passion - ett interaktivt perspektiv, 1997.
No 597 **Ling Lin:** A Value-based Indexing Technique for Time Sequences, 1997.

No 598	**Rego Granlund:** C^3Fire - A Microworld Supporting Emergency Management Training, 1997.
No 599	**Peter Ingels:** A Robust Text Processing Technique Applied to Lexical Error Recovery, 1997.
No 607	**Per-Arne Persson:** Toward a Grounded Theory for Support of Command and Control in Military Coalitions, 1997.
No 609	**Jonas S Karlsson:** A Scalable Data Structure for a Parallel Data Server, 1997.
FiF-a 4	**Carita Åbom:** Videomötesteknik i olika affärssituationer - möjligheter och hinder, 1997.
FiF-a 6	**Tommy Wedlund**: Att skapa en företagsanpassad systemutvecklingsmodell - genom rekonstruktion, värdering och vidareutveckling i T50-bolag inom ABB, 1997.
No 615	**Silvia Coradeschi**: A Decision-Mechanism for Reactive and Coordinated Agents, 1997.
No 623	**Jan Ollinen:** Det flexibla kontorets utveckling på Digital - Ett stöd för multiflex? 1997.
No 626	**David Byers:** Towards Estimating Software Testability Using Static Analysis, 1997.
No 627	**Fredrik Eklund:** Declarative Error Diagnosis of GAPLog Programs, 1997.
No 629	**Gunilla Ivefors:** Krigsspel coh Informationsteknik inför en oförutsägbar framtid, 1997**.**
No 631	**Jens-Olof Lindh:** Analysing Traffic Safety from a Case-Based Reasoning Perspective, 1997
No 639	**Jukka Mäki-Turja:**. Smalltalk - a suitable Real-Time Language, 1997.
No 640	**Juha Takkinen**: CAFE: Towards a Conceptual Model for Information Management in Electronic Mail, 1997.
No 643	**Man Lin**: Formal Analysis of Reactive Rule-based Programs, 1997.
No 653	**Mats Gustafsson**: Bringing Role-Based Access Control to Distributed Systems, 1997.
FiF-a 13	**Boris Karlsson:** Metodanalys för förståelse och utveckling av systemutvecklingsverksamhet. Analys och värdering av systemutvecklingsmodeller och dess användning, 1997.
No 674	**Marcus Bjäreland:** Two Aspects of Automating Logics of Action and Change - Regression and Tractability, 1998.
No 676	**Jan Håkegård**: Hiera rchical Test Architecture and Board-Level Test Controller Synthesis, 1998.
No 668	**Per-Ove Zetterlund**: Normering av svensk redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998.
No 675	**Jimmy Tjäder**: Projektledaren & planen - en studie av projektledning i tre installations- och systemutvecklingsprojekt, 1998.
FiF-a 14	**Ulf Melin**: Informationssystem vid ökad affärs- och processorientering - egenskaper, strategier och utveckling, 1998.
No 695	**Tim Heyer**: COMPASS: Introduction of Formal Methods in Code Development and Inspection, 1998.
No 700	**Patrik Hägglund:** Programming Languages for Computer Algebra, 1998.
FiF-a 16	**Marie-Therese Christiansson:** Inter-organistorisk verksamhetsutveckling - metoder som stöd vid utveckling av partnerskap och informationssystem, 1998.
No 712	**Christina Wennestam:** Information om immateriella resurser. Investeringar i forskning och utveckling samt i personal inom skogsindustrin, 1998.
No 719	**Joakim Gustafsson:** Extending Temporal Action Logic for Ramification and Concurrency, 1998.
No 723	**Henrik André-Jönsson:** Indexing time-series data using text indexing methods, 1999.
No 725	**Erik Larsson:** High-Level Testability Analysis and Enhancement Techniques, 1998.
No 730	**Carl-Johan Westin:** Informationsförsörjning: en fråga om ansvar - aktiviteter och uppdrag i fem stora svenska organisationers operativa informationsförsörjning, 1998.
No 731	**Åse Jansson:** Miljöhänsyn - en del i företags styrning, 1998.
No 733	**Thomas Padron-McCarthy:** Performance-Polymorphic Declarative Queries, 1998.
No 734	**Anders Bäckström:** Värdeskapande kreditgivning - Kreditriskhantering ur ett agentteoretiskt perspektiv, 1998.
FiF-a 21	**Ulf Seigerroth:** Integration av förändringsmetoder - en modell för välgrundad metodintegration, 1999.
FiF-a 22	**Fredrik Öberg:** Object-Oriented Frameworks - A New Strategy for Case Tool Development, 1998.
No 737	**Jonas Mellin:** Predictable Event Monitoring, 1998.
No 738	**Joakim Eriksson:** Specifying and Managing Rules in an Active Real-Time Database System, 1998.
FiF-a 25	**Bengt E W Andersson:** Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörsarenor i samverkan om utbyte av information, 1998.
No 742	**Pawel Pietrzak:** Static Incorrectness Diagnosis of CLP (FD), 1999.
No 748	**Tobias Ritzau:** Real-Time Reference Counting in RT-Java, 1999.
No 751	**Anders Ferntoft:** Elektronisk affärskommunikation - kontaktkostnader och kontaktprocesser mellan kunder och leverantörer på producentmarknader,1999.
No 752	**Jo Skåmedal:** Arbete på distans och arbetsformens påverkan på resor och resmönster, 1999.
No 753	**Johan Alvehus:** Mötets metaforer. En studie av berättelser om möten, 1999.
No 754	**Magnus Lindahl:** Bankens villkor i låneavtal vid kreditgivning till högt belånade företagsförvärv: En studie ur ett agentteoretiskt perspektiv, 2000.
No 766	**Martin V. Howard:** Designing dynamic visualizations of temporal data, 1999.
No 769	**Jesper Andersson:** Towards Reactive Software Architectures, 1999.
No 775	**Anders Henriksson:** Unique kernel diagnosis, 1999.
FiF-a 30	**Pär J. Ågerfalk:** Pragmatization of Information Systems - A Theoretical and Methodological Outline, 1999.
No 787	**Charlotte Björkegren:** Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999.
No 788	**Håkan Nilsson:** Informationsteknik som drivkraft i granskningsprocessen - En studie av fyra revisionsbyråer, 2000.
No 790	**Erik Berglund:** Use-Oriented Documentation in Software Development, 1999.
No 791	**Klas Gäre:** Verksamhetsförändringar i samband med IS-införande, 1999.
No 800	**Anders Subotic:** Software Quality Inspection, 1999.
No 807	**Svein Bergum**: Managerial communication in telework, 2000.

No 809 **Flavius Gruian:** Energy-Aware Design of Digital Systems, 2000.

FiF-a 32 **Karin Hedström:** Kunskapsanvändning och kunskapsutveckling hos verksamhetskonsulter - Erfarenheter från ett FOU-samarbete, 2000.

No 808 **Linda Askenäs:** Affärssystemet - En studie om teknikens aktiva och passiva roll i en organisation, 2000.

No 820 **Jean Paul Meynard:** Control of industrial robots through high-level task programming, 2000.

No 823 **Lars Hult:** Publika Gränsytor - ett designexempel, 2000.

No 832 **Paul Pop:** Scheduling and Communication Synthesis for Distributed Real-Time Systems, 2000.

FiF-a 34 **Göran Hultgren:** Nätverksinriktad Förändringsanalys - perspektiv och metoder som stöd för förståelse och utveckling av affärsrelationer och informationssystem, 2000.

No 842 **Magnus Kald:** The role of management control systems in strategic business units, 2000.

No 844 **Mikael Cäker:** Vad kostar kunden? Modeller för intern redovisning, 2000.

FiF-a 37 **Ewa Braf**: Organisationers kunskapsverksamheter - en kritisk studie av ”knowledge management”, 2000.

FiF-a 40 **Henrik Lindberg:** Webbaserade affärsprocesser - Möjligheter och begränsningar, 2000.

FiF-a 41 **Benneth Christiansson:** Att komponentbasera informationssystem - Vad säger teori och praktik?, 2000.

No. 854 **Ola Pettersson:** Deliberation in a Mobile Robot, 2000.

No 863 **Dan Lawesson**: Towards Behavioral Model Fault Isolation for Object Oriented Control Systems, 2000.

No 881 **Johan Moe:** Execution Tracing of Large Distributed Systems, 2001.

No 882 **Yuxiao Zhao:** XML-based Frameworks for Internet Commerce and an Implementation of B2B e-procurement, 2001.

No 890 **Annika Flycht-Eriksson:** Domain Knowledge Management inInformation-providing Dialogue systems, 2001.

FiF-a 47 **Per-Arne Segerkvis**t: Webbaserade imaginära organisationers samverkansformer: Informationssystemarkitektur och aktörssamverkan som förutsättningar för affärsprocesser, 2001.

No 894 **Stefan Svarén:** Styrning av investeringar i divisionaliserade företag - Ett koncernperspektiv, 2001.

No 906 **Lin Han**: Secure and Scalable E-Service Software Delivery, 2001.

No 917 **Emma Hansson:** Optionsprogram för anställda - en studie av svenska börsföretag, 2001.

No 916 **Susanne Odar:** IT som stöd för strategiska beslut, en studie av datorimplementerade modeller av verksamhet som stöd för beslut om anskaffning av JAS 1982, 2002.

FiF-a-49 **Stefan Holgersson:** IT-system och filtrering av verksamhetskunskap - kvalitetsproblem vid analyser och beslutsfattande som bygger på uppgifter hämtade från polisens IT-system, 2001.

FiF-a-51 **Per Oscarsson:**Informationssäkerhet i verksamheter - begrepp och modeller som stöd för förståelse av informationssäkerhet och dess hantering, 2001.

No 919 **Luis Alejandro Cortes:** A Petri Net Based Modeling and Verification Technique for Real-Time Embedded Systems, 2001.

No 915 **Niklas Sandell:** Redovisning i skuggan av en bankkris - Värdering av fastigheter. 2001.

No 931 **Fredrik Elg:** Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konfidens i kontroll av mikrovärlden Moro, 2002.

No 933 **Peter Aronsson:** Automatic Parallelization of Simulation Code from Equation Based Simulation Languages, 2002.

No 938 **Bourhane Kadmiry**: Fuzzy Control of Unmanned Helicopter, 2002.

No 942 **Patrik Haslum**: Prediction as a Knowledge Representation Problem: A Case Study in Model Design, 2002.

No 956 **Robert Sevenius:** On the instruments of governance - A law & economics study of capital instruments in limited liability companies, 2002.

FiF-a 58 **Johan Petersson:** Lokala elektroniska marknadsplatser - informationssystem för platsbundna affärer, 2002.

No 964 **Peter Bunus:** Debugging and Structural Analysis of Declarative Equation-Based Languages, 2002.

No 973 **Gert Jervan:** High-Level Test Generation and Built-In Self-Test Techniques for Digital Systems, 2002.

No 958 **Fredrika Berglund:** Management Control and Strategy - a Case Study of Pharmaceutical Drug Development, 2002.

FiF-a 61 **Fredrik Karlsson:** Meta-Method for Method Configuration - A Rational Unified Process Case, 2002.

No 985 **Sorin Manolache:** Schedulability Analysis of Real-Time Systems with Stochastic Task Execution Times, 2002.

No 982 **Diana Szentiványi:** Performance and Availability Trade-offs in Fault-Tolerant Middleware, 2002.

No 989 **Iakov Nakhimovski:** Modeling and Simulation of Contacting Flexible Bodies in Multibody Systems, 2002.

No 990 **Levon Saldamli:** PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations, 2002.

No 991 **Almut Herzog:** Secure Execution Environment for Java Electronic Services, 2002.

No 999 **Jon Edvardsson:** Contributions to Program- and Specification-based Test Data Generation, 2002

No 1000 **Anders Arpteg:** Adaptive Semi-structured Information Extraction, 2002.

No 1001 **Andrzej Bednarski:** A Dynamic Programming Approach to Optimal Retargetable Code Generation for Irregular Architectures, 2002.

No 988 **Mattias Arvola:** Good to use! : Use quality of multi-user applications in the home, 2003.

FiF-a 62 **Lennart Ljung:** Utveckling av en projektivitetsmodell - om organisationers förmåga att tillämpa projektarbetsformen, 2003.

No 1003 **Pernilla Qvarfordt:** User experience of spoken feedback in multimodal interaction, 2003.

No 1005 **Alexander Siemers:** Visualization of Dynamic Multibody Simulation With Special Reference to Contacts, 2003.

No 1008 **Jens Gustavsson:** Towards Unanticipated Runtime Software Evolution, 2003.

No 1010 **Calin Curescu:** Adaptive QoS-aware Resource Allocation for Wireless Networks, 2003.

No 1015 **Anna Andersson:** Management Information Systems in Process-oriented Healthcare Organisations, 2003.

No 1018 **Björn Johansson:** Feedforward Control in Dynamic Situations, 2003.

No 1022 **Traian Pop:** Scheduling and Optimisation of Heterogeneous Time/Event-Triggered Distributed Embedded Systems, 2003.

FiF-a 65 **Britt-Marie Johansson:** Kundkommunikation på distans - en studie om kommunikationsmediets betydelse i affärstransaktioner, 2003.

No 1024 **Aleksandra Tešanovic:** Towards Aspectual Component-Based Real-Time System Development, 2003.
No 1034 **Arja Vainio-Larsson**: Designing for Use in a Future Context - Five Case Studies in Retrospect, 2003.
No 1033 **Peter Nilsson:** Svenska bankers redovisningsval vid reservering för befarade kreditförluster - En studie vid införandet av nya redovisningsregler, 2003.
FiF-a 69 **Fredrik Ericsson:** Information Technology for Learning and Acquiring of Work Knowledge, 2003.
No 1049 **Marcus Comstedt:** Towards Fine-Grained Binary Composition through Link Time Weaving, 2003.
No 1052 **Åsa Hedenskog:** Increasing the Automation of Radio Network Control, 2003.
No 1054 **Claudiu Duma:** Security and Efficiency Tradeoffs in Multicast Group Key Management, 2003.
FiF-a 71 **Emma Eliason:** Effektanalys av IT-systems handlingsutrymme, 2003.
No 1055 **Carl Cederberg:** Experiments in Indirect Fault Injection with Open Source and Industrial Software, 2003.
No 1058 **Daniel Karlsson:** Towards Formal Verification in a Component-based Reuse Methodology, 2003.
FiF-a 73 **Anders Hjalmarsson:** Att etablera och vidmakthålla förbättringsverksamhet - behovet av koordination och interaktion vid förändring av systemutvecklingsverksamheter, 2004.
No 1079 **Pontus Johansson:** Design and Development of Recommender Dialogue Systems, 2004.
No 1084 **Charlotte Stoltz:** Calling for Call Centres - A Study of Call Centre Locations in a Swedish Rural Region, 2004.
FiF-a 74 **Björn Johansson:** Deciding on Using Application Service Provision in SMEs, 2004.
No 1094 **Genevieve Gorrell:** Language Modelling and Error Handling in Spoken Dialogue Systems, 2004.
No 1095 **Ulf Johansson:** Rule Extraction - the Key to Accurate and Comprehensible Data Mining Models, 2004.
No 1099 **Sonia Sangari:** Computational Models of Some Communicative Head Movements, 2004.
No 1110 **Hans Nässla:** Intra-Family Information Flow and Prospects for Communication Systems, 2004.
No 1116 **Henrik Sällberg:** On the value of customer loyalty programs - A study of point programs and switching costs, 2004.
FiF-a 77 **Ulf Larsson:** Designarbete i dialog - karaktärisering av interaktionen mellan användare och utvecklare i en systemutvecklingsprocess, 2004.
No 1126 **Andreas Borg:** Contribution to Management and Validation of Non-Functional Requirements, 2004.
No 1127 **Per-Ola Kristensson:** Large Vocabulary Shorthand Writing on Stylus Keyboard, 2004.
No 1132 **Pär-Anders Albinsson:** Interacting with Command and Control Systems: Tools for Operators and Designers, 2004.
No 1130 **Ioan Chisalita:** Safety-Oriented Communication in Mobile Networks for Vehicles, 2004.
No 1138 **Thomas Gustafsson:** Maintaining Data Consistency im Embedded Databases for Vehicular Systems, 2004.
No 1149 **Vaida Jakoniené:** A Study in Integrating Multiple Biological Data Sources, 2005.
No 1156 **Abdil Rashid Mohamed**: High-Level Techniques for Built-In Self-Test Resources Optimization, 2005.
No 1162 **Adrian Pop:** Contributions to Meta-Modeling Tools and Methods, 2005.
No 1165 **Fidel Vascós Palacios:** On the information exchange between physicians and social insurance officers in the sick leave process: an Activity Theoretical perspective, 2005.
FiF-a 84 **Jenny Lagsten:** Verksamhetsutvecklande utvärdering i informationssystemprojekt, 2005.
No 1166 **Emma Larsdotter Nilsson:** Modeling, Simulation, and Visualization of Metabolic Pathways Using Modelica, 2005.
No 1167 **Christina Keller:** Virtual Learning Environments in higher education. A study of students' acceptance of educational technology, 2005.
No 1168 **Cécile Åberg:** Integration of organizational workflows and the Semantic Web, 2005.
FiF-a 85 **Anders Forsman:** Standardisering som grund för informationssamverkan och IT-tjänster - En fallstudie baserad på trafikinformationstjänsten RDS-TMC, 2005.
No 1171 **Yu-Hsing Huang:** A systemic traffic accident model, 2005.
FiF-a 86 **Jan Olausson:** Att modellera uppdrag - grunder för förståelse av processinriktade informationssystem i transaktionsintensiva verksamheter, 2005.
No 1172 **Petter Ahlström**: Affärsstrategier för seniorbostadsmarknaden, 2005.
No 1183 **Mathias Cöster**: Beyond IT and Productivity - How Digitization Transformed the Graphic Industry, 2005.
No 1184 **Åsa Horzella**: Beyond IT and Productivity - Effects of Digitized Information Flows in Grocery Distribution, 2005.
No 1185 **Maria Kollberg**: Beyond IT and Productivity - Effects of Digitized Information Flows in the Logging Industry, 2005.
No 1190 **David Dinka**: Role and Identity - Experience of technology in professional settings, 2005.
No 1191 **Andreas Hansson**: Increasing the Storage Capacity of Recursive Auto-associative Memory by Segmenting Data, 2005.
No 1192 **Nicklas Bergfeldt:** Towards Detached Communication for Robot Cooperation, 2005.
No 1194 **Dennis Maciuszek:** Towards Dependable Virtual Companions for Later Life, 2005.
No 1204 **Beatrice Alenljung:** Decision-making in the Requirements Engineering Process: A Human-centered Approach, 2005
No 1206 **Anders Larsson:** System-on-Chip Test Scheduling and Test Infrastructure Design, 2005.
No 1207 **John Wilander:** Policy and Implementation Assurance for Software Security, 2005.
No 1209 **Andreas Käll:** Översättningar av en managementmodell - En studie av införandet av Balanced Scorecard i ett landsting, 2005.
No 1225 **He Tan:** Aligning and Merging Biomedical Ontologies, 2006.
No 1228 **Artur Wilk:** Descriptive Types for XML Query Language Xcerpt, 2006.
No 1229 **Per Olof Pettersson:** Sampling-based Path Planning for an Autonomous Helicopter, 2006.
No 1231 **Kalle Burbeck:** Adaptive Real-time Anomaly Detection for Safeguarding Critical Networks, 2006.
No 1233 **Daniela Mihailescu:** Implementation Methodology in Action: A Study of an Enterprise Systems Implementation Methodology, 2006.
No 1244 **Jörgen Skågeby:** Public and Non-public gifting on the Internet, 2006.
No 1248 **Karolina Eliasson:** The Use of Case-Based Reasoning in a Human-Robot Dialog System, 2006.
No 1263 **Misook Park-Westman:** Managing Competence Development Programs in a Cross-Cultural Organisation-What are the Barriers and Enablers, 2006.
FiF-a 90 **Amra Halilovic:** Ett praktikperspektiv på hantering av mjukvarukomponenter, 2006.
No 1272 **Raquel Flodström**: A Framework for the Strategic Management of Information Technology, 2006.

No 1277 **Viacheslav Izosimov:** Scheduling and Optimization of Fault-Tolerant Embedded Systems, 2006.
No 1283 **Håkan Hasewinkel:** A Blueprint for Using Commercial Games off the Shelf in Defence Training, Education and Research Simulations, 2006.
FiF-a 91 **Hanna Broberg:** Verksamhetsanpassade IT-stöd - Designteori och metod, 2006.
No 1286 **Robert Kaminski:** Towards an XML Document Restructuring Framework, 2006
No 1293 **Jiri Trnka:** Prerequisites for data sharing in emergency management, 2007.
No 1302 **Björn Hägglund:** A Framework for Designing Constraint Stores, 2007.
No 1303 **Daniel Andreasson**: Slack-Time Aware Dynamic Routing Schemes for On-Chip Networks, 2007.
No 1305 **Magnus Ingmarsson:** Modelling User Tasks and Intentions for Service Discovery in Ubiquitous Computing, 2007.
No 1306 **Gustaf Svedjemo**: Ontology as Conceptual Schema when Modelling Historical Maps for Database Storage, 2007.
No 1307 **Gianpaolo Conte**: Navigation Functionalities for an Autonomous UAV Helicopter, 2007.
No 1309 **Ola Leifler**: User-Centric Critiquing in Command and Control: The DKExpert and ComPlan Approaches, 2007.
No 1312 **Henrik Svensson**: Embodied simulation as off-line representation, 2007.
No 1313 **Zhiyuan He:** System-on-Chip Test Scheduling with Defect-Probability and Temperature Considerations, 2007.
No 1317 **Jonas Elmqvist:** Components, Safety Interfaces and Compositional Analysis, 2007.
No 1320 **Håkan Sundblad:** Question Classification in Question Answering Systems, 2007.
No 1323 **Magnus Lundqvist**: Information Demand and Use: Improving Information Flow within Small-scale Business Contexts, 2007.
No 1329 **Martin Magnusson:** Deductive Planning and Composite Actions in Temporal Action Logic, 2007.
No 1331 **Mikael Asplund:** Restoring Consistency after Network Partitions, 2007.