# FAULT-TOLERANT REAL-TIME SYSTEMS
*The Problem of Replica Determinism*

# FAULT-TOLERANT REAL-TIME SYSTEMS
*The Problem of Replica Determinism*

*by*

**Stefan Poledna**
*Technical University Vienna*

*Foreword by*
**H. Kopetz**

**KLUWER ACADEMIC PUBLISHERS**
**Boston / Dordrecht / London**

**Distributors for North America:**
Kluwer Academic Publishers
101 Philip Drive
Assinippi Park
Norwell, Massachusetts 02061 USA

**Distributors for all other countries:**
Kluwer Academic Publishers Group
Distribution Centre
Post Office Box 322
3300 AH Dordrecht, THE NETHERLANDS

**Library of Congress Cataloging-in-Publication Data**

A C.I.P. Catalogue record for this book is available
from the Library of Congress.

*for Hemma*

# Contents

# List of Figures

# List of Tables

# Foreword

# by H. Kopetz

*Technical University of Vienna*

Hard real-time computer systems, i.e. real-time systems where a failure to meet a deadline can cause catastrophic consequences, are replacing an increasing number of conventional mechanical or hydraulic control systems, particularly in the transportation sector. The vastly expanded functionality of a digital control system makes it possible to implement advanced control algorithms that increase the quality of control far beyond the level that is achievable by a conventional control system.

Computer controlled fly-by-wire technology has been applied widely in the military sector. This technology is now gaining acceptance in the civilian sector as well, such as the digital flight control systems of the Airbus A320 or the Boeing B777 airplane. In these applications, the safety of the plane depends on the reliability of the real-time computer system. A similar development can be observed in the automotive sector. After the successful deployment of computer technology in non safety-critical automotive applications, such as body electronics, the computer control of core vehicle functions, such as engine, brakes, or suspension control is being considered by a number of automotive companies. The benefits that are expected to result from the advanced digital control of core vehicle functions in the automobile are impressive: increased stability and safety of the vehicle, improved fuel efficiency, reduced pollution, etc., that will lead to a safer, more economical, and more comfortable vehicle operation. The mass market of the automotive sector—more than 50 million vehicles are produced worldwide every year—is expected to lead to very cost effective highly integrated computer system solutions that will have a dominant influence on many of the other real-time computer applications. It is therefore expedient to develop new real-time computer system architectures within the constraints given by the automotive applications.

In safety critical applications, such as a drive by wire system, no single point of failure may exist. At present the approach to computer safety in cars is approached at two levels. At the basic level a mechanical system provides the proven safety level that is considered sufficient to operate the car. The computer system provides optimized performance on top of the basic mechanical system. In case the computer system fails cleanly, the mechanical system takes over. Consider, for example, an Anti-lock Braking System (ABS). If the computer fails, the "conventional" mechanical brake system is still operational. In the near future, this approach to safety may reach its limits for two reasons:

(1) If the performance of the computer controlled system is further improved, the "distance" between the performance of the computer controlled system and the performance of the basic mechanical system is further increased. A driver who gets used to the high performance of the computer controlled system might consider the fall-back to the inferior performance of the mechanical system already a safety risk.

(2) The improved price/performance of the microelectronic components will make the implementation of fault-tolerant computer systems cheaper than the implementation of mixed (computer/mechanical) systems. Thus there will be a cost pressure to eliminate the redundant mechanical system.

A solution out of this dilemma is the deployment of a fault-tolerant computer system that will provide the specified service despite a failure of any one of its components. This leads naturally to domain of distributed fault-tolerant hard real-time systems. The stringent timing constraints in many automotive applications—in the millisecond range or below—require the implementation of actively redundant systems. In actively redundant systems the failure in any one of the redundant computational channels is masked immediately by the availability of a set of correct results. A necessary prerequisite for the implementation of active redundancy is the systematic solution of the problem of replica determinism: the assurance that all replicated channel will visit the same computational states at about the same point in time.

The property of replica determinism is also important from the point of view of testability. The sparse time-base of a replica-determinate real-time system makes it possible to specify exactly every test case in the domains of time and value. The reproducibility of the test results, which is a consequence of replica determinism, simplifies the validation of concurrent systems significantly.

The topic of this book, which is a revised version of a Ph.D. Thesis submitted at the Technical University of Vienna, is the systematic treatment of the problem of replica determinism in fault-tolerant real-time systems within the constraints given by the automotive environment. Some of the more formal chapters of the thesis are not included in this work and can be found in the original document. To familiarize the reader with the selected application domain, a special chapter—chapter two—has been introduced that explains the problems and constraints in the field of automotive electronics in some detail.

It was the goal of this research work to find theoretically sound system solutions to the problem of replica determinism that can be implemented within the economic and technical constraints of the automotive industry. This resulted in the formulation of a set of challenging research problems, e.g., the question about the minimal amount of information that has to be exchanged between two replicated computers in order to agree on a single view of the world. The exact specification and systematic analysis of these problems and the presentation of efficient solutions to these problems are a major contribution to the art of designing fault-tolerant hard real-time systems.

We hope that his book will be of great value to designers and implementers of hard real-time computer systems in industry, as well as to students studying the field of distributed fault-tolerant real-time computing.

*H. Kopetz*

# Preface

Real-time computer systems are very often subject to dependability requirements because of their application areas. Fly-by-wire airplane control systems, control of power plants, industrial process control systems and others are required to continue their function despite faults. Therefore, fault-tolerance and real-time requirements constitute a kind of *natural* combination in process control applications. Systematic fault-tolerance is based on redundancy which is used to mask failures of individual components. The problem of replica determinism thereby is to assure that replicated components show consistent behavior in the absence of faults. It might seem trivial that, given an identical sequence of inputs, replicated computer systems will produce consistent outputs. Unfortunately, this is not the case. The problem of replica non-determinism and the presentation of its possible solutions is the subject of the present work.

The field of automotive electronics is an important application area of fault-tolerant real-time systems. Systems like anti-lock braking, engine control, active suspension or vehicle dynamics control have demanding real-time and fault-tolerance requirements. These requirements have to be met even in the presence of very limited resources since cost is extremely important. Because of its interesting properties this work gives an introduction to the application area of automotive electronics. The requirements of automotive electronics are a topic in the remainder of this work for discussion and are used as a benchmark to evaluate solutions to the problem of replica determinism. The introductory chapter on automotive electronics is self-contained and can be read independently of the remaining chapters.

Following the chapter on automotive electronics a short presentation of the system model and related terminology for fault-tolerant real-time systems is given. This chapter starts the second part of the book which discusses the problem of replica determinism and possible solutions to the problem. First, a generally applicable definition of replica determinism is introduced. Based on this definition possible modes of non-deterministic behavior are then presented. For system design a characterization of *all* the sources of replica non-determinism is important. Such a characterization is given.

The fact that computer systems behave non-deterministically raises the question as to what the appropriate methodologies and implementations for replica determinism enforcement are. This problem is discussed with consideration to different aspects such as communication, synchronization, failures and redundancy preservation. Finally, the problem of replica determinism enforcement is discussed for automotive electronics and systems that have to respond within a short latency period. It is shown that the replication strategies active replication, semi-active replication and passive replication cannot fulfill the given requirements. For that reason two new methodologies are introduced. Firstly, a communication protocol for agreement on external events with a minimum amount of information and, secondly, the concept of timed messages is introduced which allows efficient use of preemptive scheduling in replicated systems. By applying the newly presented methodologies it is shown that replication and systematic fault-tolerance can be used in the area of automotive electronics.

This work is a revised version of my dissertation which was submitted to the Technical University of Vienna in April 1994. Greater emphasis has been especially placed on the application area of automotive electronics. This work has been supported, in part, by the ESPRIT Basic Research Project 'Predictably Dependable Computing Systems' PDCS II.

The valuable support of a number of people have made this work possible. Foremost, I would like to thank my thesis advisor Prof. Herman Kopetz for his many useful suggestions and the interesting discussions which were most fruitful for this work and which served to further my scientific interest. I would also, like to thank him for providing the foreword to this book. Furthermore, my gratitude goes to my colleagues at the Technical University of Vienna and at Robert Bosch. In addition I would like to thank Patrick Henderson and Christopher Temple for their willingness to proofread the manuscripts. Last but not least, I would like to make special mention of the great support and valuable inputs given to me by my friend Ralf Schlatterbeck.

Comments and suggestions concerning the book will be welcomed and can be sent to me by e-mail at stefan@vmars.tuwien.ac.at.

*Stefan Poledna*

# Chapter 1

# Introduction

Computer systems are increasingly used for the monitoring and control of physical processes. Examples of such applications are automotive electronics, *fly-by-wire* airplane control systems, industrial process control systems, control of power plants, robotics and others. Computer systems in these application areas are called *real-time systems*. They are not only required to deliver correct results but also to deliver timely results. The requirement for timeliness is dictated by the dynamics of the physical process to be monitored or controlled. Hence the computer system has to react to relevant state changes of the physical process within a guaranteed response time. Each relevant state change of the physical process represents a stimulus for the computer system which has to be served. Many application areas of real-time systems dictate that the services delivered by such a system are dependable.

The application area of automotive electronics is a very interesting example of real-time systems with high dependability requirements. Systems like anti-lock braking, engine control, active suspension or vehicle dynamics control have demanding safety and reliability requirements. The safety requirements for example are comparable to the already very high requirements of avionics.[1] Furthermore, there are hard real-time requirements that have to be met with minimal resources, since cost is of eminent importance. Another interesting aspect is that automotive electronics are becoming the largest application area for fault-tolerant real-time systems, if volume is considered. Because of its importance, an introduction to the application area of automotive electronics is given in this book. The requirements of this application area are used in the remainder of this book for the discussion and evaluation of solutions to the problem of replica determinism. It is also argued that systematic fault-tolerance, which is based on replication, has very attractive properties for automotive electronics.

In general, two principle approaches may be taken to satisfy the ever increasing demands for dependable computer systems: The first approach, *fault-avoidance*, entails the construction of computer systems from components that are less likely to fail. The second approach, *fault-tolerance*, involves constructing computer systems that continue to deliver their service despite faults [Lap92]. The fault-avoidance ap-

[1] cf. chapter 2.

proach proved to be fruitful in the early days of computing [Car87] because technology of hardware components was immature and could be improved by orders of magnitude. In addition there were no users who fully relied on the proper function of computer systems. In fact, a computer's result was suspected to be faulty. With the advent of computer technology, the limitations of the fault-avoidance approach are becoming evident. It is impossible to build single components that are able to guarantee the necessary level of dependability which is required for today's critical application areas. Furthermore, for complex systems the degree of fault detection and fault removal is not sufficiently high.

The second approach to dependability is fault-tolerance. This technique is aimed at providing a service complying with the specified dependability in spite of faults. Systematic fault-tolerance requires redundancy to mask failures of individual components transparently. The necessary redundancy can be provided either in the domain of information, time or space. Redundancy in the domain of space is called replication. Most application areas with high dependability requirements dictate replication because the system would otherwise rely on the correct function of any single component. The idea of replication in computer systems dates back at least to von Neumann who mentioned this principle explicitly in [Neu56]. Replication is based on the assumption that individual components are affected by faults independently or at least partially independently. These properties are achieved best by distributed computer systems where nodes communicate through message passing. Furthermore, it has to be guaranteed that faulty components are not able to disturb the communication of correctly functioning components. Faulty components are not allowed to *contaminate* the system [GT91, KGR89] by sending faulty messages which are then incorporated into the state of non-faulty components. The only shared resource of such a system is the communication media. All other components can be implemented physically and electrically isolated from one other. In general, replication may take place at different hard- or software levels. For example, non-identical replication of functional equivalent software components is not called replication, but $N$-version programming [Avi77].

If a given service is requested from a set of replicas then the result should be correct even if some of the replicas have failed, given that the number of failed replicas is below a threshold. The problem of *replica determinism* thereby is to assure that no unwanted side effects lead to disagreement among correct replicas. At a first glance it seems trivial to fulfill this property since computer systems are assumed to be par excellence examples of deterministic behavior i.e. given the same sequence of service requests, a set of computers will produce identical outputs. The following example, however, shows that even two correct computers may come to diverging conclusions: Consider two computers which are used to control an emergency valve that has to be closed in case of excess pressure. Both read an analogue pressure sensor, but due to the limited accuracy of the sensors they receive slightly different readings. As a consequence the individual computers may derive diverging decisions. One computer may conclude to shut the emergency valve, while another decides to keep the valve open. This shows that simple replication of components does not lead to a *composed* component which provides the same service as a single component with a higher degree of dependability.

This is not only the case with inconsistent sensor readings, as the example above suggests. Consider the same example. It is however assumed that the sensor readings of both computers are identical. But the specification additionally requires that the computer systems have to close the emergency valves if a somewhat lower level of pressure is exceeded for a given period. Due to minor clock speed differences it may happen that one computer decides to timeout and close the valve while the other does not timeout and decides to keep the valve open. Again the individual decisions of the computers diverge.

The problem of replica determinism is not limited to systems where a set of computers is acting in parallel, as the above given examples might indicate. In systems with standby replication the problem of replica determinism may also arise. Consider again the example of two computers which have to shut an emergency valve in case of excess pressure. Additionally it is assumed that depending on the actual processing mode of the plant there are two different thresholds for excess pressure. Computer 1 is active and knows that the actual threshold for excess temperature is $p_1$ due to the actual process mode. After some time computer 1 fails and computer 2 has to take over. But the information on the actual process mode in computer 2 is inconsistent. Computer 2's decision on over pressure is therefore based on another threshold, namely $p_2$. Again both computers take inconsistent decisions. However, this does not happen at the same time in parallel, but sequentially. In addition to the examples given, there are other sources for replica non-determinism which will be treated in a later chapter. From these observations it follows that replica determinism is an important property for systematic fault-tolerance.

Independent of the fact whether a system uses replication or not, there is an important relation between replica determinism and testing. One possible approach to system testing is to compare the behavior of the test object against the behavior of a *golden* system. The golden system is assumed to be correct. Failures of the test object are uncovered by detecting a divergence between the behavior of the test object and the golden system. This approach to testing is only applicable if replica determinism is guaranteed since otherwise there would be not much significance in the diverging behavior. The same is true if log files of known correct data are compared against the actual behavior of the system to be tested. Replica determinism is therefore also advantageous for the testability of systems [Sch93c].

Replication is not only used as a technique to achieve fault-tolerance, but it is also used to improve performance. In this case the term *parallel computing* is used instead of replication. The architecture of parallel computing systems is different from replicated fault-tolerant systems because the components of parallel computing systems are not required to fail independently. It is therefore not necessary to isolate components physically and electrically. While replicated systems are based on the paradigm of message passing, parallel computing systems typically use shared memory. The problem of replica determinism is also of relevance for parallel computing systems since the individual processors need to share common information consistently. In parallel computing systems, however, instead of replica determinism the term *coherency* (*cache-* or *memory coherency*) [TM93] is used. Due to the architectural differences and the fact that replicated systems are based on the assumption of independent failures, the solutions to the problem of replica determinism are different for replicated and parallel computing systems.

Another related area where the problem of replica determinism is of relevance is that concerning distributed data base systems [BG92]. These systems manage large amounts of data which are distributed over physically separate locations. Again fault-tolerance is not the major concern. The distributedness reflects the need for distributed access to data while maintaining high performance on data access. The primary difference between distributed data base systems and distributed fault-tolerant real-time systems is that the aspect of time is considered from a different point of view. Distributed data base systems are throughput oriented while real-time systems are oriented towards guaranteed response times. Another difference is that data base systems are concerned with large amounts of long-lived data which is processed in transactions, while real-time systems typically operate on small amounts of data which are invalidated by the passage of time.

The remainder of this book concentrates on the problem of replica determinism in fault-tolerant real-time systems. In addition to general aspects, special consideration is given to the requirements of automotive electronics.

## 1.1 Goal of this book

The main objective of this book is to investigate the problem of replica determinism in the context of distributed fault-tolerant real-time systems. The importance of replica determinism is established by the fact that almost any fault-tolerant computer system has to provide a solution to this problem. To date however, in many cases the problem of replica determinism is treated in an unstructured and application specific fashion by ad hoc engineering solutions. This book will give an introduction to the field of automotive electronics, which is an important application area for fault-tolerant real-time systems. Identified requirements of this application area are used in the following as a reference to discuss the problem of replica determinism and different solution strategies.

It is important to understand the underlying sources and effects of replica non-determinism to find systematic and adequate solutions. In order to do so, a notion of replica determinism first of all will be introduced. Currently, there are no agreed upon definitions for replica determinism. Many of them are also restricted to certain application areas or are not well suited for real-time systems. It is therefore an aim of this work to introduce an application independent definition for replica determinism. In a next step possible sources of replica non-determinism are investigated and the effects that they cause. Furthermore, the possibility of avoiding replica non-determinism is discussed. Since this is impossible—as will be shown—the basic limitations of replication are discussed. At present there is no characterization of the fundamental limitations of replica determinism. This book intends to contribute to that problem by giving a basic characterization of replica non-deterministic behavior.

Due to these limitations the enforcement of replica determinism is a necessity for replicated fault-tolerant systems. The enforcement of replica determinism is concerned with the information-exchange on non-deterministic states. Therefore, the relation between replica determinism enforcement and different communication protocols is surveyed. Furthermore, the close interdependence between replica determinism enforcement on the one hand, and architectural properties such as synchronization strategies, handling of failures and redundancy preservation on the other hand are discussed. Another important aspect that is given consideration is the synchronousness or asynchronousness of the systems.

Finally, it is the goal of this book to investigate the problem of replica determinism enforcement in the context of automotive electronics and systems which are required to respond within a very short latency period. Currently used techniques for replica determinism enforcement are not very suitable for these systems, because the coordination effort in terms of execution time and communication is too high. This book contributes to the problem of replica determinism enforcement in fault-tolerant real-time systems by minimizing the necessary effort.

## 1.2 Overview

This book is organized as follows: The following chapter will give a short introduction to the application area of automotive electronics. This application area has been selected to evaluate the problem of replica determinism enforcement under practice conditions. Automotive electronics have demanding requirements on important aspects such as real-time performance, dependability, responsiveness and efficiency.

Chapter 3 defines the system model and provides the related terminology. To discuss the properties of replication the system structure of real-time systems is presented together with the concept of dependability. This chapter also defines the various failure modes of components and the concepts of failure semantics and assumption coverage. The relation between replica groups and failure masking is also outlined. Other important properties discussed in this chapter are the synchronousness and asynchronousness of systems.

Chapter 4 discusses the basics of replica determinism and non-determinism. The first section examines the problems of different definitions for replica determinism and gives an application independent definition. Based on this definition various modes of non-deterministic behavior and their possible consequences are considered. The fundamental limitations to replication are introduced which give a classification for the various sources of replica non-deterministic behavior. It will be shown that it is impossible to avoid the problem of replica determinism in practical real-time systems.

In chapter 5 principle approaches to replica determinism enforcement and their requirements are given. A characterization of different approaches is be presented according to the criteria whether replica determinism is enforced group internally or externally, and whether the enforcement strategy is central or distributed. The close interdependence between replica determinism and basic properties of distributed fault-tolerant real-time systems are discussed. These properties are communication, synchronization, handling of failures and redundancy preservation.

Chapter 6 discusses the problem of replica determinism enforcement for automotive electronics. Different replication enforcement strategies and their effect on the latency periods are discussed. To improve the latency period an optimized communication protocol to handle non-deterministic external events is presented. An efficient solution to the problem of preemptive scheduling is also introduced.

Finally, chapter 7 concludes this work and provides a summary.

# Chapter 2

# Automotive electronics

This chapter will give a brief introduction to the application area of automotive electronics, which was selected as an application example to evaluate current solutions to the problem of replica non-determinism. Automotive electronics has been chosen because of its demanding requirements for real-time performance and dependability which have to be met in the presence of very limited resources. Furthermore, automotive applications are becoming one of the largest application areas of fault-tolerant real-time systems, with respect to the number of units in service. Functions ranging from anti-lock braking, engine control, active suspension to vehicle dynamics control are handled by these computer systems. Failures of these systems can cause loss of control over a vehicle with severe consequences to property, environment, or even a human life. Besides safety requirements there are also obvious reliability and availability requirements. Development trends indicate that future road vehicles will be equipped with distributed fault-tolerant computer systems, e.g. [GL91] and that up to 30% of the total cost will be allocated to electronic systems.

Current high-end systems typically have one to three processors with an accumulated processing speed of up to 10 million instructions per second, with approximately 256 *kbytes* of program memory, 32 *kbytes* of calibration data, 16 *kbytes* of RAM and 100 – 160 I/O pins. With the advent of semiconductor technology in the next years a dramatic performance improvement can be expect in the area of automotive electronics. Processing speeds of 100 million instructions per second and memory sizes of up to 1 *Mbyte* are coming into reach which will allow a higher level of functionality and implementation of new functions like model based adaptive control algorithms.

The following section describes the general application characteristics of automotive electronics. Special consideration is given to the stringent efficiency requirements which are dictated by hardware cost limitations in this application area. This requirement for efficiency and good resource utilization will be reflected throughout the book by putting emphasis on the complexity of replica determinism enforcement strategies. Section two describes the functional requirements of automotive applications with particular attention to engine control. In contrast to section one which describes the application characteristics from an external point of view this sections describes the internal requirements of the computer system. Section three describes dependability aspects. Special consideration is put on safety critical systems which will find more and more application. Finally, section four discusses present problems and future directions to achieve the required dependability goals. The two alternatives application-specific and systematic fault-tolerance are presented. It is argued that systematic fault-tolerance has favorable properties and thus will likely be selected for future systems if it is possible to solve the problem of replica determinism efficiently.

## 2.1 Application area characteristics

Cost is one of the strongest determining factors in the area of automotive electronics. Among car manufacturers as well as electronic suppliers there is a strong competition on the basis of cost. Since automotive electronics is a volume market—similar to consumer electronics—the most important cost factors are not development and design cost but production cost. Dependent on product and quantity up to 95% of the total cost accounts for production and material while as little as 5% accounts for development and design. These numbers show that high efficiency and good utilization of hardware resources is very important for computer systems in the area of automotive electronics. Consequently, if fault-tolerance and replication strategies are to be applied, then efficiency is of utmost concern.

Furthermore, there is a supplementary point of view that shows the criticality of efficiency and resource utilization in the area of automotive electronics. By segmenting automotive electronic products according to their functionality into three categories, the strong influence of price becomes even more apparent. The three functional categories are [WBS89]:

- ***Core vehicle functionality electronics:*** These functions are responsible for controlling or enhancing the basic vehicle functions such as acceleration, braking, steering, etc. The core vehicle functionality electronics allows to provide more functionality than would be possible with its mechanical counterparts. For example, electronic engine management allows to meet stringent emission regulations which are established by legislation.

- ***Feature electronics:*** Feature electronics are products which do not perform basic vehicle functions. They rather provide benefits to the driver in terms of comfort, convenience, entertainment or information. A typical example of this category is a car CD-player.

- ***System level electronics:*** This functional category provides system level services which are not perceivable at the level of the vehicle user. Examples are multiplexing buses for information exchange between electronic control units. The driving force behind this functional category are internal considerations of the electronics supplier and the car manufacturers such as cost and dependability. Replica determinism enforcement strategies have to be considered as system level electronics.

Out of these three categories only feature electronics is directly visible to the driver. Hence, this is the only category that is directly paid by the customer. Core vehicle functionality electronics are only partially visible, they may provide improvements in terms of comfort, safety or fuel economy. They are therefore only partially paid by the customer. System level electronics, however, are invisible to the customer. Thus, they can only be applied if they allow to achieve improvements in dependability or by reducing system complexity while maintaining or reducing the cost compared to traditional mechanical solutions. For fault-tolerance and replica determinism enforcement strategies which are categorized as system level electronics this has the consequence that they can be applied only if they are extremely efficient so that no, or scarcely any, additional costs are introduced.

## 2.2 Functional requirements

While the previous section gave an overview of the application area requirements, the aim of this section is to give a presentation of the specific functional requirements for automotive electronic systems. Particular consideration is given to the fact that automotive electronics dictates short latency periods.

The short latency periods are dictated by the fact that many control systems have to follow rotational movements of shafts with short and bounded delay times, e.g., the rotation of the engine or the cars wheels. Control systems for combustion engines have to control the timing of fuel injection and ignition. For fuel injection the engine control system has to start the injection at a certain crank angle position for each cylinder with a high temporal accuracy. The duration of the injection is determined by the required fuel quantity. Both the angle for injection start as well as the fuel quantity are control conditions of the engine controllers. The ignition timing is also a control condition which is set for a certain crank angle position. The engine control system is required to set these control conditions for each cylinder. Hence the processing of the engine control units is not only determined by the progression of real-time, but also by the crank angle position of the engine. Figure 2-1 shows a timing diagram with the injections marked as high levels and ignitions marked as down-arrows. The timing of the events shown in Figure 2-1 depends on the actual speed of the engine. For combustion engines a speed range of 50 *rpm* up to 7000 *rpm* has to be considered. Hence, dependent on the engine speed the frequency of these events varies by more than two orders of magnitude. Resolving the fuel injection and ignition timing to a crank shaft angle of $0.1°$ requires a real time resolution of at least $2\ \mu s$. While the injection and ignition timing is controlled by the crank angle position, there are other services which are controlled by the progression of real-time.

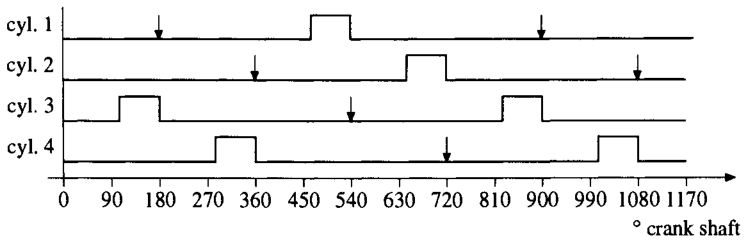

*Figure 2-1: Injection and ignition timing*

This control of the processing pace by real-time and by crank angle can be explained by the physics of combustion engines and by control theory [CC86].[2] Services, whose pacing is controlled by real-time only, are in essence *time-triggered* and have periodic activation patterns. On the other hand, services whose pacing is controlled by the crank angle or by some other a priori unknown events are *event-triggered.* The following list gives key requirements for electronic control units with particular emphasis on engine control systems:

- ***Hard real-time:*** Engine control systems are responsible for the timing of fuel injection and ignition. Injection and ignition timing are critical parameters which have to be guaranteed with high precision. Furthermore, there are some controlled conditions that have to be evaluated before start of injection and start of ignition. In the case of a missed deadline, these controlled conditions are not available timely. Timing failures not only cause undesired behavior such as high emissions but may also cause damage to the engine. This application area is considered to be hard real-time because missing a deadline has severe consequences on the environment. Other automotive applications such as anti-lock braking systems, electronic gear-box control, active suspension or vehicle dynamics control also have hard real-time requirements.

- ***Short latency period:*** Some services of engine control systems are required to respond within very short latency period. These are typically services which are relevant to the control of fuel injection and ignition timing. A service, for example, which calculates an actuating variable that is related to fuel injection has to finish its execution before the injection actually starts. Furthermore, this service should take into account the most recent information for calculating the actuating variable [LLS$^+$91, SKK$^+$86]. Therefore the service has to start only a

[2]A combustion engine is a sampling system where sampling occurs at top dead center with each cylinder that ignites. Hence, the sampling rate varies depending on the engines speed compared to fixed sampling rates in conventional control theory.

short time interval before the injection. For these applications service latency periods are typically in the range of 100 *μs*, which is shown in Figure 2-2.

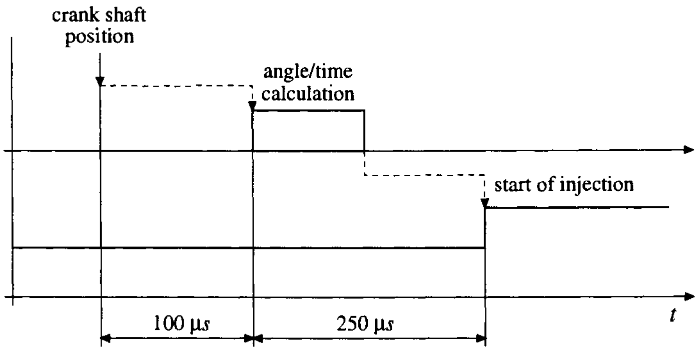

*Figure 2-2: Latency timing*

The first event in Figure 2-2, denoted by a down arrow, informs the computer system that a given crank shaft position has been passed. This event triggers a service which is responsible for programming the output logic for the injection signal. Based on the timing of previous events the service has to predict the timing of the injection start which has to be set for a given crank angle position. To allow timely programming of the output logic, the latency period of the service has to be approximately 100 *μs*. The activation period of this service is determined by the time between two consecutive injections of the same cylinder. Assuming a four stroke engine with a maximum speed of 7000 *rpm*, the minimum time between two injections of the same cylinder is 17.14 *ms*. The service's latency time of 100 *μs* is very short compared to the minimum interarrival period of activations. To achieve good performance under these conditions a combination of event- and time-triggered scheduling has to be used [Pol93, Pol95b].

- ***High activation frequencies:*** In automotive electronics there are functions which require high service activation frequencies. Some control loops require sampling frequencies of up to 4 *kHz*. In engine control applications even higher service activation frequencies are required to monitor the actual position of the crankshaft or camshaft. Some systems resolve one revolution of the engine with an accuracy of 6 degrees[3] [LLS+91]. By considering a maximum engine speed of 7000 *rpm* the time interval between the passage of 6 degree crank shaft is as little as 142 *μs*. This results in a service activation frequency of 7 *kHz* for the service which monitors the crank angle position. For these reasons cumulated ser-

[3] By interpolating between crank angle positions the achievable accuracy of this measurement technique is well below 1 degree crank shaft.

vice activation frequencies of up to 10 *kHz* are assumed. To achieve these high activation frequencies the context switch between different tasks has to be very efficient.

- ***Wide range of activation frequencies:*** Besides the high service activation frequencies, as mentioned above, automotive electronics also has requirements for low service activation frequencies. Services which are responsible for the control of mechanical entities typically have activation frequencies in the range of 10 *Hz* to 100 *Hz*. Monitoring and control of electro-mechanical entities require higher service activation frequencies within the range of 100 *Hz* to 1 *kHz*. The spectrum of service activation frequencies is therefore in the range of 10 *Hz* to 7 k*Hz*, which is nearly three orders of magnitude. Preemptive scheduling of tasks is inevitable in these systems.

- ***Low communication speed:*** While it is a well established practice to use onboard automotive networks for exchange of non-critical information, e.g. [MLM+95], the use of networks in distributed fault-tolerant safety-critical applications is still in the stage of research. Proposed protocols for communication, e.g. CAN [SAE92], support communication speeds of up to 1 *Mbps*. For cost reasons, however, automotive manufactures prefer to use communication speeds of up to 250 *kbps*. Currently, 250 *kbps* is the maximum tolerable rate over a cheap unshielded twisted pair cable where bus termination is uncritical, considering the harsh environment in the area of automotive electronics. Hence, for the remainder of this book it is assumed that the communication speed is limited to 250 *kbps* for automotive applications.

- ***Small service state:*** Typically, individual services in the area of automotive electronics have relatively small service states. The service state consists of local variables which are kept by a task between consecutive activations.[4] Most often these local variables are used to store control parameters. Compared to general purpose applications, there are no databases holding large amounts of modifiable data. Representative sizes of service states in the area of automotive electronics range from 10 *bytes* to 300 *bytes*. This small service state size is mostly determined by the application semantics. However, to a lesser degree it is also determined by implementation considerations concerning hardware restrictions.

- ***Small size of service requests and responses:*** The size of service requests and responses is in the range of 2 *bytes* to 100 *bytes*, which is relatively small. This small size is determined by the fact that typical information exchange between services only concerns controlled conditions, actuating variables and set points. Due to the accuracy of sensors and actuators 16 *bits* are typically

[4]Read only data, which may be of considerable size, is not considered part of the service state.

sufficient for representation of these data. For current high end control units the accumulated rate of service request and responses is approximately $10^5$ times per second. There is no exchange of large amounts of data. For example, the large amounts of data for control maps and calibration is stored for read only access and does not have to be exchanged between services.

Together these requirements describe a hard real-time system that has to handle high frequent service requests with the additional requirement to react within short latency periods. There is a small amount of data that is modified with very high frequencies. Compared to general purpose applications the amount of modifiable data is very small because automotive electronic systems are *closed systems*. A closed system has a service specification that is fixed during its operational phase. Such systems benefit from the a priori knowledge about services and hence can store most information as unmodifiable data, e.g. control maps and calibration data.[5]

While on the one hand it is possible to exploit these benefits of a priori knowledge, on the other hand the very demanding requirements have to be considered. The limited communication speed restricts the amount of information that can be exchanged between nodes. This restriction, however, conflicts with the demand to handle high frequent service requests in a distributed real-time system. Furthermore, the high cost sensitivity in the area of system level electronics has to be considered. This leads to the conclusion that automotive electronics have very high efficiency requirements to deliver high performance at low cost.

## 2.3 Dependability requirements

Together with cost and efficiency the second strongest determining factor is dependability and its related attributes reliability, availability, safety and security [Lap92]. Reliability and availability are of abundant importance for automotive electronics. Failure data on vehicle breakdowns collected by independent automotive associations, like the ADAC in Germany, play an important role for vehicle manufacturers. These figures have a big impact on the image of a brand and the customers interest in vehicles of a certain brand. Furthermore, it should be noted that core and system level electronics often replaces functions which previously have been implemented by mechanical systems. These mechanical solutions were highly reliable. It is therefore very difficult to achieve a similar level of reliability for the more complex electronic systems at comparable cost. In addition to the higher complexity, the harsh operating conditions for automotive electronics have to be considered. There are con-

[5]To be accurate it should be noted that there are possibilities to modify control maps and calibration data during an evaluation and tuning phase. This possibility, however, is not implemented by the electronic control systems themselves. Modification of this data is rather carried out by means of special tools which are transparent to the electronic control systems in the car.

tact problems, EMI, mechanical stress on cabling, vibrations, heat or fuel vapors, to mention but a few.

To meet the high dependability requirements under such harsh operating conditions, fault-tolerance has to be applied. Especially, defects in the cabling, sensors and actuators need to be tolerated. The ambitious reliability goals are underpinned by the following estimations (cf. [Kop95]). Based on the figure of one on-road service call per 1000 cars per year which is caused by the electronic system this gives a MTTF (mean time to failure) of approximately $9 \times 10^7$ hours or 114 FIT. This estimation reflects the actual achieved reliability for cars under the assumption that control units are always active. While this assumption is true for some control units others are only active if the car is operated. Based on the average rate of 300 operating hours the MTTF is $3 \times 10^5$ hours or 3333 FIT. By taking the conservative assumption that 10 electronic control units are used per car the MTTF for one control unit inclusive wiring, sensors and actuators will range between 11 and 333 FIT. This numbers give an indication for the already very high reliability standard of automotive electronics.

Recently, security has also become an issue in the area of automotive electronics. There are two main reasons for this. Firstly, theft avoidance devices and car access control are implemented by control units. Secondly, the calibration data and control maps of engine control units needs to be protected against unauthorized manipulations. Such manipulations are carried out by car tuners to increase engine power. Besides violating regulations on pollution these manipulations can cause excessive wear on the cars mechanical components and ultimately destruction of the engine. Security measures against unauthorized access and manipulation of calibration data is therefore implemented by cryptographic algorithms.

Many systems in the area of automotive electronics also have high safety requirements, e.g. anti-lock brakes, engine control, transmission control or active suspension. Anti-lock braking systems for example are able to reduce the brake pressure to avoid that some wheels of the vehicle get locked. If, however, the brake pressure is reduced erroneously then it may happen that the stopping ability of the car is reduced unintentionally. This may lead to severe consequences. Besides high cost even human lives may be endangered. The high safety requirements for these applications compare to the safety requirements for avionics, which are given by the probability of $10^{-9}$ critical incidents per hour [Kop95]. Most of these critical functions are part of the concept of "drive-by-wire". In analogy to the aircraft notion "fly-by-wire" the term drive-by-wire is defined as the control of vehicle functions by computers which process sensor data and control actuators. It should be noted that even some feature electronics which provide comfort are subject to dependability requirements. If, for example a personal programmable rear-view mirror moves during a critical driving maneuver to a wrong position it may happen that the driver fails to notice an approaching car.

Currently legislation, traffic authorities and national road administrations are working on requirements for the assessment of tolerable risk levels in safety critical automotive applications. However, up to now there are no agreed upon requirements. One possibility is a stochastic specification, in analogy to the requirements in critical aircraft applications, e.g. [ARI92]. This would require that the probability of a system failure, classified according to a scheme of severity levels, is below a given threshold. Another possibility is to assign each system to a certain safety class. For each class, rules for design, test, verification, documentation, etc. are specified and the fault-tolerance requirements for the safety class are defined. A third approach has been developed by the working group on vehicle technology of the German traffic authority [Fid89]. According to this approach there are no fixed requirements but a set of rules how to derive requirements for a certain safety-critical application. Additionally, there are requirements on how to document the development process. Regardless of which assessment strategy will be adopted, the functional dependability of safety critical systems has to be guaranteed.

## 2.4 Dependability: Present problems and future directions

This section presents the current state of the art in solutions to achieve dependability and fault-tolerance for automotive electronics. The advantages and disadvantages of these solutions are discussed. Furthermore, directions for future advances and improvements are presented.

To address these dependability requirements, currently almost exclusively application-specific engineering solutions to fault-tolerance are used rather than systematic approaches [Pol95a]. The reason for this is cost. Systematic fault-tolerance requires replication of components to attain redundancy. Past and present automotive computing systems consists of independent control units where each control unit is responsible for a distinct functionality, see Figure 2-3. Although, in some cases there is communication between individual control units by means of dedicated interconnections or by multiplex buses. It is however characteristically for these systems that correct functionality does not depend on this communication. In the worst case, a degraded functionality may result if the communication fails. Therefore, replication would have to be applied to each control unit individually in the case of systematic fault-tolerance. This would impose very high costs. Instead, application-specific solutions to fault-tolerance have been adopted. That is, fault detection is carried out by reasonableness checks and state estimations are used for continued operation.

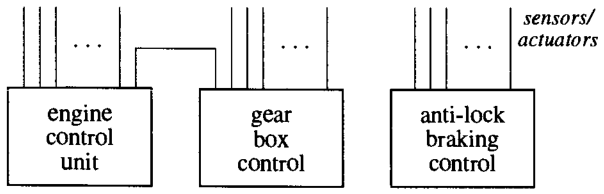

*Figure 2-3: Conventional automotive electronic system*

New developments in safety critical automotive electronics for advanced functionality such as vehicle dynamics control require a more close functional coupling of control units for engine, gearbox, brakes, steering, suspension control and others, see Figure 2-4. These system consists of a set of closely cooperating control units with distributed functionality instead of independent single control units. Coupling is facilitated by means of a real-time multiplex bus, e.g., [KG94] which is used to exchange information by means of messages. There is a transition from individual control units to a truly distributed computer system with real-time and fault-tolerance requirements.

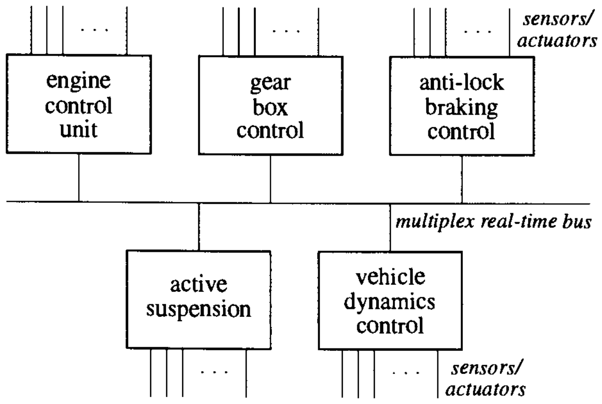

*Figure 2-4: Advanced coupled automotive electronic system*

This structure inherently provides replicated components such as a set of communicating processors. If a processors fails one of the others can act as a replacement. Also, sensor inputs for safety critical functions, such as engine speed, accelerator pedal, brake pedal, brake pressure, and steering angle, are in most cases measured by at least two control units. These signals are also measured by at least two control units in the conventional system, since control units are designed to be independent of each other. Thus, coupling in the advanced coupled system leads to a structure where existing replicated components can be used without the introduction of additional cost, compared to the conventional system. Application-specific and systematic fault-tolerance are not only of relevance to automotive electronics, they are rather two different fundamental approaches to fault-tolerance. A general characterization of these two approaches will be given in the following:

***Application-specific fault-tolerance:*** By application-specific fault-tolerance we subsume methods which uses reasonableness checks to detect faults and state estimations for continued operation despite of faults.

This combination of reasonableness checks and state estimations allows the implementation of fail-operational behavior. By using reasonableness checks alone fail-safe functionality can be implemented. Reasonableness or plausibility checks are based on application knowledge which is used to judge whether a component operates correctly or not. This application knowledge in turn is based on the fact that the computer system interacts with a certain controlled object or physical process and that the behavior of the physical process is constrained by the laws of physics. By implementing these laws of physics, the computer system can check for reasonableness. Examples are signal range checks for analog input signals or checks on the acceleration/deceleration rate of engine or wheel speed. These reasonableness checks enable the detection of faulty behavior without actually requiring replicated components. Once a component failure is detected this information can be used to bring the system into a safe state. Additionally, the state of the failed component can be estimated (since there are no replicated components available) to implement fail-operational behavior. If for example an engine temperature sensor fails then it is possible to estimate the engine temperature by using a model that considers ambient temperature, engine load and thermodynamic behavior of the engine. A simpler possibility is to assume that the engine temperature has some constant fixed value. Another example for state estimation is the calculation of the vehicle speed by taking into account the actual engine speed and the transmission rate of the currently engaged gear. This state estimation can be used in case the vehicle speed sensor fails.

Besides application specific fault-tolerance the second approach is systematic fault-tolerance:

***Systematic fault-tolerance*** is based on replication of components, where divergence among replicas is used as a criterion for fault-detection. Redundant components are used for continued service.

If among a set of replicated components, some—but not all replicated components—fail then there will be disagreement among replicas. This information can be used to implement fail-safe behavior. Systematic fault-tolerance therefore does not uses application knowledge and it takes no assumption on the physical process or the controlled object. Depending on the number of replicas and their failure semantics fail-operational behavior can be implemented if enough correct replicas are available. There are different strategies to implement fail-operational behavior in a replicated system, most notably active replication, semi-active replication and passive replication.[6] Systematic fault-tolerance assumes that some kind of agreement between replicated components is available to decide between correct and faulty components: Correct components should show *corresponding* behavior, while faulty components exhibit *diverging* behavior. Or in other words, the components are required to show replica determinism. But since it is shown in chapter 4 that almost any "real" computer system behaves non-deterministically it is necessary to enforce replica determinism (an example for non-deterministic behavior was already given in the introduction).

The obvious advantage of application-specific fault-tolerance is that no additional costs for replication of components are introduced. On the other side, there are severe disadvantages. Firstly, the fault detection capability of reasonableness checks is limited. There is a *gray zone* [Pol95b] where it is not exactly predictable whether a component is faulty or not. This can lead to cases where faulty components are considered to be correct and vice versa. It also puts a considerable additional burden on the application programmer to evaluate the physical process and to design reasonableness checks. Especially, for highly dynamic systems it is difficult to derive reasonableness checks, since they depend on numerous inputs, state variables, time and possible operating modes [LH94]. Also, there are many cases where no precise mathematical way exists to describe the reasonableness checks. Another problem is that reasonableness checks can only be used for quantities that are related to a physical process. For the typical message traffic in a distributed systems it is impossible to use reasonableness checks, since most of this messages are not related to physical quantities and therefore have no physical semantics. Also, it may be the case that the complexity of exact mathematical reasonableness checks is prohibitively high. Reasonableness checks are therefore frequently based on simplistic assumptions and on empirical data. Often, this leads to a behavior where such a simplistic reasonableness check fails sporadically. It might detect faults although components are correct. As a result the reasonableness checks are designed to be overly tolerant to avoid such false alarms. This, however, leads to a low fault detection coverage. Since these problems with reasonableness checks are likely to occur under peak-load conditions and in rare event scenarios it is extremely difficult to diagnose and correct such problems.

The second problem of application specific fault-tolerance is associated with the quality of state estimations which differs depending on the application area. Again it depends on the availability and complexity of the mathematical model for the physical process to perform state estimations. While it is for example possible to detect a fault of the engine speed sensor by performing reasonableness checks on the engine speed acceleration ratio, it is impossible to estimate the engine speed with the in-

[6]The necessary number of correct replicas for different failure semantics and strategies for fault-tolerance are discussed in chapter 5, "Enforcing Replica Determinism".

formation that is typically available in an engine control unit of a car. The reason for this is that the engine speed depends on a multitude of parameters, like injected fuel quantity, weight of the car, selected gear, engaged or disengaged clutch and many others. Furthermore, in cases where it is possible to perform state estimations, the quality of the state estimation will be lower, compared to fault free operation. Consider the example where the vehicle speed is estimated by engine speed and gear ratio. If the clutch is not engaged it is impossible to estimate the vehicle speed. Also it may happened that the driving wheels are spinning on a slippery surface and that the vehicle speed is therefore estimated incorrectly. Hence, the quality of the state estimation depends on the physical process and the operating conditions.

The third problem is the high complexity which is inherent to application-specific fault-tolerance. Application-specific reasonableness checks as well as state estimations of failed components are typically complex functions by themselves. But additionally, because of their application-specific nature, regular functionality and functionality for fault-tolerance are closely intertwined, which imposes additional complexity. This makes analytic modeling, validation and testing of such systems a tedious endeavor, if not impossible.

The advantages of systematic fault-tolerance are the independence from application knowledge and independence from regularity assumptions which are required for reasonableness checks. There is no need for an exact mathematical model of the physical process. Systematic fault-tolerance is therefore not limited to data that has physical semantics. The only criterion for fault detection is correspondence or divergence between replicas. Since this criterion is simple and void of any application knowledge it can be implemented by a separate functional level for fault-tolerance. With systematic fault-tolerance it is therefore possible to separate the concerns for application functionality and fault-tolerance mechanisms (which allows utilization of one of the most successful design methodologies: divide and conquer). This allows the implementation of fault-tolerance services which are transparent to regular functionality. Hence, systematic fault-tolerance has a lower complexity, compared to application-specific fault-tolerance. Furthermore, the separation of fault-tolerance handling and application functionality allows the reuse of the fault-tolerance mechanisms. In case of application-specific fault-tolerance this is not possible, it is necessary to designed the fault-tolerance mechanisms (reasonableness checks and state estimations) for each application. Modifications of application functionality are for the same reason much simpler in the case of systematic fault-tolerance. With the continuously shortening time-to-market, reuse of fault-tolerance mechanisms with systematic fault-tolerance is of great importance. Also, the separation of fault-tolerance mechanisms and application functionality allows analytic modeling, validation and testing of these levels individually.

The obvious disadvantages of systematic fault-tolerance is cost. Cost of replicated components, cost of replica determinism enforcement and cost of fault-tolerance management. In the area of automotive electronics, with the transition from independent control units to advanced coupled systems and distributed computing, replicated components are becoming available. Also, it is not necessary to have enough replicas to continue all services of such a system in the presence of faults. It is rather sufficient to continue the most critical services while some other services can be abandoned to reclaim resources. For example, it will be acceptable to abandon comfort functionally or to reduce the control quality in such cases. Cost of replica determinism enforcement and fault-tolerance management is therefore the major critical issue for the application of systematic fault-tolerance in the area of automotive electronics.

The problem of replica determinism in fault-tolerant real-time systems will be treated throughout the remainder of this book in a general context but with special relations to the requirements for automotive electronics. It is the authors firm belief that systematic fault-tolerance is the future direction to achieve the required level of dependability for the next generation of systems in the area of automotive electronics.

# Chapter 3

# System model and terminology

This chapter introduces the system model and related terminology that is used throughout this book. Partially, the terminology as defined in [Cri91b, Lap92] was chosen since it covers hardware as well as software aspects. Generally speaking, each system is defined by its system structure and by the behavior of its constituting components. The two aspects of structure and behavior have to be covered by a system model to allow appropriate abstraction from the complexity of the "real" system.

In the following the structure of a real-time system is described together with definitions of hard and soft real-time systems. Furthermore, the concept of dependability is presented along with a classification of failure modes, failure semantics and the assumption coverage. Important system parameters such as synchrony, partial synchrony and asynchrony of processors and communication are defined. The possibility of failure masking and resiliency is also described in this chapter.

## 3.1 System structure of a real-time system

The basic unit that constitutes a system is called a server. A server implements a specific service while hiding operational details from the users, who only need to know about the service specification. A service specifies a set of (deterministic) operations which are triggered by inputs or the passage of time and result in outputs to the service user and/or in service state changes.[7] One server may execute one or more service requests at a time, i.e. services of higher precedence may interrupt services of lower precedence. Implementation of servers may be by means of hardware, software or both. For example a low level graphic display service may be implemented in hardware as a special graphic processor. Alternatively the service could be implemented by a software server. This software server in turn is executing on a hardware

[7]Note that this definition does not agree with the definition of state machines as given by Schneider [Sch90] since outputs of state machines are defined to be independent of time. If, however, state machines are associated with clocks that may request services, the two definitions are identical.

server which is implemented by a general purpose processor. Based on this server/service concept a real-time systems is defined as follows:

- ***Real-time system:*** A real-time system is a system which delivers at least one *real-time service*. A real-time service is a service that is required to be delivered within a time interval dictated by the environment.

Real-time systems are subdivided in *hard* and *soft* real-time systems. This classification is based on the *criticality* of a systems service, which may be expressed through a *time-utility* function [Bur91]. The time-utility function specifies the relation between a service's contribution to the overall goal of the system and the time at which results are produced.

- ***Hard real-time system:*** A real-time system is characterized as a *hard real-time* system if the time-utility function for late service delivery indicates *severe* consequences to the environment [SR88], or is classified as a *catastrophic* failure [Bur91].

- ***Soft real-time system:*** A real-time system is characterized as a *soft real-time* system if the time-utility function for late service delivery indicates a low or zero utility, but no severe consequences or catastrophic failures upon late service delivery.

The distinction between hard and soft deadlines is useful for general discussion on real-time systems. Actual applications may, however, produce hybrid behaviors such that hard and soft real-time requirements are mixed.

After this classification of real-time systems we now turn to the structural aspects. At the top level of abstraction the whole real-time systems is viewed as a server. The real-time systems service specification is identical with the service specification of this top level server. Inputs to this server are made by means of sensors or interface components from other systems, service outputs are delivered by actuators or interface components. If the real-time system is used to monitor a physical process, then the process delivers service request to the top level server. If the physical process is not only monitored but controlled by the real-time system, then it delivers service requests on the one hand and is also a user of the services provided by the top level server on the other hand. Additionally there may be an operator who also sends service requests and uses the provided services. The operator interface is implemented by devices such as a console (for process control) or a brake pedal (for an anti-blocking system in a car) which allows human interaction. Figure 3-1 shows this relation between the top level server and its service users.

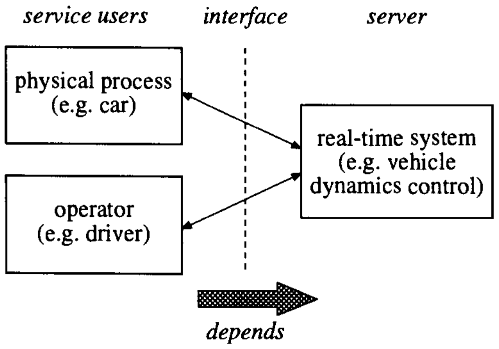
service users
interface
server
physical process (e.g. car)
operator (e.g. driver)
real-time system (e.g. vehicle dynamics control)
depends

*Figure 3-1: Model of a real-time system*

Besides the relation of a server to its environment and servers among each other another structural property is *distributedness*. A system that uses replication to achieve fault-tolerance is by definition a distributed system. This leads to the question as to what the properties are that distinguish a distributed computer system from a central one.[8] The definitions of a distributed computer system are not exact. Lamport, for example, defines a distributed system as follows: "A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process." [Lam78b] An additional important fact is the possibility of independent failures (which is the basic motivation for replicated systems) [Sch93b]. It may happen that messages, which are used for exchange of information, are being lost or that replicated servers may fail. In an asynchronous system, furthermore, infinitely large message delays and processing delays have to be considered. In the context of fault-tolerant real-time systems, distributedness is defined according to the following constituting properties:

- ***Independent failures:*** Because there are different servers involved, some might fail while others function correctly. It is furthermore required that the system has to continue operation, despite failures of individual servers (which are covered by the fault hypothesis).
- ***Non-negligible message transmission delays:*** The interconnection between servers provides lower bandwidth and higher latency communication than that available within a single server [Gra88].
- ***Unreliable communication:*** The interconnections between the individual servers are unreliable. This means that the connections between servers are unreliable compared to the connections within a server.

[8]This question seems to be obvious, but, for example, in the case of a computer with dual processors it is not all clear whether such a system should be considered central or distributed.

## 3.2 The relation depend and dependability

The relation "depend", indicated by the gray arrow in Figure 3-1, shows the depend relation between the service user and the server. Service users depend on the correct function of the used servers. That is, a service user may potentially fail if it *depends* on a lower level server that has failed. Note that a service must not necessarily fail if it uses a server which fails, rather a service may use a set of replicated servers to mask failures of individual servers. Hence the service does not necessarily depend on a single server, but can depend on a replicated set of servers.

Dependability is defined as the trustworthiness of a server such that reliance can justifiably be placed on the service it delivers [Car82]. The attributes of dependability are defined as follows [Lap92]:

- with respect to the readiness for usage, dependable means **available**,
- with respect to the continuity of service, dependable means **reliable**,
- with respect to the avoidance of catastrophic consequences on the environment, dependable means **safe**,
- with respect to the prevention of unauthorized access and/or handling of information, dependable means **secure**.

The most relevant attributes of dependability for real-time systems are reliability, safety and availability (in many real-time systems security is not an issue). The impairments to dependability are failures. They are circumstances under which a service no longer complies with its service specification. The service specification has to prescribe correct behavior in the value and time domain. Failures result from the undependability of servers. A server is said to be *correct* if, in response to inputs and the passage of time, it behaves consistently with the service specification. If a server does not behave in the manner specified, it *fails*.

This system model defines the server/service relation and the depend relation in a recursive manner. A server provides a service by using lower level services, which in turn is provided by servers. Hence the higher level server depends on the lower level service. This chain of server/service relations starts with the top level server at the highest abstraction level and continues down to the atomic service unit. The atomic service unit has to be chosen in accordance with the lowest (or most detailed) abstraction level that is considered. Corresponding to the chain of server/service relations, failures of the top level server can be tracked along the depend relation. The origin of a failure is located if the lowest level servers are found that are either atomic service units or ones that depend only on correct lower level servers.

## 3.3 Failure modes, -semantics and assumption coverage

Servers can fail in different ways which are categorized as *failure modes*. Failure modes are defined through the effects, as perceived by the service user. In the literature of fault-tolerant systems a multitude of failure modes has been defined. However, some of them have found broader acceptance and will be used throughout this book:

- ***Byzantine or arbitrary failures*** [LSP82]: This failure mode is characterized by a non-assumption: there is no restriction on the effects a service user may perceive. Hence this failure mode is called *malicious* or *fail-uncontrolled*. This failure mode includes "two-faced" behavior, i.e. a server can send a message "fact $\varphi$ is true" to one server and a message "fact $\varphi$ is false" to another server. Additionally a server may forge messages of other servers.
- ***Authentification detectable byzantine failures*** [DS83]: In this case servers may show byzantine behavior, but they are not able to forge messages of other servers (messages are authenticated). Or in other words, a server cannot lie about facts which are sent by other servers.
- ***Performance failures*** [CASD85, Pow92]: Under this failure mode servers have to deliver correct results in the value domain. In the time domain, however, results may be delivered **early** or **late**.
- ***Omission failures*** [PT86, Pow92]: Omission failures are a special case of performance failures. If service requests are only subject to infinitely late service responses, this failure mode is called omission failure.
- ***Crash failures*** [LF82, Pow92]: If a server suffers only from omission failures and also does not respond to any subsequent service requests, the server is said to have *crashed*.
- ***Fail-stop failures*** [SS83, Sch84]: A fail-stop server can only exhibit crash failures, but it is additionally assumed that any correct server can detect whether any other server has failed. Also the assumption is made that every server employs a stable storage which reflects the last correct service state of the crashed server. The stable storage can be read by other servers, even if the owner of the stable storage has crashed.

The above listed failure modes build up a hierarchy where byzantine failures are based on the weakest assumptions (a non-assumption) on the behavior of servers and fail-stop failures are based on the strongest assumptions. Hence byzantine behavior is the most *severe* and fail-stop behavior is the least severe failure mode. The byzantine failure mode covers all failures classified as authentification byzantine, which in turn covers all performance failures, and so on. More formally: Byzantine failures $\supset$

authentification detectable byzantine failures $\supset$ performance failures $\supset$ omission failures $\supset$ crash failures $\supset$ fail-stop failures. Additionally, the failure modes can be characterized according to the viewpoints domain and perception by the service users. The failure domain viewpoint leads one to distinguish:

- ***Value failures:*** The value of the service response does not agree with the service specification. This class includes byzantine and authentification detectable byzantine failure modes.
- ***Timing failures:*** The timing of the service response does not agree with the service specification. This class covers performance, omission, crash and fail-stop failures.

When a server has several service users, the viewpoint of failure perception leads one to distinguish:

- ***Consistent failures:*** All service users have the same perception of the failure. This class includes performance failures, omission failures, crash failures, and fail-stop failures.
- ***Inconsistent failures:*** Different service users obtain different perceptions of the failure. This class includes byzantine and authentification detectable byzantine failures.

Furthermore failures may be characterized by their consequences for the environment. Under the viewpoint of failure severity one can distinguish the following consequences:

- ***Benign failures:*** The consequences of a service failure are of the same order of magnitude as the benefit provided by a correct service delivery.
- ***Catastrophic failures:*** The consequences of a service failure are vastly more severe than the benefit provided by a correct service delivery.

The above given classifications of failure modes is not restricted to individual instances of failures, but can be used to classify the failure behavior of servers, which is called a server's *failure semantic.*

***Failure semantic:*** A server exhibits a given failure semantic if the probability of failure modes which are not covered by the failure semantic is sufficiently low.

If for example a server is said to have crash failure semantics, then all individual failures of the server should be crash or fail-stop failures, the possibility of more severe failures, e.g. omission failures, should be sufficiently low. The failure semantic is a stochastic specification of the failure modes a server may exhibit, which has to be chosen in accordance with the application requirements. In other words: The failure semantic defines the most severe failure mode a service user may have to consider. Fault-tolerant systems are designed with the assumption that any server that fails will do so according to a given failure semantics. If, however, a server should fail in a way that the failure semantics is violated, then it may happen that the system fails as a whole. This consideration leads to the important concept of *assumption coverage* [Pow92] which is intimately related to the definition of failure semantics.

***Assumption coverage:*** Assumption coverage is defined as the probability that the possible failure modes defined by the failure semantics of a server proves to be true in practice conditions on the fact that the server has failed.

For servers defined to have byzantine failure semantics assumption coverage is always total (or 1) because byzantine failures are the most severe failure mode. In all other cases the assumption on the failure semantics may be violated because the server may show byzantine behavior. Assumption coverage is less than 1 in these cases. The assumption coverage is a very critical parameter for the design of fault-tolerant systems. If the assumptions are relaxed too much to achieve a good assumption coverage, the system design becomes too costly and overly complicated, since severe failure semantics (e.g. byzantine failures) have to be considered. On the other hand, if too many assumptions are made, the system design is easier, but the assumption coverage may be unacceptably low. Hence, an application specific compromise between complexity and assumption coverage has to be made.

## 3.4 Synchronous and asynchronous systems

Another important criterion which reflects the system behavior is the synchrony of servers and communication. Servers may either be *synchronous* or *asynchronous*. A server is said to be synchronous if there exists an a priori known upper bound $\Delta$, such that every server takes at least one processing step within $\Delta$ real-time steps.[9] A server is said to be asynchronous if no a priori known upper bound $\Delta$ exists. The *communication delay* can either be bounded or unbounded. If every service request sent between servers arrives at its destination within $\Phi$ real-time steps, for some a priori known $\Phi$, the communication delay is bounded. Hence, this type of communication is called synchronous. For asynchronous communication there exists no known bounded delay $\Phi$. Another distinction between synchronous and asynchronous systems is the existence of local clocks with a bounded drift rate. Synchronous systems may have local clocks with a bounded drift rate such that every server $S_i$ has a local clock $C_i$ with a known bounded drift rate $\rho \geq 0$. A clock is said

[9]Since this definition is aimed at real-time requirements it is somewhat more restricted than the general definition which only requires that each server takes at least 1 processing step if any server takes $s$ steps.

to be synchronous if for all clock readings $C_i(t)$ at real-time instants $t$ and for all $i$ with $t > t'$ the following condition holds:

$$1-\rho \le \frac{C_i(t)-C_i(t')}{t-t'} \le 1+\rho \tag{3.1}$$

It is therefore possible to use the timeout mechanism in synchronous systems for failure detection. The local clocks in real-time systems are often not only bounded by a given drift rate, but are *approximately synchronized.* In this case the following property is additionally satisfied: there is an a priori known constant $\varepsilon$, called clock synchronization precision, such that the following condition holds for all $t$ and for all pairs of servers $S_i$ and $S_j$:

$$\left|C_i(t)-C_j(t)\right| \le \varepsilon \tag{3.2}$$

This condition guarantees that any two clock readings, made by different servers at the same point in time, differ for at most $\varepsilon$ time units. It is possible to implement approximately synchronized clocks even in the presence of failures [LM85, KO87]. Based on the behavior of servers and communication the synchrony of a system is defined as follows:

- ***Synchronous system:*** A system is called synchronous if both the individual servers and the communication are synchronous. Servers may have local clocks with a bounded drift rate.
- ***Asynchronous system:*** A system is called asynchronous if either one of the servers and/or the communication between servers is asynchronous. There are no local clocks with a bounded drift rate.

The attraction of the asynchronous system model arises from the fact that algorithms are designed without relying on bounds for processing and communication speed. In practice, asynchrony is introduced by unpredictable loads and computation times. The problem with asynchronous systems is the impossibility to distinguish between a message or a service response which was delayed or that is entirely missing. Consequently, the case of timing failures is only pertinent to the synchronous system model.

Real-time systems, however, are per definition synchronous: services have to be delivered within a fixed time interval. Hence, the assumption of asynchrony may only be used for the algorithmic design but not as a description of the service behavior. A further problem is the well known fact that some problems of fault-tolerant distributed computing, which can be solved in synchronous systems, cannot be solved for asynchronous systems [FLP85, DDS87, Lyn89]. To circumvent these impossibility results and to take advantage of the fact that timeouts are commonly used for failure detection (even in practical implementations of "asynchronous systems"), several modes of *partial synchrony* have been defined for servers and communication:

- ***Partial synchrony:*** A server (the communication) is called partial synchronous if one of the following conditions hold: (1) There exits a bound on the processor speed $\Delta$ (communication delay $\Phi$), but the bound is unknown [DLS88]. (2) An a priori known bound on the processor speed $\Delta$ (communication delay $\Phi$) exists, but this bound holds only after a unknown time [DLS88]. (3) The bound on the processor speed $\Delta$ (communication delay $\Phi$) is unknown and the bound holds after an unknown time [CT91].

Combining these definitions of partial synchronous servers and partial synchronous communication, several different definitions for **partial synchronous systems** can be given. Systems based on these assumption are not limited by the impossibility results for totally asynchronous systems [DLS88, CT91]. Hence, the concept of partial asynchrony lies between the cases of synchronous systems and asynchronous systems. The definition of partial asynchrony resembles real requirements much closer than the assumption of an asynchronous system, since infinite delays are unacceptable, at least for us humans.

## 3.5 Groups, failure masking, and resiliency

To achieve fault-tolerance, servers are replicated. Such replicated sets of servers are commonly called **groups** [Cri91b, Pow91b, VR92]. At the software level there is no agreed upon terminology for the basic unit of replication. Often the notation *object* is used to describe software components that are replicated; examples are the ISIS system at Cornell [BJRA84], the Psync/x-Kernel [MPS89], the Amoeba System [Tan90] or in [HS92]. The Delta-4 project uses the term *software component* [Pow91a]. Other entities for replication are *processes* [TS90, AMM$^+$93], *transactions* [SB89], *remote procedures* [Coo84] and *state machines* [Lam78a, Sch90]. In any case the replicated entity consists of program[10] and state. For this book the term server or replica will be used for the basic unit of replication, regardless whether the implementation is done by means of software or hardware.

A group as a whole may be considered as a single server providing a service one abstraction level higher. Ideally, the group should behave like a single dependable server. In practice, however, there are fundamental limitations to groups, which do not exist for single servers (see section "Fundamental limitations to replication"). In

[10]In this context the term program does not necessarily imply a software implementation. A program rather denotes the logic which is used to implement a service. The implementation can be hardware- or software based.

case of server failures the group behavior can be characterized by the kind of *failure masking* technique that is employed. Two distinct possibilities exists to achieve fault-tolerance by using groups:

- ***Hierarchical failure masking:*** The servers within a group come up with diverging results and the faults are resolved by the service user one hierarchical level higher.

- ***Group failure masking:*** The group output is a function of the individual group members output (e.g. a majority vote, a consensus decision). Thus failures of group members are hidden from the service user.

The difference between hierarchical and group failure masking lies in the location where failure masking occurs, at the level of the service user or at the server (service provider) level. As a consequence, hierarchical failure masking is not transparent to the service user. It is typical done by *exception handling*, which is a convenient way to propagate and handle failures across hierarchical levels [Cri89].

The number of replicas within a group is called **replication level**. Fault-tolerance of a server group is typically measured in terms of the maximum number of servers that can fail. If up to $n$ failures are tolerated while providing a correct service, then the group is said to be **$n$-resilient**. Reliability and safety requirements on the other hand are most often described by random variables since failures are of stochastic nature. These stochastic requirements, however, may be transformed to a $n$-resiliency requirement by calculating the probability of $n$ server failures within a given period.

# Chapter 4

# Replica determinism and non-determinism

This chapter is aimed at a fundamental discussion of replica determinism and non-determinism without resort to actual implementations and methodologies that enforce replica determinism. The first section introduces a definition of replica determinism. It will be shown that a generic definition of replica determinism can only be a framework. Any detailed or formal definition is strictly application specific. A refinement of the generic definition of replica determinism to certain application scenarios will be given. The second section lists possible modes for non-deterministic behavior of replicated servers. This list shows that replica non-determinism is introduced by standard functions such as on-line scheduling, timeouts or readings of replicated sensors. In the third section the fundamental limitations of replica determinism are discussed. It will be shown that it is impossible to achieve *total* replica determinism. By defining a cause consequence relation on the possible effects of replica non-determinism a notion of the atomic source for replica non-determinism is introduced. The important problem of how to characterize all possible atomic sources for replica non-determinism will be discussed. It will be shown that replica determinism, knowledge and simultaneity are closely related. The fourth section will consider the question whether there are replicated systems that do not require the enforcement of replica determinism. Two possible cases for such systems will be discussed. It will be shown, however, that any "real" non-trivial replicated system has to enforce replica determinism.

## 4.1 Definition of replica determinism

To discuss the problem of replica determinism we first need to define the general concept of determinism. In the field of philosophy the antagonistic pair of determinism and non-determinism (freedom) has been extensively discussed at least since the days of early Greek philosophy. There have been many versions of deterministic theories in different contexts, such as ethical determinism, logical determinism, theological determinism or historical determinism. The following definition of determinism is taken from the encyclopedia of philosophy [Edw67]:

> *"Determinism is the general philosophical thesis which states that for everything that ever happens there are conditions such that, given them, nothing else could*

*happen. (. . .) an event might be said to be determined in this sense if there is some other event or condition or group of them, sometimes called its cause, that is a sufficient condition for its occurrence, the sufficiency residing in the effect's following the cause in accordance with one or more laws of nature."*

For the purpose of this work we are concerned with physical determinism since computer systems are representatives of the physical world. Based on the findings of quantum theory it is known that the behavior of matter is in essence non-deterministic and based on stochastic processes. At a higher level, however, it is justifiable to assume deterministic behavior. It is for example valid to assume that an electronic flip-flop behaves deterministically (in the absence of failures). Digital computers are at first sight par excellence examples of physical entities showing such deterministic behavior. Given the same sequence of inputs and the same initial state, a computer should always produce identical outputs if failures are neglected. In analogy deterministic servers are defined as follows:

***Deterministic server:*** A server is said to be deterministic if, in the absence of server failures, given the same initial state and an identical sequence of service requests, the sequence of server outputs is always identical.

It is important to note that for real-time systems the connotation of "identical" has to be considered in the domain of value and time. Given a group of replicated servers, the problem of replica determinism is to assure that all correct servers in the group behave identically. Note that replica determinism is a group-wide property. If some of the servers are failing, the remaining servers in the group should still behave identically and correctly. This definition is based on the assumption that servers are affected by failures independently.

The problem with a general definition of replica determinism is to cover a sufficiently broad range of replication strategies. Most often the definition is restricted to a class of replication strategies where each server within a group is active and delivers outputs. Also the aspect of time is neglected in many definitions of replica determinism. For example, the Delta-4 project uses the following informal definition of replica determinism [Pow91b]:

*"A replica group is deterministic if, in the absence of faults, given the same initial state for each replica and the same set of input messages, each replica in the group produces the same ordered set of output messages."*

This definition would rule out any replication strategy where some servers deliver no outputs but keep their state up-to-date. The MARS system, for example, may have *shadow components* that deliver outputs only if some other server in the group has failed [KDK+89]. A strictly formal definition of replica determinism, which is based on the CSP [Hoa85] trace model, is given in [MP88, KMP91, KMP94]. This formal definition, as well as the definition of Delta-4 project for replica determinism are not addressing real-time aspects. Both definitions state only that each replica in the group should produce the same ordered sequence of output messages. Real-time systems, however, depend not only on consistent information but on consistent *and* timely information. We therefore introduce the following definition of replica determinism to cover a broader range of replication strategies while incorporating real-time aspects.

***Replica determinism:*** Correct servers show *correspondence* of server outputs and/or service state changes under the assumption that all servers within a group start in the same initial state, executing *corresponding* service requests within a *given time interval.*[11]

The flexibility of this definition is traded for being somewhat generic. This definition of replica determinism is intentionally given on an informal basis to cover a broad spectrum of replication strategies since a formal definition would be too restrictive. On the other hand, a formal definition for replica determinism has the advantage that a specific replication strategy is made accessible to formal methods.

This definition of replica determinism is not restricted to actively replicated systems where replicated services are carried out in parallel by replicated servers. In addition, passive or standby replication is covered because such systems have *corresponding* service states. According to the definition of replica determinism replicated servers must either have common concurrent service outputs, or they must have a shared service state. For example, the checkpoint information of a standby server has to correspond with the actions recently carried out by the active server and with its actual service state. If neither of the two conditions is true, the system may also be considered as replicated, but then replica determinism is a non-problem since there is no service state and output which has to correspond. An example of such a replicated system are two personal computers which are used by an author to write a paper. The source text is on a floppy disk. If one computer fails, the floppy may be easily transferred to the other computer. The computer, the operating system and the word processor can be of a different type as long as the format of the source text is compatible, but the two computers have no corresponding outputs and service state. For the remainder of this book only replicated systems which share service outputs, service state or both are considered.

To cover specific replication strategies more precisely with the above given definition of replica determinism, the wording "*correspondence* of server outputs and/or service state changes", "*corresponding* service requests" and "within a *given time interval*" has to be defined more precisely. This definition does not require replica outputs to be identical, they rather have to fulfill a correspondency requirement in the

[11]The terms "correspondence" and "within a given time interval" will be discussed later on in this section.

value and time domain which is application specific. For a MARS shadow component this correspondency requirement may be translated to:

- all service requests have to be issued identically within a given time interval
- all service state changes have to be identical within a given time interval
- active components have to produce identical outputs within a given time interval
- a shadow component has to switch to active mode within a given time interval if one active component in the group fails

Another example may be a system which uses floating point calculations where results are not exactly identical. For such a system the correspondence requirement has to be relaxed from identity to a given maximum deviation between any two outputs. Assume there is a metric to measure the deviation of server states and outputs, then the correspondence requirement may be translated as follows:

- all service requests have to be issued identically within a given time interval
- within a given time interval the maximum deviation between service states, as defined through the metric, has to be bounded
- within a given time interval the maximum deviation between outputs, as defined through the metric, has to be bounded

While it is possible in practice that replicas produce identical results in the value domain, it is impossible in "real" systems to guarantee that all results are delivered at exact the same point in time. Therefore the identity requirement has to be relaxed to an application specific correspondence requirement which covers temporal uncertainty among server outputs and/or state changes.

Real-time aspects have to be considered for the output domain of servers as well as for the input domain. Services of real-time systems generate outputs on explicit requests but also on implicit requests which are activated by the passage of time [Lam84]; i.e. real-time services are time dependent. To obtain corresponding server outputs it is therefore necessary that service requests are issued "within a *given time interval*". This wording, as given by the definition, has to be defined in more detail to cover specific application requirements. A process-control problem for example may require that service requests are made within a time interval of say 1 millisecond, for systems with short latency time requirements such as automotive electronics there are even requirements in the range of microseconds, for human interaction 0.3 seconds may be appropriate, for time independent functions an infinite time interval is sufficient.

The above examples have shown that different applications require different definitions of correspondency. Hence it follows that any exact or formal definition of replica determinism is application specific. A general definition of replica determinism—as the one given above—can only define a framework which has to be specified in more detail to cover a specific application.

## 4.2 Non-deterministic behavior

Assuming that the individual servers in a group are deterministic, and given an identical sequence of input request, ideally a server group should satisfy replica determinism in a strict sense. That is, all server outputs and/or service state changes should be identical at the same point in time. Unfortunately this is not achievable for replicated computer systems, as experience has shown.[12] It is neither possible to attain exactly identical service inputs nor is it possible to attain identical server responses. A direct consequence of this observation is that computer systems cannot be considered as deterministic servers. At first glance this seems to be a contradiction of the intuition that computer systems behave deterministically in the absence of failures. A closer investigation shows, however, that computer systems only behave *almost* deterministically. Under practice conditions there are minor differences in the behavior. This immediately leads to the question: what are the sources for these minor differences that lead to replica non-determinism? A non exhaustive list of possible modes for non-deterministic behavior is given below:

### 4.2.1 Inconsistent inputs

If inconsistent input values are presented to servers then the outputs may be different too. This happens in practice typically with sensor readings. If, for example, each server in a group reads the environment temperature by an analog sensor, it may happen that, due to the inaccuracy and the digitizing errors, slightly different values will be read. The same effect may also occur if only one sensor is used but different servers read the sensor at different points in time.

This problem of inconsistent sensor readings is not restricted to the value domain. It may also occur in the time domain. If, for example, the point in time of an event occurrence has to be captured, different servers may observe different times due to minor speed differences of the reference clock. It is important to note that slightly different inputs can lead to completely different outputs. Consider for example two servers which are used to issue an alarm if a critical process temperature of 100 °C is exceeded. If one server reads a process temperature of 100 °C and the other server reads 100.01 °C then the resulting outputs will be no alarm and alarm, respectively.

[12]This basic impossibility to attain replica determinism in a strict sense is discussed in the next section from a theoretical point of view.

### 4.2.2 Inconsistent order

Similarly, replica determinism is lost if service requests are presented to servers in different order, under the assumption that service requests are not commutative. The assumption of non-commutativity of service request holds for most systems. A typical example is a message queue where higher precedence messages may overtake lower precedence ones. Due to slight differences in the processing speed it may happen that one server has already dequeued the lower precedence message and started to act upon the message, while in the message queue of the other server the higher precedence message has overtaken the lower precedence one. One server acts first on the lower precedence message while the other acts first on the higher precedence one. As a consequence inconsistent order arises, unless there is global coordination.

Generally speaking, the problem of inconsistent order may occur in all cases where different time lattices are related to each other. For example this problem is relevant in all systems where external events occur in the domain of real-time, and these events are related to internal states of servers. This principle is illustrated in Figure 4-1.

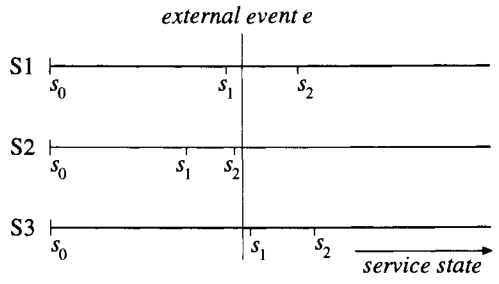

*Figure 4-1: Inconsistent order*

Figure 4-1 shows the service state progression of the three servers S1, S2, and S3. The initial service state of all servers is $s_0$, followed in sequential order by the states $s_1$ and $s_2$. Since the processing speed of the individual servers varies, the order of service states and external events is inconsistent. Server S1 sees the order $s_0 \rightarrow s_1 \rightarrow e \rightarrow s_2$, server S2 sees $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow e$, and server S3 sees the order $s_0 \rightarrow e \rightarrow s_1 \rightarrow s_2$, where $\rightarrow$ denotes the happened before relation as defined by Lamport [Lam78b]. Sources for the varying processing speed are differences in the clock speeds or delays which are caused by other service requests. If, for example, server 3 runs on top of an operating system, it may happen that the processing of server 3 is delayed due to processing requests from other servers. If the operating system on which server 2 runs has to manage a different set of servers, it may happen that at the same time while server 3 is delayed, server 2 may act immediately.

### 4.2.3 Inconsistent membership information

A system-wide and consistent view of the group membership of servers is important for replicated systems. But it has to be considered that group membership may change in time. If a server fails or leaves voluntarily, then it is removed form the membership. If a server is recovered or newly added to the system, then it joins some group(s) as a member. In case of inconsistencies in the group membership view it may happen that replica determinism is lost.

Figure 4-2 gives an example of replica non-determinism which is caused by inconsistent membership information. Consider the two servers S1 and S2. Server S1 assumes that servers S10, S11 and S12 are members of group *G* while server S2 assumes that only S10 and S11 are members of group *G*. Thus server S1 will send service requests to S10, S11 and S12, but server S2 will only send requests to S10 and S11. Group *G* becomes inconsistent because servers S10 and S11 are acting on different requests than server S12 does.

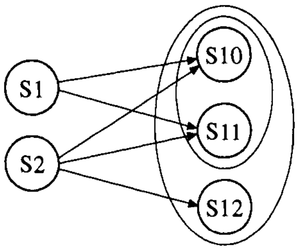

*Figure 4-2: Inconsistent membership information*

### 4.2.4 Non-deterministic program constructs

Besides intended non-determinism, like (hardware) random number generators, high level languages such as Ada, OCCAM or Fault-Tolerant Concurrent C (FTCC) have non-deterministic statements for communication or synchronization [TS90].

```
task server is
    entry service₁();
    ...
    entry serviceₙ();
end server;
```

```
task body server is
begin
    select
        accept service₁() do
            action₁();
        end;
        ...
    or
        accept serviceₙ() do
            actionₙ();
        end;
    end select;
end server:
```

*Figure 4-3: Non-determinism in Ada*

Figure 4-3 illustrates non-determinism in Ada programs. The task named server advertises its services named $\text{service}_1$ through $\text{service}_n$. If more than one service call is pending when the **select** statement is executed, selection between the **accept** statements is arbitrary. In a replicated system this may result in different execution orders among replicas.

### 4.2.5 Local information

If decisions within a server are based on local knowledge, that is information which is not readily available to other servers, a server group loses its determinism. Uncoordinated access to a local clock is a typical example. Since it is impossible to achieve ideal synchronization on a set of distributed clocks [LM85, KO87, DHS86], clocks have to be considered as local information. Another example of local information is the processor or system load as supplied by an operating system. This information depends on the behavior of all servers that are managed by the operating system. Assume the operating system runs on two processors managing different sets of servers. If one server is replicated on both processors, each operating system will return different information on the processor load.

### 4.2.6 Timeouts

A similar phenomenon can be observed if timeouts are used without global coordination. If the timeout is based on a local clock, the above entry "local information" applies. But even if an ideal global time service is assumed, it might happen that some servers will decide locally to timeout and others will not. The reason for this divergence are the differences in the processing speed of individual servers. Figure 4-1 may be used to illustrate the effect of uncoordinated timeouts. The external event $e$ is interpreted as the timeout, and it is assumed that at the service state $s_1$ each server decides to generate an output, dependent on whether the timeout has elapsed or not. Servers S1 and S2 will decide to generate the output, while server S3 decides not to generate an output.

### 4.2.7 Dynamic scheduling decisions

The scheduling problem is to decide on a sequence, how to grant a set of service requesters access to one or more resources (processors). If this decision is made on-line, scheduling is called *dynamic*. Diverging scheduling decisions on a set of replicated servers may lead to replica non determinism. This is caused by diverging execution orders on different servers. Consider two service requests, one is "add 1 to a variable $x$", where $x$ represents the servers internal state, the other service request is "multiply the variable $x$ by 2". It is obvious that a different execution order of these two service requests leads to inconsistent results since $(x + 1)\,2 \neq 2x + 1$. There are two principle reasons for diverging scheduling decisions. One reason is that scheduling decisions are based on local information. This is typically the case if non-identical, but overlapping sets of servers are scheduled on two or more processors. Each scheduler has different information on services to schedule, but to guarantee replica determinism, the joint set of servers has to be scheduled in identical order. The second reason for inconsistent scheduling decisions are minimal processing speed differences of servers—even if the decisions are based on global information [Kop91]. At the hardware level for example, an asynchronous DMA-transfer may be scheduled immediately on one server but is delayed on another server since its last instruction was not completed. Software servers may interrupt their execution to take a scheduling decision, but the actual server state may differ. For example the faster server has finished a service request while the slower server needs a few more instructions to finish.

### 4.2.8 Message transmission delays

The message transmission delay depends on various parameters. Among them are media access delay, transmission speed, the distance between sender and receiver or delays caused by relay stations which are used for message forwarding. In a replicated system this may lead to the case where replicas obtain service requests at different times or that service outputs are delivered at different times. These variabilities in the message delivery time can cause replica non-determinism. Consider two servers which are used as watchdogs to monitor a critical service. If they timeout before a new service request arrives to retrigger the watchdog, an emergency exception is signaled. Due to variable message transmission delays the request to retrigger the watchdog may arrive at one server before the timeout has elapsed, while the request arrives late at the other server. This server will raise the emergency exception. The other server will detect no emergency condition, hence replica determinism is lost. Variable communication delays can also lead to replica non-determinism in cases where no timeouts or external events are involved. Consider the example illustrated in Figure 4-4:

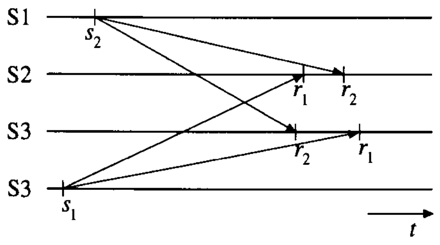

*Figure 4-4: Variable message transmission delays*

At time $s_1$ server S1 sends a service request (message) to the servers S2 and S3. At time $s_2$ server S4 also sends a service request (message) to S2 and S3. Due to the varying message transmission delays the receive order of messages is reversed for server S2 and S3. In this case the variable message transmission delay leads to inconsistent order of service requests.

## 4.2.9 Consistent comparison problem

Since computers can only represent finite sets of numbers their accuracy in mathematical calculations is limited[13] in general. Each number represents one equivalence class. It is therefore impossible to represent a dense set of numbers, as for example the real numbers. If the result of an arithmetic calculation lies very close to the border between two equivalence classes, different service implementations may come up with diverging results even though the inputs were the same. If this result is now compared against some fixed threshold in parallel, different servers will take different decisions. Consider the calculation of $(a-b)^2$, where $a = 100$ and $b = 0.005$. From a mathematical point of view $(a-b)^2 = a^2 - 2ab + b^2$ holds. But when, for example, two servers carry out the calculations in a floating point arithmetic with a mantissa of 4 decimal digits and rounding, the results will differ.

| *math. expr.* | *exact results* | *4 digit floating point results* |
|---|---|---|
| $a-b$ | 99.995 | $1.000 \times 10^2 - 5.000 \times 10^{-3} = 1.000 \times 10^2$ |
| $(a-b)^2$ | 9 999.000025 | $1.000 \times 10^2 \; 1.000 \times 10^2 = 1.000 \times 10^4$ |

*Table 4-1: 4-digit result of $(a-b)^2$*

| *math. expr.* | *exact results* | *4 digit floating point results* |
|---|---|---|
| $a^2$ | 10 000 | $1.000 \times 10^2 \; 1.000 \times 10^2 = 1.000 \times 10^4$ |
| $-2ab$ | -1 | $-2.000 \times 10^0 \; 1.000 \times 10^2 \; 5.000 \times 10^{-3} = -1.000 \times 10^0$ |
| $b^2$ | 0.000025 | $5.000 \times 10^{-3} \; 5.000 \times 10^{-3} = 2.500 \times 10^{-5}$ |
| $a^2 - 2ab + b^2$ | 9 999.000025 | $1.000 \times 10^4 - 1.000 \times 10^0 + 2.500 \times 10^{-5} = 9.999 \times 10^3$ |

*Table 4-2: 4-digit result of $a^2 - 2ab + b^2$*

Due to the limited number of digits one server will calculate the result $1.000 \times 10^4$ (see Table 4-1) while the other server will calculate $9.999 \times 10^3$ (see Table 4-2). The same inconsistency may also arise if the same calculation steps are used but different servers use different number formats. If, for example, 9999 is a critical value above which a valve has to be shut, one server will keep the valve open while the other will give the command to shut. This kind of potential non-deterministic behavior—

[13]Even if an exact arithmetic exists for a certain application, it has to be considered that many algorithms are inexact, as for example the solution of non closed-form differential equations.

introduced by different service implementations—is referred to as consistent comparison problem [BKL89]. Note that different service implementations are introduced by $N$-version programming, by usage of diverse hardware or even by different compilers.

As the above listed modes for replica non-determinism indicates, replica determinism is not only destroyed by using esoteric or obviously non-deterministic functions, as true random number generators[14] are, but even by "vanilla" functions such as timeouts, reading of clocks and sensors, dynamic scheduling decisions and others. Another aspect with the different modes for replica non-determinism is that one mode may cause the other and vice versa. For example varying message delays may be the cause for inconsistent order. But it may just as well happen, that inconsistent order in a replicated message forwarding server leads to varying message delays. Hence, it is not possible at this level to say one mode of replica non-determinism is more basic than another. This problem will be treated in the next section.

## 4.3 Fundamental limitations of replication

This section defines a structure on the various modes of replica non-determinism by introducing a cause consequence relation. Along this structure an atomic cause for replica non-determinism can be identified. The main part of this section is then devoted to considerations about how to characterize *all* the atomic sources and a description of the atomic sources for replica non-determinism. It will be shown that it is impossible to avoid replica non-determinism in practical real-time systems.

It is important to note that replica determinism as well as replica non-determinism are group-wide properties. That is, either a group as whole behaves deterministically or not. Sources for replica non-determinism therefore have no single point of location but are usually distributed over a group. Hence, the term "source of replica (non-)determinism" should be understood by the connotation of "cause" rather then "(single) point of location" of replica non-determinism.

### Structuring the sources of replica non-determinism

In the previous section a non-exhaustive list of modes for non-deterministic behavior of computer systems was given. These various modes of replica non-determinism are not structured. That is, the consequences of a replica non-deterministic behavior may in turn be the source for another kind of non-deterministic behavior. For example the effects of inconsistent inputs may cause inconsistent order. But it is strictly application specific whether a certain source of replica non-determinism causes non-deter-

[14]Note that purely software implemented random number generators behave deterministically.

ministic behavior that is perceivable at the level of the service user. There is no general relation between the sources of replica non-determinism and non-deterministic behavior of servers. Thus the miscellaneous modes of replica non-determinism form a cause/consequence chain for a specific kind of non-deterministic server behavior that is observed at the server's interface, see Figure 4-5.

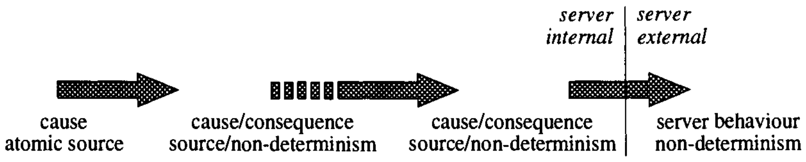

*Figure 4-5: Cause-consequence relation of non-determinism*

From a server groups point of view there is an atomic cause for a specific type of replica non-determinism. This atomic cause is the basic or *atomic* source for replica non-determinism at the level of service outputs. The relation between the non-determinism of server outputs and the atomic source can be traced along a server internal cause/consequence chain. From a service user's point of view only the non-deterministic behavior of service outputs may be perceived, not the atomic source for this behavior. The cause of this non-deterministic behavior may be located either in the group of servers that behaves non-deterministically, or in one or more servers at lower service level. The atomic source of replica non-determinism is located, if the lowest level servers are found that are either atomic or that depend only on lower level servers which behave deterministically.

## On defining all atomic sources of replica non-determinism: A first attempt

Considering the existence of an atomic source for replica non-determinism leads one to the question how to differentiate between atomic sources and derived sources of replica non determinism. Additionally, the even more important question is raised, whether it is in essence possible to describe *all* atomic sources for replica non-determinism. This question gains its importance from the practical aspect that complete knowledge of all possible sources for replica non-determinism is a prerequisite requirement to design replicated fault-tolerant systems. Otherwise the construction of replicated systems would be restricted to cycles of testing and ad hoc modifications until no more non-deterministic behavior may be observed during the test phase (which does not guarantee that the system will behave non-deterministically during operation, unless the test-coverage is 100%).

A first attempt to describe all the possible atomic sources of replica non-determinism can be made indirectly. By defining the requirements for replica determinism, any violation of these requirements is then considered as an atomic source of replica non-determinism. Such a definition for example is given in [Sch90], accordingly "replica coordination" requires the properties "agreement" and "order". Another example is the correspondency based definition of replica determinism which is introduced in section 3.1. While these definitions are appropriate to give an application independent framework to define replica determinism, they fall short in characterizing all the atomic sources of replica non-determinism. The reason for this shortcoming is that no exact or formal definition of replica determinism can be given which is completely application independent. The correspondency based definition for replica determinism is application specific per definition since the *correspondency* itself is an application specific property. The "agreement" and "order" based definition of replica determinism suffers from the same shortcoming. Neither agreement nor order can be defined as a total requirement. That is, these requirements are partial, they have to hold only for a certain level of abstraction or for certain service states and outputs.

Consider the following example which illustrates this problem: Two servers are used to control an emergency valve. According to the definition of replica determinism both servers have to deliver agreed upon service outputs in identical order, if no failure occurs. These servers, however, depend on lower level services. Hence, these lower level servers must also behave deterministically. In this example it is assumed that servers at some lower level are implemented by means of microprocessors. If the definition of replica determinism would require agreement and order as total properties then the microcomputers in our example have to fulfill these properties. The consequence of this definition would be that every output and every single transistor of the microcomputer must have the same state at the same time. It is obvious that this requirement would rule out nearly all practical implementations of fault-tolerant systems. It would be impossible to use the microcomputer for other, possibly non-replicated, services. Furthermore it would be impossible to use different kinds of microcomputers for each replica. It would be even impossible to use replicated clocks for the microprocessors. As a consequence, the requirements for replica determinism have to be defined differently for different levels of abstraction.

A reasonable definition of replica determinism for this example could be as follows: To require agreement and order from the top level server down to a certain software level, but excluding hardware. This definition allows an implementation of replicated servers by means of diverse microcomputers because there is no requirement for agreement and order at the hardware level. But with this definition it becomes practically impossible to decide which diverging behavior of microprocessors has to be considered as *correct* and which is an atomic source for non-deterministic behavior at higher service levels. It is obvious that such a decision criterion is strictly application specific, if it exists at all.

**The impossibility of total replica determinism**

This impossibility to define replica determinism totally is not only based on the possible non-deterministic behavior of lower level servers. It is impossible for practical reasons to define total replica determinism for any given service level, considering "real" systems. The following example will show this impossibility: Again the two servers are used to control an emergency valve. A total definition of replica determinism would then require that the service outputs of each server be delivered at exactly the same point in time. For this example it is assumed that the service output is implemented by a single digital signal. A high level indicates a closed valve and a low level indicates a open valve. It is impossible to guarantee that both signals have the same level at the exactly same point in time. Or, in other words, it is impossible to attain simultaneity. A timing diagram in Figure 4-6 illustrates this behavior .

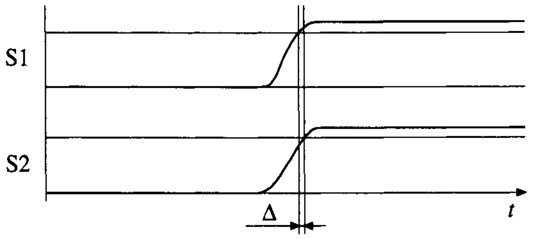

*Figure 4-6: Inaccuracy of real world quantities*

The servers S1 and S2 both decide to close the emergency valve. The output of each server changes its state from low to high, but not at the very same point in time. There is an inaccuracy of Δ between the two state changes. A possible source for this inaccuracy (or replica non-determinism) is a diverse implementation of servers with different processing speeds. But even with an identical implementation of services the processing speeds of servers may vary slightly due to the clock drifts of the individual oscillators. To avoid this source of inaccuracy, identical servers with one common clock source might be used. But even then it is impossible to guarantee a infinite accuracy, such that $\Delta = 0$. The reasons for the remaining inaccuracy may be based on the electrical tolerances of parts or by the varying signal transit time due to topological differences. It is possible to improve the accuracy, but it is impossible to achieve infinite accuracy.

It is therefore impossible to consent on time in a strict sense. That is, it is impossible to guarantee absolute simultaneity on the actions of replicated servers. It is only possible to consent on discrete time representations of real-time, but not on real-time itself. This impossibility is caused by the fact that any time observation through a clock is impaired by minor inaccuracies. Hence, the clock readings will show replica non-determinism. Consensus on time can only be achieved on discrete past clock values (because the time to achieve consensus is not zero). The processing speed of servers is also governed by a clock, which in turn governs the progress of the consensus protocol. Thus the finishing time of the consensus protocol is subject to replica non-determinism. This recursive dependency on time as a real world quantity makes it impossible to achieve simultaneity or consensus on real-time. It is therefore impossible to define replica determinism *totally* as an application independent property.

**A possible characterization of all sources of replica non-determinism**

An alternative would be a positivistic definition of all atomic sources for replica non-determinism. Such a definition should enumerate the atomic sources for non-determinism exhaustively. Since there is no exact and formal definition of replica determinism, it is impossible for theoretical reasons to show that any list of atomic sources for replica non-determinism is complete. This impossibility is based on the fact that there is no possibility to define an induction rule or to give a contradiction without a formal definition. Because of the impossibility to enumerate all atomic sources, an alternative approach may be taken: To describe the atomic sources for replica non-determinism by a basic characterization of these effects. For the remainder of this section the effort is made to give such a characterization and argue for its completeness. The following three classes are introduced which will be discussed in the following three subsections:

- The real world abstraction limitation
- Impossibility of exact agreement
- Intention and missing coordination

### 4.3.1 The real world abstraction limitation

The first and most fundamental source for replica non-determinism is caused by the limitations that are introduced due to necessary abstractions which are made by computer systems. Computer systems function on a discrete[15] basis. To interface with continuos real world quantities, computers have to abstract these continuos quantities by finite sets of discrete numbers. Each discrete number represents an equivalency class of continuos numbers. This abstraction of continuos quantities by discrete numbers is a basic source for replica non-determinism. Before discussing the

[15]This book is only concerned with digital computer systems. Analog computer systems are not covered, the word *computer* is used as a synonym for *digital* computer.

limitations introduced by this abstraction we will show the preeminent importance of continuos quantities for real-time computer systems.

## The continuos real world

Nearly all quantities that are of interest for real-time computer systems are continuos. This can be shown by considering the basic units of measurement. The SI-systems (Système International d'Unités) as well as ISO 31 and ISO 1000 define the following basic units (see Table 4-3). All other units of measurement are derived from these basic units.

| *quantity* | *SI-unit* |
|---|---|
| distance | meter [*m*] |
| mass | kilogram [*kg*] |
| time | second [*s*] |
| electrical current | ampere [*A*] |
| thermodynamic temperature | degree kelvin [*K*] |
| gramme-molecule | mol [*mol*] |
| luminous intensity | candela [*cd*] |

*Table 4-3: SI units of measurement*

Each of these measurement quantities is described by the real numbers **R**. Truly discrete quantities can only be observed in the atomic structure of matter. For example the electron charge is given by $1.75896 \times 10^{-11}$ $Askg^{-1}$, the mass of an electron is given by $9 \times 10^{-31}$ $kg$. But almost any real-time system in practice is not concerned with such quantities at an atomic level. Real-time systems are rather concerned with macroscopic quantities which are continuos. Furthermore, the quantity of time—which is of utmost importance for real-time systems—is continuos. For distributed non real-time systems, for example data base systems, it seems at a first glance that there is no interaction with continuos quantities. All inputs and outputs to this system are made via digital devices like data terminals. But again, time has to be considered as continuos input quantity, if faults are considered as *input events* to the system. There is a close interaction between the continuos quantity *time of fault occurrence* and the systems internal notion of time. It follows that any fault-tolerant system has to deal with at least one continuos quantity, the quantity of time. Having shown that fault-tolerant systems and real-time systems especially are always concerned with continuos quantities, we will now turn to the real world abstraction limitations.

### Abstracting continuos quantities

The name *real world abstraction limitation* was chosen to indicate that the real world of continuos quantities has to be abstracted by discrete values and that this abstraction has certain limitations. Continuos quantities are described by dense sets of numbers. The quantities measured by the SI-units, for example, are described by the real numbers **R**. Computer systems, however, can only represent finite sets of numbers. Because of the finite amount of numbers, they are discrete numbers. Computer systems typically use the following abstraction for continuos quantities: Each discrete number represents an equivalency class of numbers from the dense set. For example the discrete number 100 can be used to represent all real numbers in the interval [100, 101[.

If a computer system has to observe a continuos quantity in the environment, it uses some kind of interface device which transforms the continuos real-world quantity to a discrete (digital) value. Examples of such interface devices are temperature sensors in combination with A/D-converters or a capture unit which records the timing of events. Any "real" interface device which is used to observe a continuos quantity has a finite accuracy. That is, for any real world quantity the interface device may deliver an observation out of the interval defined by the accuracy.

### Non-determinism of real-world observations

The limited accuracy is independent of the problem that continuos quantities have to be represented by discrete numbers. It is rather based on the fact that all entities of the physical world are finite and thus have only a finite accuracy. This limited accuracy is the reason why a *total* definition of replica determinism is impossible for systems which deal with continuos quantities: For any set of replicated services, which interacts with the real world, it is impossible to guarantee that continuos quantities are observed identically. Or, in other words: The observation of real world quantities introduces replica non-determinism.

Consider the following example. Two temperature sensors, combined with A/D-converters, are used to measure some temperature. Assume the temperature is exactly 100 °C. Furthermore it is assumed that the discrete number 99 represents the temperature range [99, 100[ °C and that 100 represents the temperature range [100, 101[ °C. If both components would have infinite accuracy both would deliver the discrete observation 100. But since "real" components have a finite accuracy it may easily happen that one result is 100 while the other result is 99.

This does not state that it is completely impossible to achieve replica determinism if real world quantities are observed. It rather states that replica non-determinism is introduced by observing real world quantities under the assumption (1) observations are of finite accuracy and (2) observed quantities are continuos. Because both assumptions hold for any real-time system, they have to be considered as a basic characterization for an atomic source of replica non-determinism. A formalization of this problem and a proof are given in [Pol94].

### Non-determinism of arithmetic operations

Besides the non-determinism introduced by real world observations the non-determinism introduced by arithmetic operations is also subsumed as a real world abstraction limitation. This problem is also caused by the fact that continuos real world quantities are abstracted by computer systems through discrete values. Computer systems perform arithmetic operations on discrete sets of numbers. These arithmetic operations are defined in analogy to their arithmetic counterparts on continuos numbers. Computers typically represent continuos numbers by means of floating- or fixed-point numbers (which are discrete numbers). The arithmetic operations on discrete numbers are inaccurate since the set of discrete numbers is only finite. The resulting inaccuracy can cause different results of arithmetic operations for a non-identical replicated system. Hence the results may show replica non-determinism even if the inputs are identical. The following three sources for this effect can be identified:

- ***Different representation of continuos numbers:*** There are two possibilities by which the representation of continuos numbers may differ. First, the sets of discrete numbers may have different cardinalities which results in different accuracies or ranges for representable numbers. The number of representable digits in a fixed point arithmetic for example depends on the cardinality of the set of discrete numbers. If the number of representable digits varies for different service implementations, the results may also be different. For floating-point numbers individual representations may differ even if the cardinality of the set of discrete numbers is equal. This second possibility of different representations is caused by partitioning the set of discrete numbers between mantissa and exponent differently. If for example $2^{80}$ discrete numbers are available and one server uses $2^{70}$ for the mantissa and $2^{10}$ for the exponent, while the other server uses $2^{64}$ for the mantissa and $2^{16}$ for the exponent, then the accuracy and range of representable numbers are different. Again this difference may lead to replica non-determinism.

- ***Different implementations of arithmetic operations:*** Even if the accuracy and the range of representable numbers are identical for all servers, the result of an arithmetic operation may differ among servers. The reason for this can be a different implementation of arithmetic operations. Because of the limited accuracy results have to be rounded or truncated. If the algorithm for rounding differs among replicas, this also may lead to replica non-determinism. An example for this behavior may be a set of replicated servers, where different floating-point units are used for calculations. Another example is if one server uses an arithmetic coprocessor while another server uses a software emulation for floating point calculations [Pöp93].

- ***Different evaluation order of arithmetic terms:*** But even in the case where all individual arithmetic operations are implemented identically, calculation results may be inconsistent. If an arithmetic term is broken down to individual arithmetic operations, there are many different sequences of arithmetic operations which are mathematically equivalent. Hence, different compilers or different programmers may evaluate arithmetic terms in different order. But since the mathematical equivalence does not hold for discrete arithmetic operations, the results of the calculations may differ among servers (see Table 4-1 and 4-2).

**Relating the real-world abstraction limitations**

There is a basic difference between the replica non-determinism introduced by real world observations and the replica non-determinism caused by arithmetic operations. In the case of the arithmetic operations, non-determinism may only be introduced by non-identical implementations of replicated servers. Hence, it is possible to avoid this source of non-determinism by implementing the replicated services identically (which also includes that all lower level services are implemented identically). On the other hand, the replica non-determinism introduced by real world observations is independent of whether replicated services are implemented identically or not. It is therefore impossible to avoid this source of replica non-determinism.

### 4.3.2 Impossibility of exact agreement

As the preceding subsection has shown, it is impossible to avoid the introduction of replica non-determinism at the interface between the real-time system and its environment. This subsection is concerned with the problem why it is impossible to eliminate or mask the replica non-determinism *totally* in a systematic manner, once it has been introduced. It will be shown that replica determinism can only be guaranteed in conjunction with the application semantics. For that reason the two possible approaches have to be considered.

- ***Analysis:*** Guarantee by analysis of the application semantics that non-deterministic observations have no effect on the correctness of the system's function. Non-deterministic observations are *similar* in most cases. If it is possible to guarantee by this similarity[16] that service responses in a replicated group will correspond to each other, then it is not necessary to exchange information on the individual observations. For replicated systems, however, it is very difficult to

[16]The effect of similarity has also been considered for real-time data access [KM93, KM92] but without taking replication into consideration.

carry out an analysis which shows that the effect of non-determinism has no effect on the correctness of the system's function. For most non-trivial systems the application semantics will not allow inconsistent observations. But even if it were possible to take this approach, it must be taken into consideration that the design and analysis of such a system is orders of magnitude more complex than of a system which does not have to deal with replica non-determinism at the application level. Any binary or *n*-ary decision that is based on a non-deterministic observation may lead to completely different execution paths and replica non-determinism. The only advantage of this approach is that no additional overhead for coordination of the non-deterministic observations is required on-line.

- ***Exchange of information:*** Using this approach the servers exchange information on their individual observations in order to improve their knowledge. This knowledge may in turn be used to select one single agreed upon observation for the continuos real world quantity. The main advantage of this approach is that non-determinism has to be considered only to a very limited extent at the application level. This reduces the application complexity considerably. The substantial complexity which is introduced by replica non-determinism can be masked under a protocol that achieves a notion of consensus. In the context of distributed computer systems the consensus problem was first introduced by Pease, Lamport and Shostak [PSL80] and has found broad consideration since. However, there are certain limitations to the state of agreement or knowledge that is attainable by a consensus protocol. Some states of agreement and knowledge cannot be reached by certain classes of systems, e.g., [FLP85, DDS87], while some states cannot be reached at all [HM90]. Furthermore, the complexity of such protocols in terms of time and information is high.

The first approach to ensure replica determinism is restricted to a very small area of applications. It is also an application-specific and unstructured solution. For this reason and because of the high complexity of system design and analysis, this approach is not suited for masking replica non-determinism in general. Rather the second and systematic approach, to exchange information on the individual observations of servers has to be considered for control of replica non-determinism. All servers should then act only on consenting values. Note that the requirement to exchange information on non-deterministic observations is not limited to the value domain. It has also to be considered for the domain of time. The servers have to agree upon the timing of service request. But, as mentioned above, it is important to recognize that the approach to exchange information cannot hide the complexity of distributedness and replica non-determinism completely (see the following).

## Limited semantics and states of knowledge

A protocol for information exchange can be considered as a means to transform a state of knowledge in a replicated group of servers. Also replica non-determinism can be considered as a state of knowledge which we would like to transform to another state of knowledge, namely replica determinism. There are various states of knowledge in a distributed system. Each state of knowledge defines a specific semantics for the achievable actions the system can carry out. The impossibility of eliminating replica non-determinism by an information exchange protocol lies in the inability of practical distributed systems to attain a certain state of knowledge. We follow Halpern and Moses [HM90] with their hierarchical definition of states of knowledge. Assume a group of servers $G$ and a true fact[17] denoted $\varphi$. According to the definition of Halpern and Moses the group $G$ has *distributed knowledge* of a fact $\varphi$ if some omniscient outside observer, who knows everything that each server of group $G$ knows, knows $\varphi$. This state of knowledge is denoted $D_G\varphi$. It is the weakest type of knowledge, since none of the individual servers necessarily has to have the knowledge of fact $\varphi$. The next stronger state of knowledge is when *someone* in group $G$ *knows* $\varphi$, denoted $S_G\varphi$. That is, at least one server $S$ in group $G$ has knowledge of $\varphi$. *Everyone* in group $G$ *knows* $\varphi$ is the next stronger state, denoted $E_G\varphi$. The state of knowledge where every server in $G$ knows that every server in $G$ knows that $\varphi$ is true is defined as *$E^2$-knowledge*. Accordingly *$E^n$-knowledge* is defined such that "every server in $G$ knows" appears $n$ times in the definition. The strongest state of knowledge, *common knowledge* $C_G\varphi$, is defined as the infinite conjunction of $E^n$-knowledge, such that $C_G\varphi \equiv E_G\varphi \wedge E_G^2\varphi \wedge \ldots \wedge E_G^n\varphi \wedge \ldots$ holds. It is obvious that these states of knowledge form a hierarchy with distributed knowledge as the weakest and common knowledge as the strongest state of knowledge.

## Replica non-determinism as distributed knowledge

For a single server as well as for a distributed system with common memory the different notions of knowledge are not distinct. If the knowledge is based on the contents of a common memory, a situation arises where all states of knowledge are equivalent $C_G\varphi \equiv E_G^n\varphi \equiv E_G\varphi \equiv S_G\varphi \equiv D_G\varphi$. However, the systems considered in this book have no common memory, they rather exchange information by means of messages. The various states of knowledge are therefore distinct in such systems. A replicated system that observes a continuos quantity in the environment starts at the lowest level in the knowledge hierarchy. Because of replica non-determinism each individual server may have a diverging discrete representation of the continuos quantity. The knowledge of the observations is therefore distributed knowledge $D_G\varphi$. Hence it is not possible, in the general case, to take actions that are based on dis-

[17] A formal definition of what it means for an server $S_i$ to know a given fact $\varphi$ is presented in [HM90].

tributed knowledge while ensuring replica determinism. Only an omniscient external observer, who knows every individual observation, has knowledge of one deterministic observation.

### "Someone knows" and its semantics

By sending all observations to one distinguished server the next stronger state of knowledge, someone knows $S_G\varphi$, is reached. At this state of knowledge it is possible that one distinguished server takes actions based on its knowledge. This may be sufficient for some applications, but there is a semantic difference between a single server with an omniscient state of knowledge and a replicated group of servers that has the state of knowledge someone knows: First, for some applications it is insufficient that only one server can take the action. Secondly, if all the replicated servers are required to base their further processing steps on the knowledge of fact $\varphi$, it is clearly insufficient that only one server has knowledge of $\varphi$.

### "Everybody knows" and its semantics

The next stronger state of knowledge, everyone knows $E_G\varphi$, may be attained if all servers exchange their observation with each other. Hence every server knows fact $\varphi$. But if the communication is unreliable then a certain server $S_i$ may not know whether some other server $S_j$ actually has knowledge of $\varphi$. Hence, the servers are still limited in their semantics because one server does not know whether any other server in the group has already taken the same action. This is a severe limitation since servers can not take actions which are dependent on the knowledge of other servers. For example, to commit an update operation in a distributed data base system, it is not sufficient that everybody knows whether to commit or abort. Rather the state of knowledge $E_G^2\varphi$ is required, i.e. everybody needs to know that everybody knows that commit will be carried out. This is typically achieved by 2-phase commit protocols [Gra78, Lam81].

### "Everybody knows that everybody knows ..." and its semantics

This next higher state of knowledge can be attained by an additional round of information exchange. Every server sends an acknowledgment message to every other server in the group to inform then that he now knows $\varphi$, so $E^2$-knowledge has been reached. By exchanging a total of $n$ acknowledgment message rounds the state of $E^{n+1}$-knowledge can be reached. But there is still one problems which cannot be solved by any state of $E_G^n\varphi$ knowledge with a finite $n$. One example is an atomicity property where a set of replicas in a group is required to take an action simultaneously or that none of the replicas take the action.

### "Common knowledge" as a requirement for *total* replica determinism

If all correct servers in a group take all actions simultaneously then it is impossible for a service user to differentiate between the service response of a single server or a set of service responses—in both cases the service is provided as specified. The perfectly synchronized service responses correspond to *ideal* replica determinism. But unfortunately it is impossible to achieve simultaneity which is equivalent to common knowledge in practical distributed systems (without common memory or perfectly synchronized clocks) [HM84]. Hence, it is impossible to mask replica non-determinism transparently and independently of the application semantics. However, for most application semantics simultaneity is not required.

### Relaxing the semantics towards achievable replica determinism

This negative conclusion on the possibility of common knowledge does not say that it is completely impossible to achieve replica determinism. This would be a contradiction of the fact that replicated fault-tolerant systems *do* exist and that these systems are able to handle replica non-determinism, e.g., [KDK$^+$89, CPR$^+$92]. It rather says that due to replica non-determinism a replicated group—that observes continuos real world quantities—can never behave exactly the same way as a single server. It is therefore only impossible to achieve replica determinism if one is not willing to drop the requirement of simultaneity. There are two possible approaches to restrict the semantics of a replicated group:

- By relaxation of common knowledge
- By "simulation" of common knowledge

In the following various possibilities of relaxing common knowledge are presented.

### Epsilon common knowledge

There are various possibilities to relax the notion of common knowledge to cases that are attainable in practical distributed systems. One such case is *epsilon common knowledge* [HM90] which is defined as: Every server of group $G$ knows $\varphi$ within a time interval of $\varepsilon$ time units. This weaker variant of common knowledge can be attained in synchronous systems with guaranteed message delivery. Since there exists an a priori known bound $\delta$ for message delivery in synchronous systems, it is therefore valid to assume that all servers will receive a sent message within a time interval $\varepsilon$. Just as common knowledge corresponds to simultaneous actions, this state of knowledge corresponds to actions that are carried out within an interval of $\varepsilon$ time units, or to replica non-determinism that is bounded by a time interval of $\varepsilon$ time units. This state of knowledge or replica determinism is sufficient for most practical systems. In the context of real-time systems epsilon common knowledge is a "natural" relaxation for common knowledge.

However, there is a basic difference in the interpretation of a fact $\varphi$. In the case of common knowledge, a fact $\varphi$ is *known* by all servers, in the case of epsilon common knowledge, a fact $\varphi$ is only *believed*. The former interpretation of $\varphi$ is a *knowledge* interpretation while the latter is an *epistemological* one. This difference in the interpretation of facts is not only true for epsilon common knowledge but for any achievable relaxation of common knowledge. It is always possible to construct a case where some server $S_i$ believes $\varphi$ and some other server $S_j$ believes $\neg\varphi$ at the same point in time since the requirement for simultaneity has to be dropped. This corresponds to a system behavior where replica non-determinism is observable at certain points in time.

### Eventual and time stamped common knowledge

Another relaxation of common knowledge is *eventual common knowledge* [HM90] which is a definition for asynchronous systems. According to this definition every server of group $G$ will eventually know $\varphi$. This state of knowledge can be attained in asynchronous systems. Eventual common knowledge corresponds to eventual actions and to eventual achievement of replica determinism. A more general definition is *time stamped common knowledge* [HM90, NT93], which defines that a server $S_i$ will know $\varphi$ at time $t$ on his local clock. Dependent on the properties of the local clocks either epsilon common knowledge or eventual common knowledge can be attained. The former state of knowledge requires approximately synchronized clocks while the latter is based on local clocks with a bounded drift rate, but no synchronization.

### Concurrent common knowledge

While the above presented relaxations of common knowledge were based on temporal modalities, it is also possible to define common knowledge without the notion of time by using the concept of potential causality [Lam78b]. To use the concept of causality rather than time is a viable alternative for asynchronous systems. It is possible to define *concurrent common knowledge* [PT88, PT92] by requiring all servers along a consistent cut[18] to have knowledge of fact $\varphi$. This state of knowledge corresponds to *concurrent actions*. A system that can attain concurrent common knowledge can also achieve replica determinism along some consistent cut. The time based relaxations of common knowledge are incomparable to the causality based concurrent

[18]Consistent cuts are defined such that if event $i$ potentially causally precedes event $j$ and event $j$ is in the consistent cut, then event $i$ is also in the consistent cut.

common knowledge [PT92]. However, there are actual systems which have communication services that have both semantics [BSS91].

## Simulating common knowledge

The other possibility of restricting the semantics of a replicated group is *internal knowledge consistency* [Nei88, HM90] or *simulated common knowledge* [NT87, NT93]. The idea is to restrict the behavior of the group in such a way that servers can act as if common knowledge were attainable. This is the case if all servers in the group never obtain information from within the system to contradict the assumption that common knowledge has been attained. In the context of replicated systems this corresponds to systems which *believe* that total replica determinism is achievable and have no possibility to prove the opposite. The system observes continuos quantities which are known to be non-deterministic. By exchanging information on these observations the system believes it has attained total replica determinism. Or in other words, the system believes that simultaneous agreement is reached on the replicated information.

This concept of simulating common knowledge has been formalized by Neiger and Toueg [NT87, NT93]. The class of problems where this simulation is possible is defined as *internal specification*. These are problems that can be specified without explicit reference to real-time, e.g., detection and prevention of deadlocks or atomic commitment. Based on this restricted class of specifications it is possible to simulate perfectly synchronized clocks. That is, all servers in the group cannot detect any behavior which contradicts the assumption that clocks are perfectly synchronized. It is possible to simulate perfectly synchronized clocks either on the basis of logical clocks or on real-time clocks [NT87]. Simulated perfectly synchronized clocks, however, are the prerequisite requirement for simultaneous actions and hence allows to attain simulated common knowledge.

An alternative approach to simulate perfectly synchronized clocks and common knowledge is the introduction of a *sparse time base* [Kop91]. Although this concept has not been presented in the context of distributed knowledge, it can be understood as simulating common knowledge. This concept is based on a synchronous system model with approximately synchronized clocks. However, relevant event occurrences are restricted to the lattice points of the globally synchronized time base. Thus time can be justifiably treated as a discrete quantity. The limits to time measurement in a distributed real-time system are used to establish criteria for the selection of lattice points [Kop91]. It therefore becomes possible to reference the approximately synchronized clocks in the problem specification while assuming that the clocks are perfectly synchronized. Hence, it is possible to simulate common knowledge and total replica determinism. Again the class of problems that can be treated by this approach is restricted to internal specifications. In this case it is possible to reference the approximately synchronized clocks, but it is not allowed to reference arbitrary external events occurring in real-time. For external events it is necessary to carry out a consensus protocol. But since it is possible to achieve a clock synchronization accuracy in the range of micro seconds and below [KO87, KKMS95], this simulation of common knowledge and simultaneous actions comes very close to perfect simultaneity for an outside observer.

### Comparing relaxations of semantics

For the purpose of achieving replica determinism, simulated common knowledge has the most desirable properties. The system can be built under the assumption of total replica determinism at the price of restricted problem specifications. This restriction—called internal specification—requires that there are no references to real-time. While this restriction may be valid for distributed database systems, it is certainly not in the context of fault-tolerant real-time systems. To guarantee timely response, a real-time system has to have a notion of real-time. By introduction of a sparse time base, it becomes possible to reference approximately synchronized clocks. Hence, it is possible to have a notion of real-time. But it is still impossible to reference arbitrary external events in the environment directly. System external events are related to real-time rather than to the approximately synchronized clocks. In particular, since faults are external events, it is impossible to handle the fault-tolerance with internal specifications. It is therefore important to differentiate between the occurrence of a fault and the recognition of the fault occurrence. The former is a system external event while the latter can be made system internal.

The other possibility is to relax the semantics of common knowledge. The time based eventual common knowledge as well as the causality based concurrent common knowledge are too weak for real time systems. These notions of common knowledge cannot guarantee a fixed temporal bound within which some fact $\varphi$ will become common knowledge. Alternatively the semantics of common knowledge can be relaxed to epsilon common knowledge or time stamped common knowledge. These two relaxations are the most suitable ones for real-time systems. Both are based on the synchronous system model of computation. In the case of epsilon common knowledge it is guaranteed that a fact $\varphi$ will be known within a time interval of $\varepsilon$ time units. Hence, this corresponds to within $\varepsilon$ time units bounded replica non-determinism.

### Practical possibilities to achieve replica determinism

A practical possibility for the achievement of replica determinism in a fault-tolerant real-time system is to use a relaxed state of common knowledge together with a simulation of common knowledge. For all problems that have no relation to system external events, common knowledge is simulated, based on a sparse time base. Correspondingly, total replica determinism can be simulated for this class of problems. The remaining problem specifications have to be solved by applying epsilon common knowledge. For these cases replica non-determinism has to be considered, for a maximum interval of $\varepsilon$ time units.

In practice there is a broad variety of protocols to achieve epsilon common knowledge. These protocols reflect the requirements in different application areas. Among them are consensus or interactive consistency [PSL80], a variety of broadcast protocols such as atomic broadcast [Lam78a, CASD85], causal broadcast [BJ87] or non-blocking atomic commitment [Gra78]. The actual kind of agreement that is needed depends on the application requirements.

### 4.3.3 Intention and missing coordination

The above presented limitations to replica determinism, the real world abstraction limitation and the impossibility of exact agreement are fundamental for computer systems. There is no way around these limitations and it is therefore impossible to achieve total replica determinism. The third and last characterization for the atomic sources of replica non-determinism is the "intentional" introduction of replica non-determinism. A typical example of intentional non-determinism is the usage of "true" random number generators.[19] The other possibility is that replica non-determinism is not introduced by "intention" but by omitting coordination for non-deterministic behavior. Examples for missing coordination are non-deterministic language constructs and usage of local information, as described in section "Possible sources for replica non-determinism".

Obviously, intention and missing coordination are atomic sources for replica non-determinism that can be avoided by proper system design. However, since it was the task to characterize all the atomic sources for replica non-determinism, this possibility has to be considered as well. Missing coordination also reflects the problem of design errors.

### 4.3.4 Characterizing possible non-deterministic behavior

In the previous three subsections a characterization has been given which describes all atomic sources for replica non-determinism. In the following the applicability of this characterization is shown. The various modes of non-deterministic behavior, as presented in the section "Non-deterministic behavior", can be characterized according to their atomic source. As shown above, the characterization consists of the three classes: (1) real world abstraction limitation, (2) impossibility of exact agreement

[19]Note that purely software implemented random number generators behave deterministically.

and (3) intention and missing coordination. Most modes of replica non-deterministic behavior are caused by more than one atomic source.

For example inconsistent order is caused if system internal events are related to system external events. This inconsistent order may be caused by the (1) real world abstraction limitation since an external event may be time-stamped with different values by different replicas. Furthermore, due to (2) the impossibility of exact agreement it is impossible that an internal event occurs at the same point in (continuos) time within two replicas. It is obviously possible to introduce inconsistent order of events (3) by intention or by missing coordination. The atomic sources for replica non-determinism are therefore orthogonal to the effects they cause. Omitting intention and missing coordination, which is an avoidable atomic source for replica non-determinism, the remaining two atomic sources can be classified as follows: Roughly speaking, the real world abstraction limitation corresponds to replica non-determinism that is introduced by observing the continuos quantities in the environment of the computer system. The impossibility of exact agreement corresponds to replica determinism that is introduced within the computer system. Or, in other words: the real world abstraction limitation is the system external atomic source for replica non-determinism, while the impossibility of exact agreement is the system internal atomic source for replica non-determinism. Most of the possible modes of replica non-deterministic behavior are caused by both atomic sources since the cause may either be system external or system internal.

## 4.4 When to enforce replica determinism

As the above presented results have shown, there is a broad variety of possibilities in a system which may lead to non-deterministic behavior. The possible sources for replica non-determinism indicate that replica determinism is not only destroyed by using esoteric or obviously non-deterministic functions, such as true random number generators, but even by standard functions such as timeouts, reading of clocks and sensors, dynamic scheduling decisions and others. This raises the question whether there are replicated systems which do not require special action to enforce replica determinism. From a theoretical point of view there are two cases when it is not necessary to enforce replica determinism. These two cases are considered in the following and it will be argued that both cases are not attainable in "real" fault-tolerant real-time systems.

### Absence of the sources for replica non-determinism

The first case where enforcement of replica determinism is not necessary is characterized by the absence of the sources for replica non-determinism. To avoid the real world abstraction limitation this would require a service specification which does not relate to continuos quantities in the environment, especially real-time. Furthermore, the individual servers have to be mutually independent so that they do not have to consider the impossibility of simultaneous actions. By considering these requirements it becomes obvious that there are no practical fault-tolerant real-time systems which do not observe continuos quantities in the environment and where replicas do not communicate with each other. Especially the observation of time is of utmost importance for real-time systems, which would not be allowed.

Since simultaneity is an ideal but unattainable property, it is valid to argue that it may be possible to relax this requirement only to a small extent, such that the differences in the timing of actions are negligible. To approximate simultaneity very closely, either a common clock service or common memory has to be used. The former allows a good approximation of simultaneity while the latter may be used to resemble common knowledge. But both approaches are unsuitable for fault-tolerant real-time systems because the assumption of independent replica failures is violated. The common clock as well as the common memory are single points of failures. Hence it follows that systems which are required to be fault-tolerant cannot avoid the atomic sources for replica non-determinism.

### Non-determinism insensitive service specifications

The second class of systems that does not require the enforcement of replica determinism is characterized by a service specification which is *insensitive* to the effects caused by replica non-determinism. Systems with such a specification function correctly if servers are simply replicated without considering non-deterministic behavior. It is, however, practically impossible to build a non-trivial replicated system that exhibits fault-tolerance without enforcement of replica determinism. Only trivial service specifications are insensitive to the effects of replica non-determinism. But even the simple example where two servers have to control an emergency valve that has to be closed in case of excess pressure shows the need for enforcement of replica determinism. Because the readings of the replicated pressure sensor behave non-deterministically, both servers will diverge unacceptably.

On a formal basis the insensitivity may be defined as a *self-stabilizing* property [Dij74]. A system is said to be self-stabilizing if, starting from any state that violates the intended state invariant, the system is guaranteed to converge to a state where the invariant is satisfied within a finite number of steps. In our case we are interested in the invariant which is defined by replica determinism. Hence, self-stabilizing guarantees that the system converges from any non-deterministic state within a finite number of steps to a state of replica determinism. However, self-stabilization is a very strong property which is difficult, and in some cases even impossible, to achieve. It is thus extremely unlikely that self-stabilization is achieved by chance when implementing a service specification.

The following example is used to illustrate the possibility of a very simple replicated system without enforcement of replica determinism. Consider an automatic-control system which is composed of *n* replicas. Each of them consists of a sensor to read the controlled condition, a processor acting as a PD-controller, and an actuator to output the operating variable. This control structure has no binary or *n*-ary decision points. Furthermore it has only a limited history state. The P-controller requires only the actual controlled condition while the D-part of controller requires the actual and the previous observation of the controlled condition. Consequently, non-deterministic observations of the controlled condition cannot add up over time. Due to this simple structure it can be guaranteed that the results between individual replicas diverge only within a certain a priori known bound.

However, a change in the structure from PD-control to PI-control invalidates the statement that replica determinism need not be enforced. The I-part of the controller adds the difference between the controlled condition and the set point over time. It follows that differences, introduced by the replica non-determinism of controlled condition observations, add up over time. Since the direction or amount of differences that add up are unpredictable, it becomes impossible to bound the different results of the replicated controllers. This example shows that minor changes of the specification may require the introduction of replica determinism enforcement.

Another problem with the property of self-stabilization is the fact that it does not guarantee continuos correct service behavior in the presence of faults. It guarantees only that the service eventually converges to correct function after the occurrence of a fault. The duration of this convergence period is a critical factor for real-time systems. It is important that the convergence is strong enough to ensure an acceptable behavior of the system. At the current stage of research the design of self-stabilizing services with a guaranteed convergence time is not well understood [Sch93a]. Taking this fact together with the fact that almost any non-trivial specification does not guarantee self-stabilizing behavior shows that replica determinism must be enforced. To build replicated real-time systems the focus therefore has to be put on appropriate methodologies to enforce replica determinism.

# Chapter 5

# Enforcing replica determinism

The necessity to enforce replica determinism in non-trivial systems has been shown in the previous chapter. This chapter is concerned with methodologies and implementations that are appropriate to enforce replica determinism in the context of fault-tolerant real-time systems. Since it is impossible to achieve ideal replica determinism, all implementations—as presented in this chapter—relax the semantics to achievable cases of replica determinism. Achievable semantics are either based on epsilon common knowledge or on concurrent common knowledge.

When enforcing replica determinism—which is often called *replica control* or *coherency control*—there are two questions to classify different methods: *where* and *how*. Server internal or server external replica control are the answers to the question *where*. This aspect of replica determinism enforcement will be treated in the next section. Possible answers to the question *how* to enforce replica determinism are "central" or "distributed". Both approaches to replica control are compared and discussed in section two.

Following these two sections, in section 3, the problem of replica determinism enforcement under real-time constraints is surveyed in the context of the communication problem for distributed systems. Depending on the replication strategy there are different requirements the communication service must fulfill: establishment of consensus, reliable broadcast with additional ordering requirements. The properties of these protocols are discussed for synchronous as well as for asynchronous system architectures.

Section 4 treats the relation between synchronization and replica determinism enforcement. In this context synchronization is understood as a means to control the amount of inconsistency between replicas. Synchronization can therefore be considered in the value domain as well as in the time domain. While it is argued that only strictly synchronous approaches are appropriate in the value domain, in the time domain there is a broad spectrum of possibilities.

In section 5 the interdependence between replication strategy and the failure semantics of servers is discussed. The possible failure semantics a server may exhibit is closely related to the degree of centralism or distributedness of the replication strategy. In addition, the actions that are necessary to recover after server failures depend on the replication strategy.

Finally, section 6 discusses the problem of redundancy preservation. To guarantee correct function while providing fault-tolerance, the level of redundancy has to be maintained above a given threshold in a replicated system. Depending on the failure semantics of servers, conditions for the correct function of server groups are given. Furthermore, the problems with on-line adding and removing servers to or from a group are discussed.

## 5.1 Internal vs. external

The degree of server internal non-determinism depends on the functions that are used to implement a given service. Server external non-determinism on the other hand depends on the characteristics of the server's environment, such as communication services and sensors. A server group may minimize internal non-determinism by restricting the functions used to implement a given service. This means avoiding non-deterministic program constructs, exclusively using global information, avoiding uncoordinated timeouts, taking no dynamic scheduling decisions, using a global coordinated time service, etc. Furthermore, diverse implementations of parallel active services should be ruled out to avoid the consistent comparison problem. Server internal enforcement of replica determinism is therefore defined as follows:

***Server internal replica determinism:*** A server group enforces internal replica determinism if correct servers show *correspondence* of server outputs and/or service state changes under the assumption that all servers within a group start in the same initial state, executing identical service requests.

This definition of internal replica control is very similar to the definition of replica determinism, except that all external inputs to the server are assumed to be identical. Since time is also considered as external input to servers, this definition requires the clocks of individual servers to be identical. Internal replica determinism is a property which cannot be verified in a "real" system due to limited accuracy (there are no perfectly synchronized clocks). A server group implements internal replica determinism enforcement if the atomic source for replica non-determinism, intention and missing coordination, is absent. However, it is only possible to reduce non-deterministic behavior, since the remaining two atomic sources, namely the real world abstraction limitation and the impossibility of exact agreement cannot be prevented by internal replica determinism enforcement. Server internal replica determinism is therefore an abstraction which can be enforced partially by defining a set of functions to implement a service and by assuring that services are replicated identically.

The restrictions of server groups to internal replica determinism enforcement has in some cases undesirable properties. For example there are applications that have to be written in Ada, but the Ada language has non-deterministic statements [TS90]. Another example is dynamic scheduling, which is desirable in some application areas to obtain good resource utilization. But dynamic scheduling destroys internal replica determinism. A further severe restriction with internal replica determinism enforcement is the requirement to replicate servers identically, which is in many cases unsuitable. One reason for this lies in the basic assumption of fault-tolerant replicated systems which requires that servers fail independently. Hence it is desirable to implement servers diversely. Another reason for non-identical replication is the need for resource sharing. For example, two processors are used as servers for a replicated service, but, additionally, each of the processors is also used for execution of different sets of non-replicated services. Since both processors have different sets of services to execute, the timing of service executions will differ if on-line scheduling is used. It may happen for example that timeout decisions will be different due to the different execution timing of servers. In all cases where internal replica determinism is violated, the resulting non-deterministic decisions have to be made globally available, such that agreement can be achieved by the whole server group. Interactive achievement of group-wide agreement incurs a considerable temporal overhead and consumes communication bandwidth. The number of non-deterministic decisions which need to be resolved on a group-wide basis depends on the service implementations. The more often agreement has to be achieved the bigger the overhead becomes. Especially in the case of an on-line scheduler it is impossible to attain group-wide agreement on each scheduling decision. The frequency of scheduling decisions would be too high for typical real-time systems. Hence, it is necessary to find a compromise between the functional restriction for internal replica determinism enforcement and the communication overhead to resolve non-deterministic decisions by the whole group. For example the "State Machine Approach" as described by Schneider [Sch90] and the MARS system [KDK$^+$89] require servers to implement internal replica determinism. The ISIS system [BJRA84, Bir93], the work described in [TS90] or Delta-4's semi-active replication approach [Pow91] does not require servers to implement internal replica determinism, but rather to reach agreement on non-deterministic decisions.

It is impossible to avoid the introduction of server external replica non-determinism, as has been shown for the real world abstraction limitation and the impossibility of exact agreement in the previous chapter. Fault-tolerant clock synchronization [LM85, KO87] is one example of discrete valued sensors—namely clocks—which shows the requirement to control replica non-determinism. For continuos valued sensors, the problem of replication and fault-tolerance is treated by Marzullo and Chew [Mar90, CM91]. In the field of computer vision the problem of replicated sensors is also one of relevance. In this area one is concerned with acquiring consistent information from dissimilar images showing the same object from a different perspective. This problem—called *sensor fusion*—is treated without resort to real-time requirements, e.g., [Hag90, Hun92, Per92]. It is a principle requirement that any system which uses replicated sensor information needs to coordinate this information. Depending on application considerations this coordination may take place directly after reading the sensor information or later after processing of the sensor information. Regardless whether sensors are discrete or continuos, replica determinism and fault-tolerance depend on the selection of a proper voting or adjudicating function [GS90] to coordinate sensor inputs. Such coordination functions have to be chosen to reflect the individual application requirements.

Just as it is necessary to control replica non-determinism of sensor observations, the same is true for communication services. To minimize non-determinism introduced by communication, the characteristics of the communication service have to be chosen properly. Important characteristics are the availability of reliable broad- or multicast services, the ability to reach consensus and, furthermore, preservation of order among messages. The requirements for server external deterministic behavior are most commonly decomposed into the following basic abstractions [Sch90, Cri91b].

- ***Membership:*** Every non faulty server within a group has timely and consistent information on the set of functioning servers which constitutes a group.
- ***Agreement:*** Every non-faulty server in a group receives the same service requests within a given time interval.
- ***Order:*** Explicit service request as well as implicit service requests, which are introduced by the passage of time, are processed by non-faulty servers of a group in the same order.

It is important to note that these abstractions are only able to reduce replica non-determinism, but they cannot prevent non-determinism. Due to the impossibility of exact agreement, the abstractions of agreement and membership are not *total* properties, rather they have to be relaxed for "real" systems. As discussed in the subsection "Impossibility of exact agreement", it is impossible to achieve common knowledge. Only weaker states of agreement, such as epsilon common knowledge or concurrent common knowledge can be attained. The agreement and membership abstractions are therefore inconsistent during a certain time interval. Hence, it is only possible to require that membership and agreement is reached within an a priori fixed time interval. If not mentioned explicitly, for the remainder of this chapter it is assumed that the abstractions of agreement and membership are relaxed to achievable cases.

The definitions of order and agreement both imply a system-wide group membership view. Hence, consistent group membership information is a prerequisite for replica control. When defining the order property, membership changes have to be considered. If, for example, a new server joins a group, the integration has to be done by synchronizing the new servers service state to the other servers in the group (see also section "Redundancy preservation"). The abstractions of membership, agreement and order may be relaxed to use cheaper protocols. This, however, requires semantic knowledge of service requests. The order requirement may be relaxed, for example, if service requests are known to be commutative or independent. Agreement may also be relaxed if servers are implemented by $N$-version programming such that identical services may have slightly different service requests due to different implementations. Another example of protocol optimization by taking advantage of application semantics is described in [BJ87a]. Hence, these solutions are application specific solutions to fault-tolerance. Their major disadvantage is that they burden the application programmer and add complexity to the system design. Besides exploitation of semantic knowledge there are other system parameters exerting influence on the communication complexity (see section "Communication").

For real-time systems the tradeoff between internal and external enforcement of replica determinism will be tilted towards internal enforcement for the following reasons. Firstly, the processing speed in most systems is at least one order of magnitude higher than communication speed [Gra88]. Hence, it is faster to do more processing because of functional restrictions than to do more communication (and processing) for the execution of a consensus protocol. Secondly, consensus requires global synchronization which makes scheduling of processors and communication media more difficult and lowers resource utilization.

## 5.2 Central vs. distributed

To classify different replication control methods, the degree of centralism or distributedness is a useful taxonomy. This taxonomy is basic since it covers all possible approaches, and furthermore has impact on the communication complexity of replica control. At the one end is the strictly *centralized* or *asymmetric* approach: there is one distinguished server to which all remaining servers in the group are synchronized. The central server controls replica determinism by forcing the follower or standby servers among the group to take over its decisions and processing pace. The term *follower servers* [Pow91c] is used for servers that are receiving and processing service requests together—but slightly delayed—with the central server. The notation *standby server* [BMST92a, BMST92b] is used for servers which do not receive service request but rather information on service states by means of checkpoint messages from the central server. Consequently, the central approach is characterized by the different communication protocols that are executed by the central server on the one hand and the remaining servers in the group on the other hand. Examples are the semi-active and passive replication strategy of Delta-4 [Pow91c, Pow91d], the ISIS system [BJRA84] or database oriented systems which use checkpointing [SLL86, KT87]. The obvious advantage of the central approach is the simplicity of communication protocols to achieve order. Figure 5-1 illustrates the principle of a central approach to replication. There are two server groups, where each group has one central server which is attached to the communication subsystem.

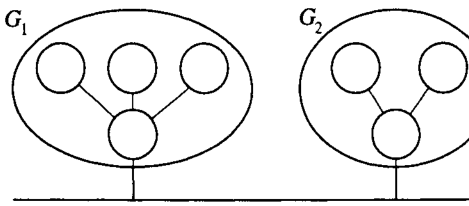

*Figure 5-1: Central replica control*

On the other end is the *distributed* or *symmetric* approach: there is no leader role. Each server within the group performs exactly the same way. Hence all servers in the group execute exactly the same communication algorithm. Figure 5-2 shows two server groups without any central server. To guarantee replica determinism the server group has to reach consensus on non-deterministic decisions and the processing pace. Examples are MARS [KDK+89], Totem [AMM+93], Delta-4's active replication [CPR+92] and MAFT [KTW+88].

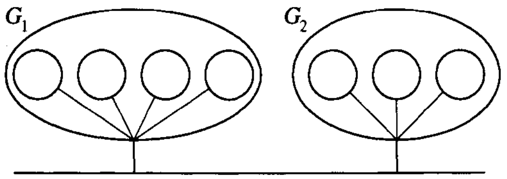

*Figure 5-2: Distributed replica control*

If group internal as well as group external communication is relayed via the central server, then order is guaranteed implicitly. Group internal non-deterministic decisions may also be resolved by sending them to the central server. This in turn is also the major disadvantage because a single failure of this central server is critical. As a consequence, the central server cannot have byzantine failure semantics. The distributed approach on the other hand has the advantage of being independent of any single server. Hence, there are no restrictions on the failure semantics of individual servers. An example of a somewhat intermediate approach is the broadcast protocol for the Amoeba system [KT91]. This protocol uses message sequence numbers that are generated centrally while the remaining communication is distributed. Another example is the rotating token site used by the broadcast protocol described in [CM84]. The token site (or central server) changes with the progression of time although the communication algorithm is central.

From the viewpoint of real-time systems there is a slight preference for distributed replica control since a central server may easily become a performance bottleneck. Furthermore, the remaining servers in the group lag behind the central server because they have to await the decisions taken by the central server. Regardless whether a central or distributed approach has been selected, the whole group is involved in reaching consensus. The determining factor for central or distributed replica control will therefore be failure semantics. Nevertheless, both approaches to replica control, central as well as distributed, have common requirements on communication services, which will be discussed in the next section.

## 5.3 Communication

Communication among servers is basic to replica control as well as for replication strategies in general. By exchanging information on non-deterministic server states the degree of replica non-determinism can be minimized. That is, the state of knowledge is transformed from non-determinism to a higher state of knowledge.[20]

### 5.3.1 Replica control and its communication requirements

The complexity of communication services in time and information depends on various parameters. The most important parameters are: the service specification, the fault hypothesis, whether processing or communication delays are bounded or unbounded, and the network topology. Dependent on the replica control strategy there are different requirements for communication services:

- ***Distributed replica control:*** Any distributed replica control strategy requires a communication service assuring consensus and ordering properties.
- ***Central replica control:*** The requirements on the communication service for central replica control depends on the processing mode of the follower or standby servers: (1) If the follower servers are processing service requests together with the central server, then a communication service is necessary which assures reliable broad- or reliable multicast (2) If the standby servers are not processing service requests but rather receive checkpoint information, then a point to point communication service which guarantees FIFO message delivery without duplication is sufficient.

The main difference between both replica control strategies is that distributed replica control requires the communication service to establish order. The central approach to replica control can establish order by relaying service requests via the central

[20]For the various states of knowledge see subsection "Impossibility of ideal distributed computing".

server. If the central server and the follower servers are processing service requests in parallel, a reliable broad- or multicast communication service is necessary, since all servers have to receive identical service requests, even if the central server fails. If the service requests are not sent to all servers in the group but only to the central server, then a FIFO point to point communication service without message duplication is sufficient. In this case the central server sends state or checkpoint messages to its standby servers after having received a service request. For the remainder of this section the point to point service is not considered because implementing a service with FIFO ordering and no message duplication is relatively straightforward using sequence numbers.

In contrast, the distributed approach to replica control requires a server group not only to agree on a single input value, but also on the value and order of several input values from individual members in the requesting group. Since there is no central server, requests have to be sent to each server within the group[21] individually. It is therefore of paramount importance that the communication service delivers order properties. A distributed consensus protocol can establish total order if server local views of the order are input to the consensus protocol. However, the main purpose of the consensus protocol is to reach agreement on one service request since every service request is sent to all group members individually. Note that if replicated sensors are used, the sensor readings also have to be considered as service requests. In this case a consensus protocol has to be executed to resolve the non-determinism introduced by the sensor observations.

## 5.3.2 Consensus protocols

The consensus problem was first defined by Pease, Shostak and Lamport [PSL80]. It is sometimes also referred to as *byzantine agreement* [LSP82]. Protocols to solve the consensus problem have been extensively studied under different assumptions on the system model and the fault hypothesis.[22] The properties of a distributed consensus service are defined as follows:

***Consensus:*** Each server starts a protocol with its local input value, which is sent to all other servers in the group, fulfilling the following properties:

- *Consistency:* All correct servers agree on the same value and all decisions are final.
- *Non-triviality:* The agreed-upon input value must have been some server's input (or is a function of the individual input values).

[21]For the remainder of this section it is assumed that consistent group membership information is available.

[22]A survey of different consensus protocols is given in [BM93]

- *Termination:* Each correct server decides on a value within a finite time interval.[23]

In the following, important properties of the consensus problem are presented. It is known that the consensus problem has no deterministic solution in systems where servers are asynchronous and communication is not bounded in the presence of even one single failure [FLP85, DDS87]. While consensus can be achieved in asynchronous systems by non-deterministic algorithms [Ben83, Bra87] or by *failure detectors* [CT91], real-time systems are per definition synchronous or at least partially synchronous. Under both assumptions consensus may be achieved deterministically [DLS88]. Under the assumption that the communication topology is point-to-point and up to $t$ failures have to be tolerated, the number of servers $n$ has to be selected as follows: For servers which exhibit fail-stop, crash, omission or authentification detectable failures $n$ have to be at least $t + 1$. Under a byzantine failure assumption $n$ has to be at least $3t + 1$. These bounds are both tight. It is also known that the connectivity of the network has to be at least $2t + 1$ under a byzantine failure assumption, while weaker assumptions require a connectivity of at least $t + 1$. These bounds are also tight [Dol82]. To tolerate up to $t$ failures, any protocol requires at least $t + 1$ rounds to achieve consensus in the worst case [FL82]. There exist *early stopping* algorithms [DRS90, TPS87, GM95] that are algorithms requiring only $O(f)$ rounds when the actual number of failures $f$ is less than the maximum number of failure such that $f \leq t$. The lower bound $min(f+2, t+1)$ on the number of rounds for early stopping algorithms was also shown in [DRS90]. The minimum bound on the number of messages that have to be exchanged under a byzantine failure assumption is $O(nt)$, where $n$ is the number of servers. For authentification detectable byzantine failures or omission failures the number of messages is $O(n + t^2)$ [DR85]. While these bounds on the message complexity are tight, no such bound is known for crash or fail-stop failures. Recently it has been shown that fully polynomial consensus can be attained in $t + 1$ rounds [GM95], i.e., the communication protocol uses a polynomial amount of computation and the message length is also polynomial.

All results presented above are based on the round abstraction. A round consists of an arbitrary number of communication and processing steps, but each message sent in a given round cannot depend on messages received in the same round. This implies that round based protocols require all participating servers to have a priori knowledge of the initiation time of each protocol execution. A typical possibility for approximately simultaneous protocol initiation is to use a priori agreed upon start times, based on approximately synchronized clocks. If the knowledge of the protocol initiation time is not known a priori, a *distributed firing squad* protocol can

[23]To reflect real-time requirements, this definition restricts the general definition which only requires that each server decides within a finite number of processing steps.

be used to initiate a consensus protocol. There are no early stopping algorithms for the distributed firing squad problem. Under these assumptions the $t + 1$ round's barrier is the best case [BL87, CD86]. Alternatively a *distributed bidding* protocol may be used [BGT90]. This protocol may be started by any server, without a priori knowledge of the protocol initiation time, and guarantees that all correct servers eventually agree on the subsequently delivered inputs of the remaining servers. Distributed bidding protocols are early stopping, the bounds are $min[(f+4)D, (t+2)D]$, $min[(f+5)D, (t+4)D]$ for crash and byzantine failures respectively, where $D$ is the maximum communication delay [BGT90]. However, such a protocol does not achieve (approximate) simultaneous agreement but rather eventual agreement. Hence, it follows that the property of (approximate) simultaneity, as guaranteed by the distributed firing squad protocols is harder to achieve than the eventual agreement of distributed bidding protocols.

These theoretical results indicate the high complexity of the consensus problem. Experience indeed has shown that the complexity of consensus under byzantine failure assumptions is prohibitive for many real-time applications, as for example performance measurements of SIFT [PB86] or FTMP [CSS85] have shown.

### 5.3.3 Reliable broadcast protocols

To reduce the communication complexity, many practical and experimental real-time systems are based on more benign failure assumptions than authentification detectable byzantine or byzantine failures are. These systems most often use reliable broadcast rather than distributed consensus as their basic communication service, i.e. one server broadcasts its input to a group of servers, where all correct servers agree on the input, if the transmitter was not faulty. The advantage of this protocol is that there is only one service request input to the protocol. Hence, it is not necessary to agree on one service request out of a set. Furthermore, since there is only one input (one server is sufficient to start the protocol) there is no requirement that all participating servers have to know the protocol initiation time in advance. The definition of a reliable broadcast service is very similar to the definition of the consensus problem. The main difference lies in the non-triviality property, which is given below [BD84]:

***Reliable broadcast:*** A distinguished server, called the transmitter, sends its local service request to all other servers in the group, fulfilling the following properties:

- *Consistency:* All correct servers agree on the same value and all decisions are final.
- *Non-triviality:* If the transmitter is non faulty, all correct servers agree on the input value sent by the transmitter.

- *Termination:* Each correct server decides on a value within a finite time interval.[24]

The non-triviality property reflects the difference between consensus and broadcast protocols. While all servers deliver their input to the consensus protocol, only the transmitter delivers its input to the reliable broadcast protocol. It is therefore required that all correct servers agree on the value input by the transmitter. Most replicated real-time systems use reliable broadcast as their basic communication service for the following reasons:

- The possibility of—compared to consensus protocols—more efficient implementations.
- The fact, that under non byzantine failure assumptions only a few decisions require consensus.
- Order properties and consensus are relatively easy to implement on top of many reliable broadcast communication services.

### Broadcast communication topologies

While the consensus problem was studied under the assumption of point-to-point communication network topologies, see Figure 5-3, the close resemblance between broadcast communication and broadcast network topologies, see Figure 5-4, allows a substantial reduction of the communication complexity.

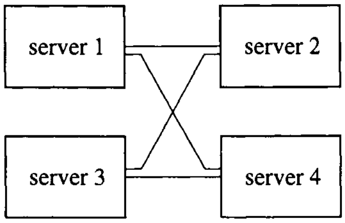

*Figure 5-3: Point-to-point topology*

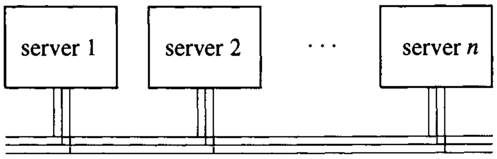

*Figure 5-4: Broadcast topology*

[24]Again the general definition which requires that each server decides within a finite number of steps is restricted.

Typical communication channels for real-time systems such as Ethernet [MB76] or CAN [CAN85, SAE92] have broadcast properties, i.e. a sent message is transported by the physical channel such that a group of communicating participants will receive the message. If it is guaranteed that a broadcast channel either transports a message to at least $b$ receivers or to none of them, then the channel is said to have a *broadcast degree* of $b$. With a broadcast channel that guarantees a certain broadcast degree it is possible to implement reliable broadcast with fewer rounds than $t + 1$: Under a byzantine failure assumption 2 rounds are sufficient if $b > t + n/2$ ($t$ is the maximum number of tolerable faults, $n$ is the number of servers in a group). Under the assumption of omission failures 2 rounds are sufficient if $b > t$ and $t - b + 3$ rounds are sufficient for $2 \leq b < t$ [BSD88]. Experimental results indeed indicate that the assumption of atomicity, such that a message is transported to at least $b$ receivers or none, is justified for some spatially limited broadcast channels [Rei89]. By assuming that $b = n$, the MARS system as well as the TTP protocol are able to implement reliable broadcast in only one round [KDK+89, KG94]. To tolerate faults, for space redundancy, messages are transmitted over $r$ redundant channels. For time redundancy, each message is repeated $t + 1$ times, even in the absence of failures. Since the protocol execution is identical in the fault free case as well as in the case of faults, these protocols have a very small temporal uncertainty.

## Order properties

While consensus protocols can establish agreement and hence order, the properties of reliable broadcast are too weak to establish order. For replica control, however, the establishment of order is very important. Hence, it is necessary to specify and implement reliable broadcast protocols that guarantee additional order properties. Possible order properties are FIFO, causal,[25] and total order [HT93]. Reliable broadcast protocols which guarantee a total order for the delivery of messages are called atomic broadcast protocols, i.e. if two services S1 and S2 receive two service requests $r_1$ and $r_2$ then S1 receives $r_1$ before $r_2$ if and only if S2 receives $r_1$ before $r_2$. While causal order is strictly stronger than FIFO order, total order is an independent property. Hence, possible protocol specifications are reliable broadcast, FIFO broadcast, causal broadcast, atomic broadcast, FIFO atomic broadcast, and causal atomic broadcast. Whether FIFO or causal order is required depends on the application semantics. Total order should be considered as a default for real-time systems, because for any weaker order property, consistency with the application semantics has to be shown. This is because total order guarantees that service requests, which are sent from different servers to members of a group, are processed by the group members in the same order. Correspondingly, server groups should be able to achieve total order. Some im-

[25]The word *causal* is used in this context to reflect a *potentially* causal relation among events as defined by Lamport [Lam78b].

plementations of reliable broadcast protocols for replicated systems guarantee order properties, such as FIFO, causal, or total order. There is even a trend to integrate not only order properties, but also a membership service into reliable broadcast protocols, e.g. ISIS [Bir93], TTP [KG94] or Totem [AMM+93].

Whether an implementation of causal ordering semantics at the level of the communication service is beneficial or not depends on the application semantics [CS93]. In most real-time systems, however, causal ordering is not implemented for two reason. Firstly, real time systems are open systems that interact closely with the environment. Causal ordering at the communication system level cannot capture causal relations outside the scope of the communication system. Hence, false causal dependencies are introduced. Secondly, real-time systems with approximately synchronized clocks can establish temporal order. Temporal order includes causal order of $p$-precedent events [VR89] without introduction of false causality. That is, temporal ordering guarantees correct causal ordering of events that are at least $p$ time units apart. For these reasons most real-time systems implement total and temporal order as the default ordering property of the communication system. On the other hand, if the application semantics allows a reliable broadcast protocol with weaker ordering properties, it is possible to use protocols that are cheaper to implement. One example is the CBCAST communication service of ISIS [BJ87b]. But since exploitation of application semantics burdens the application programmer, a compromise between general applicability of the communication service and efficiency has to be taken. Recent developments have shown that it is even possible to implement very efficient protocols with total ordering properties [AMM+93]. Whether the broadcast protocol itself is required to guarantee total order or not depends on the distributedness of the replica control strategy. Any central replication strategy does not require total order because order can be established by the central server.

Strictly asynchronous distributed deterministic protocols without randomization or failure detectors can generate only FIFO or causal order, but no temporal and total order such as synchronous protocols. This is because consensus can be reduced to atomic broadcast[26] and consensus is not solvable deterministically in asynchronous systems. The reason for this impossibility to achieve total order is due to the fact that a server cannot decide in an asynchronous system whether there are messages outstanding that would change the total order once delivered. Since messages may be delayed for arbitrary periods, total order can never be established. On the other hand, atomic broadcast can be solved by randomization or failure detectors in asynchronous systems with crash failures. Again, the reason is that consensus can be solved under these conditions [Ben83, CT91] and consensus is reducible to atomic broadcast.

[26]Atomic broadcast can be used to implement consensus: Every server proposes a value which is sent by atomic broadcast. To decide on an agreed value, each server picks the value which was first received. By total order of atomic broadcast, all correct servers pick the same value. Hence, the properties of consensus are satisfied.

Most practical implementations of "asynchronous" reliable broadcast protocols, e.g. Totem [AMM+93], therefor assume a partially synchronous system or they use fault detectors to achieve total order, since total order is a very important property.

## Asynchronous broadcast protocols

As with consensus, reliable broadcast protocols can be divided into synchronous and asynchronous protocols. Asynchronous protocols, e.g., Chang and Maxemchuck [CM84], ABCAST, GBCAST and CBCAST [BJ87b], Trans/Total [MM89 MMA90], Amoebas broadcast protocol [KT91], Delta-4's AMp [VRB89] or Totem [AMM+93] are based on acknowledgment and retransmit schemes. They typically generate order by a central approach: Examples are ABCAST where the sender selects a maximum priority from all receiver acknowledgments [BJ87b]. Chang and Maxemchuck use a central server to order messages [CM84], but this central server, called token site, rotates among the members in the group with the passage of time. Delta-4's AMp uses a rotating token which is passed among servers, message transmission is only allowed if a server owns the token. Hence, the order of messages is determined by the token passing order [Pow91e]. Another protocol which uses a logical token ring for message ordering is the Totem protocol [AMM+93]. Furthermore, Totem also achieves total ordering in multiple logical rings that are interconnected by gateways. The Amoeba system uses a central sequencer where each transmitter has to request a sequence number which is responsible for generation of total order [KT91]. Another central approach to establish total order is the propagation graph algorithm of Garcia-Molina and Spauster [GS89]. This algorithm transmits messages along a point-to-point network topology which forms a spanning tree. It is therefore guaranteed for each pair of messages that they are routed over at least one common server. Hence, it is possible for this server to establish order. The Trans/Total protocol [MM89, MMA90] is an exception since it is the only asynchronous protocol that attains total order by a strictly distributed approach. Determination of the total order does not occur immediately after a message is broadcast but must wait for reception of broadcasts by other servers. Each server builds a dependency graph which is based on the last received broadcast messages and its acknowledgments. Based on this dependency graph it is possible to include received messages into the total order.

Asynchronous protocols are not well suited for real-time applications because they incur considerable temporal overhead to establish total order. This overhead to establish total order is introduced since lengthy timeouts have to be used for fault detection to avoid too many false timeouts. In the case of central protocols, messages have to be relayed via a central server. This takes some additional time for communication and message processing. In the case of the distributed protocol, order is established some number of messages behind the actual broadcast. In addition, the computation of the dependency graph as well as garbage collection requires substantial processing resources. Among asynchronous reliable broadcast protocols logical token passing protocols are best suited since they have little overhead for establishment of total order, and furthermore they are not dependent on a single central server [AMM+93]. However, for real-time systems the temporal overhead for token regeneration has to be considered if the token is lost due to a communication fault.

## Synchronous broadcast protocols

Compared to the acknowledgment retransmit schemes of asynchronous protocols, synchronous protocols are mostly clock-driven [BD84, CASD85, Cri90, KG94], i.e. the protocol progression is guaranteed by the advance of time. Total order is established by assigning time stamps to messages. These received messages have to be delayed for some fixed time—the *action delay*—to guarantee that there are no messages underway which were generated earlier. After the action delay, the total order of service requests is stabilized in all servers. This ordering approach is strictly distributed. The action delay's duration is predominantly determined by the temporal uncertainty of the communication protocol's latency [KK90]. To keep the action delay low, which is important for real-time systems, the CPU scheduling discipline as well as the communication media access strategy have to be chosen accordingly [Kop86]. Most often synchronous communication protocols are based on approximately synchronized clocks, but it is also possible to build clock-less synchronous broadcast protocols [Ver90]. This is only of relevance to communication services with a large temporal uncertainty where the expected message delivery time is typically close to the minimum and the maximum message delivery time is much larger [HK89]. It is therefore faster on the average to wait for acknowledgment messages, than to wait for maximum message delivery delay. However, for replica control—as for real-time systems in general—it is a very desirable property to have a small temporal uncertainty. If the temporal uncertainty is small, the $\varepsilon$ of epsilon common knowledge is also small. Or, in other words, the replicas within a group have diverging states only for a short period [KK90]. It follows therefore that real-time communication services are best supported by approximately synchronized clocks [LM85, KO87].

The requirements for real-time replica control on communication services can be summarized as: The availability of a reliable broadcast service with total and temporal order (which also guarantees solution of the consensus problem) that is fast, efficient in the number of messages and which has a short temporal uncertainty. This kind of protocol efficiency is best achieved by using synchronous protocols which exploit the broadcast properties of broadcast channels and by justifying benign failure assumptions. An example of such a highly efficient protocol is TTP [KG94]. In the remainder of this chapter the communication service is assumed to be reliable broadcast with total and temporal order, if not stated otherwise.

## 5.4 Synchronization

The synchronization between servers in a replicated group can be used as a criterion to classify replica control. This classification has two aspects: the degree of synchronization may be considered in the value domain and the time domain. Starting with the value domain there are methods such as *epsilon-serializability* [WYP91] to control the amount of inconsistency [BG90] or asynchrony among a replica group. This approach allows inconsistency among servers, but guarantees that they will eventually converge to a consistent state. In the context of database and transaction systems controlled inconsistency has the advantage of higher parallelism and availability [SK92]. For real-time systems, which heavily rely on consistency and where system states are short-lived [Kop86], this asynchronous approach in the value domain is impractical.

Another approach to control the amount of inconsistency in the value domain is the similarity based data access protocol as described in [KM92, KM93]. The idea behind this protocol is to exploit the fact that real-time systems are typically based on a periodical processing paradigm, i.e. for many applications it is sufficient to access data on a periodic basis without using read/write-locks to guarantee serializability. The concept of similarity thus allows different but sufficiently timely data to be used in a computation without adversely affecting the outcome. While this relaxation of synchrony is effective for real-time data access in multi-processor systems, it is problematic in the context of replicated systems. In non-identically replicated systems this relaxation of synchrony may introduce additional non-deterministic behavior because individual servers in a group may access similar but different data, such that the service responses are non-deterministic. To attain replica determinism it is necessary in the general case that each replicated server accesses exactly the same version of data. It is therefore assumed in the following that although there may be weaker correctness criteria, all replicated servers access the same data and the access is atomic. Hence this approach to data access is considered to be synchronous. For many real-time systems even strict serializability is no problem because the amount of data is usually small. Since the reaction period of a real-time system depends on the duration that is required to gain consistent information, in the following section only synchronous approaches to replica determinism in the value domain are considered.

In the time domain there are two different approaches to synchronization: The first is virtual synchrony [BJ87a] and extended virtual synchrony [MAM$^+$93]. This approach is based on logical clocks as introduced by Lamport in his seminal paper on the ordering of events in distributed systems [Lam78b]. The second approach to synchronization in the domain of time is real-time synchrony. This approach is based on approximately synchronized clocks [LM85, KO87] which are used to introduce a sparse time base [Kop92].

### 5.4.1 Virtual synchrony

Virtual synchrony is based on the relaxation of common knowledge to concurrent common knowledge. This corresponds to a relaxation of replica determinism to eventual replica determinism. In a system with virtual synchrony, see Figure 5-5, all members in a group observe the same events in the same order. This applies not just to service requests, but also to events such as failures, recoveries, group membership changes and others. All events are ordered relative to some system internal logical time and precedence. This system internal logical time, however, need not have any resemblance to real time and thus allows diverging relative processing speeds of individual servers. As the example in Figure 5-5 indicates, the advantage of virtual synchrony is the ability to resemble a closely synchronized system by an asynchronous system with little delay and overhead under the restricting assumptions that only processor internal events are observed and that instead of total order causal order is used. Each server in a group can proceed at its individual processing speed as long as the ordering of events is consistent with its group members. This allows the efficient implementation of server groups on top of asynchronous processors and communication, as demonstrated by the ISIS tool kit [BJ87a, Bir93] and Totem [AMM+93].

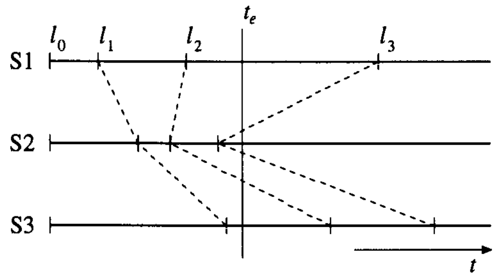

*Figure 5-5: Logical clocks*

While virtual synchrony is well suited for commercial applications or fault-tolerant database systems, it is not well suited for real-time systems. This is due to the fact that events are not related to real-time. Since the concept of virtual synchrony allows arbitrary difference between logical time and real time, that individual servers in a group may have different logical times and processing states at a certain point in real-time. Events occurring in real-time therefore have to be reordered to be consistent with logical time. The example given in Figure 5-5 shows an external event $e$ at time $t_e$, the logical times of server 1 to 3 are $l_2$, $l_3$ and $l_1$, respectively. To keep order consistent, the event $e$ has to be delayed at server 1 and 3 until logical time $l_3$ has passed. Due to the pipelining effect of virtual synchrony, considerable delays are introduced if events occurring in real-time are considered. A similar problem arises if total order is considered. Formally, total order requires that if two correct servers S1 and S2 observe events $e_1$ and $e_2$, then S1 orders $e_1$ before $e_2$ if and only if S2 orders $e_1$ before $e_2$. To achieve total order a server needs to know when its ordering of events becomes stable. This stability requires the knowledge that there are no more pending events which have been observed by other servers and would therefore require a reordering. To achieve total order a stronger state of agreement has to be achieved which requires all servers among a group to agree on a consistent cut which includes the real-time event. Hence, all servers in the group have to wait for the slowest server to synchronize, which can cause considerable delays in an asynchronous system.[27] The advantage of virtual synchrony—less delay and overhead due to synchronization—is therefore traded for the strong disadvantage that events are not related to real-time. Virtual synchronization is therefore not suited for real-time systems.

### 5.4.2 Real-time synchrony

Real-time synchrony is based on the relaxation of common knowledge to epsilon common knowledge. This corresponds to a relaxation of replica determinism where the inconsistency is bounded by an a priori known time interval. Compared to eventual replica determinism, as with virtual synchrony, this relaxation is much better suited for real-time systems. Real-time synchrony is based on a discrete time grid, see Figure 5-6.

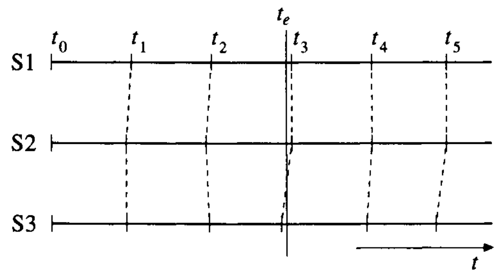

*Figure 5-6: Real-time clocks*

[27]In fact total order cannot be achieved in any purely asynchronous systems [FLP82] unless the asynchrony is relaxed to partial asynchrony [DLS88] or failure detectors are used [CT91].

The deviation among individual clocks is bounded by some fixed interval, called *precision*. The limits to time measurement in a distributed real-time system can be used to establish criteria for the selection of lattice points. Especially, the duration of a single clock tick should be longer than the precision. This allows the introduction of a discrete (or sparse) time base where time is treated as a discrete quantity. By restricting relevant event occurrences to the lattice points of the globally synchronized time base, it is possible to achieve a simulation of common knowledge where agreement on order and timing of events is guaranteed implicitly [Kop92]. But for external events it is impossible to restrict these events to the lattice points of the time grid since event occurrence is not in the computer system's sphere of control. Instead of reordering external events—as with virtual synchronization—the temporal uncertainty of one clock-tick (as for any discrete system) needs to be handled. Since the granularity of the approximately synchronized clock is usually in the range of microseconds, the resulting uncertainty is acceptable for many application areas.[28]

If, however, it is unacceptable that time stamps of replicas are off by one tick, an agreement protocol has to be carried out. Due to the result of this protocol, external events have to be delayed by at most one clock-tick, compared to the arbitrary time delay with logical clocks. Hence with real-time clocks it is possible to respond to external events within a bounded action delay. The action delay is defined by the granularity of the real-time clocks, since an external event stabilizes within one clock-tick (after execution of the agreement protocol), i.e. within one clock-tick it is guaranteed that there is no earlier event underway. The establishment of total order is therefore much simpler, compared to logical clocks.

Another important advantage of real-time clocks is the fact that clock readings of individual servers are comparable in the domain of time (with the temporal uncertainty of one clock-tick), while this is impossible for logical clocks. Clearly this synchronization method is better suited for real-time systems than virtual synchrony because it reflects the interaction between computer system and environment under real-time constraints.

The degree of synchronism in the time domain ranges from a very high degree of synchronism with fine granularity to asynchronous approaches with coarser granularity. The following subsections classify replica control strategies according to their degree of synchronism in the time domain, starting with the tightest synchronization.

[28]While the temporal uncertainty of event occurrences is bounded with real-time clocks, there is no such bound for logical clocks. This is the reason why reordering of external events with logical clocks may introduce arbitrary action delays on external events.

### 5.4.3 Lock-step execution

With lock-step execution, synchronization typically takes place at the hardware level. Every output of a single operation is compared among the replicated group. This type of server group synchronization is typically implemented by pairs of identical hardware processors. Examples are Stratus [TW89], Sequoia [Ber88] and DEC's VAXft 3000. Another example are the microprocessor V60 [NEC86] and more recently Intel's P6 [Ben95], which can operate either as master or as checker in a lock-step execution group. The main advantage of lock-step execution is the ability to execute arbitrary services which need not consider the problem of replica determinism. Because the individual processors are driven by a common clock source, they perform their operations almost simultaneously. This is especially important for commercial systems since readily available software may be used without modifications. To use application software for single processors by a replicated group of processors, all inputs to the system have to be identical. This implies that—besides the clock—other external inputs to the servers also have to be identical. Hence, all the inputs to the group are central. Only the processors are replicated.

The main disadvantage of lock-step synchronization is that the whole group relies on a common clock service and on common input devices. Hence, lock-step execution has to be considered as a strictly centralized approach. Furthermore, transparency of replication is traded for a reduced tolerance of transient faults. Faults can affect the server group as a whole by either affecting the common clock service or by affecting the replicated group of processors at the same point in computation [KKG+90]. The basic assumption that faults affect servers independently may no longer be justified. Another disadvantage is that the high clock speed and restricted fan-out are limiting the number of processors in a group. In addition, the physical distance between processors is limited by the clock speed. As a consequence, lock-step execution at the hardware level may not be considered as a truly distributed approach to replication and fault-tolerance.

### 5.4.4 Active replication

Using this technique all servers within the group execute the same service requests in parallel. Typically, all servers generate outputs. The replicated services are completely distributed, the only common resource is the communication media for message passing. Examples are CIRCUS [Coo84], Clouds [ADLW87], Delta-4's active replication [CPR+92], MAFT [KTW+88], MARS [KDK+89] and SIFT [WLG+78]. The "State Machine Approach", as described by Schneider [Sch90], treats this replication method in a very detailed manner. Active replication is typically implemented by executing software servers on different processors. Since all replicas are processing a functionally equivalent service in parallel, synchronism is relatively tight. The functional equivalence of services also implies that non-deterministic decisions have to be resolved by a distributed agreement protocol. This is because individual replicas take their decisions independently. Active replication is therefore strictly distributed or symmetric. Compared to lock-step execution there is no restriction on the physical distance and number of replicas,[29] since information exchange is accomplished by means of messages.

The main advantage of active replication over other methods is that individual servers are not restricted in their failure semantics because there is no single point of failure. All decisions are carried out in a distributed manner. This also causes the major disadvantage of this method: It requires the highest degree of replica control. Because all services are strictly distributed and execute service requests in parallel, the problem of replica non-deterministic behavior occurs most frequently. All the atomic sources for replica non-determinism are active with this synchronization strategy. In particular, this requires an agreement protocol to handle the replica non-determinism caused by the real world abstraction limitation. In addition, servers may use only deterministic functions. Non-deterministic functions also have to be resolved by an agreement protocol. This effectively rules out dynamic scheduling decisions and task preemptions, although these mechanisms are commonly used for most commercial real-time operating systems, e.g., [Rea86, Sof91]. The reason is the unacceptably high overhead to perform an agreement protocol on each scheduling and preemption decision. These restrictions are a severe limitation to many fault-tolerant real-time systems. To guarantee a short latency period, it is often necessary to take dynamic scheduling decisions and preempt executing tasks. The high effort for replica control and the restriction in the functions which have to be used to implement a service are the main disadvantages of active replication.

## 5.4.5 Semi-active replication

While the servers in the previous approach are symmetric, servers with semi-active replication are asymmetric in the sense that one server within the group is distinguished. This distinguished or central server takes all non-deterministic decisions. Examples of this technique are Delta-4 XPA [BHB+90, Pow91c] and ISIS [BJRA84, BJ87a, Bir93]. Within Delta-4 the distinguished server is called *leader*, the remaining servers in the group are called *followers*; within ISIS the wording *coordinate-cohort* is used. Semi-active replication allows the leader to execute non-deterministic functions. However, follower replicas have to be informed of the leader's decision. For example, it is not necessary to keep message order consistent for a server group. Rather the leader selects a message to serve next and communicates his selection to the follower servers. Deterministic operation sequences can be executed by the leader

[29]For very high speed systems the speed of light is a limiting factor.

and the followers concurrently. Outputs may be generated by the leader only or by the whole group.

Compared to lock-step execution and active replication, synchronization is coarser. Follower servers lag systematically behind because they have to await the leader's decision on non-deterministic functions. The advantage of this method is that non-deterministic functions are allowed without the need to carry out a consensus protocol. For example, task preemption may be allowed if preemption points are inserted in the software and the leader informs his followers to take the same preemption point [Pow91c]. The advantage of semi-active replication lies therefore in the lower complexity of the communication protocol. With active replication non-deterministic decisions have to be resolved by a consensus protocol while an information dissemination protocol, such as reliable broadcast, is sufficient for semi-active replication.[30] But even though the communication complexity is reduced with semi-active replication, the communication overhead for non-deterministic decisions may still be unacceptable for many application areas. In the above given example of task preemptions, the time that is required for communication will be a limiting factor for the number of preemptions per time. For systems where the communication speed is low, compared to processing speed, this solution to task preemption will be unsuitable. The disadvantage of semi-active replication is the fact that decisions, taken by the leader, are single points of failures. Furthermore, the failure semantics of servers is restricted since the follower servers cannot detect faulty decisions of the leader. Typically, servers are assumed to exhibit crash failure semantics.

### 5.4.6 Passive replication

Compared to the above presented approaches, for passive replication only one server among the group is active. The active server is called the *primary* and is responsible for generating outputs. The other servers are in *standby*, i.e. they are performing no service functions and are generating no outputs [AD76]. Although the standby servers perform no service functions, they have to perform housekeeping actions to remain synchronized with the primary server. Passive replication is alternatively called *primary-standby* technique. Typical application examples are transaction oriented commercial systems, e.g. Tandem GUARDIAN [BBC$^+$90]. In case of failures of the primary, some standby server has to take over its role. To do so, the standby servers have not only to store the last service request, but they have to be regularly updated with the primary's internal service state. This operation is called *checkpointing* [SLL86, KT87] and requires the standby servers to store snapshots of the primary's service state before each service request. An alternative approach consists of periodic updates with the primary's service state and an additional history of the recent

[30]See the preceding section "Communication" for a comparison of the communication complexity of consensus and reliable broadcast protocols.

service requests since the last checkpoint. Because the standby servers are never in synchrony with the primary, this approach is called asynchronous. If, however, the primary fails, one standby server has to start execution from the last checkpoint and synchronize to the last failure free state of the failed primary.

For real-time systems the duration of this resynchronization or failover period has to be bounded and considered in the operational timing behavior. The maximum duration for resynchronization from the last checkpoint to the actual service request is a critical factor. For real-time systems the maximum interval between two consecutive checkpoints has to be kept short. Hence a compromise between the communication overhead for checkpointing and the resynchronization period has to be made. Compared to all other replication methodologies this method requires the least processing resources, since standby replicas have to perform only house-keeping actions. This enables a backup server to perform additional services with its spare processing capacity. If the standby has to take over as primary, less critical services can be abandoned or relocated to another server. Passive replication—as well as semi-active replication—requires the primary to be fail restrained because standby servers may not detect failures of the primary (except by timeouts). While active and semi-active replicated servers exhibit group failure masking semantics, passive replication can either have hierarchical or group failure masking semantics. In the case of hierarchical failure masking service requests are sent to the primary only. If the primary crashes, the service request has to be repeated by the service user, but the destination is now the standby server which has become the new primary. In the case of group failure masking, service requests are broadcasted to the primary and its standby servers. Standby servers can now detect failures of the primary and the new primary can respond to the last service request without intervention of the service user.

## 5.5 Failures and replication

There is a close interdependence between the replication strategy and possible failure semantics of servers. The degree of centralism or distributedness is a determining factor for both. Under any central approach the leading server is required to be fail restrained. Because the remaining servers in the group follow the leader or primary, they cannot detect performance or byzantine failures. However, crash or omission failures of the leader may well be detected by guarding the leading server with timeouts. This is typically implemented by forcing the leader to send periodic "heartbeat" or "I am alive" messages. If a follower or standby server detects that the heartbeat message times out, it knows that the leader has failed. To employ this fault detection method, the servers are required to have omission, crash or fail-stop failure semantics. Early service responses or failures of the leader in the value domain cannot be detected by this method. However, a further improvement of this strategy is possible by checking whether a particular decision taken by the leader is among a set of valid decisions. This enables follower servers to detect some failures in the value domain. By checking the leader's decisions, it is possible to reduce the window of vulnerability for single point failures, but a small remaining window of vulnerability is inherent to any centralized approach. Hence, semi-active and passive replication strategies require servers to exhibit crash or omission failure semantics. Tolerating $t$ crash or omission failures requires at least $t + 1$ servers under the central approach. A typical implementation is to use fail-stop processors [SS83, Sch84] and take the output of any server that has not failed. For example the output of the fastest server may be taken. This is sufficient for service requests used inside and outside a group. It is interesting to note that the $t + 1$ bound holds only if a reliable broadcast communication service is used to send the service requests to all servers within a group. For passive replication with hierarchical failure masking, service requests are only sent to the primary. It has been shown for this assumption that the $t + 1$ bound does not hold. If the communication service is point to point, at least $2t + 1$ servers are required to tolerate omission failures [BMST92a, BMST92b].

In contrast, the distributed approach poses no restriction on the failure semantics of individual servers within a group. There is no single server which the remaining servers in a group depend on because all decisions are carried out by all servers. It is therefore possible to build groups out of servers which exhibit byzantine behavior. There is no window of vulnerability where a single failure may cause the whole group to fail. The advantage of no single point failures is traded for a higher communication complexity. Group internal non-deterministic decisions have to be resolved by execution of an agreement protocol, e.g. a distributed consensus protocol.[31] To achieve agreement among group members in the presence of up to $t$ performance failures $2t + 1$ servers are required. Under a byzantine failure assumption $3t + 1$ servers are required. In contrast to the central approach it is not sufficient with a performance or byzantine failure assumption to take any single service request or service. For example, the first service request that arrives may be too early or it may be faulty in the value domain. Instead, each server has to wait for at least $t + 1$ corresponding service requests or services responses from different servers to tolerate $t$ failures. Under a crash or omission failure assumption at least $t + 1$ servers are required to tolerate $t$ failures, which is identical to the central approach. In this case it is also sufficient to take the service request or service response of one server.

After occurrence of a failure (that is covered by the fault hypothesis), the server group has to perform some recovery[32] strategy. The kind of recovery strategy is closely related to the replication strategy, especially to the degree of centralism or

[31]Also see section "Communication" in this chapter.

[32]In this book the term *recovery* is used only for actions that are required to mask the actual fault and return to a fault free state. Actions that are required to keep the minimum required replication level, such as reintegration of repaired servers, are termed *redundancy preservation* and are treated in the next section.

distributedness. Under the central approach, failures of followers require no recovery actions. Because servers are required to have crash or omission failure semantics, failing follower servers are masked transparently. If a leader server fails under the central approach, the followers have to take recovery activities. At first a new leader has to be elected. The new leader can either be elected by some previously established ranking [CDD90] or at runtime [GM82]. After its election, the new leader has to take over the service of the failed leader. Dependent on the replication method different strategies are used to take over the leaders service.

In the case of passive replication, backward or rollback recovery [LA90, CW92] is used. This is the re-execution from the last known fault-free state. Typically, the new leader resynchronizes to the service state which was communicated by the last valid checkpoint information and replays all service requests which have been received since then. Under real-time constraints this re-execution time has to be considered for the operational behavior of a server group. If the failed server has to make outputs outside the group, another problem that is inherent to backward recovery has to be considered: There is a window of uncertainty where it is impossible for the re-executing server to know whether the last output was successful or if the server has crashed before. As a consequence, output devices are required to have *status* semantics to guarantee consistent re-execution or idempotence of service requests. Status semantics are typically *at least once* semantics opposed to event semantics which are *exactly once* semantics. Hence the re-executing server can make an output twice if it is uncertain whether the crashed leader has already made that output. However, compared to the implementation of status semantics, it is easier to implement output devices with event semantics. Furthermore, while it is possible to roll back the state of a computer system and start a re-execution this is not always the case for the computer systems environment. For example, it is not possible to "roll back" the state of a chemical process or to undo the fuel injection in a combustion engine. Since real-time systems are typically interacting with some physical process, this is the reason why roll back recovery is not suited for fault-tolerant real-time systems.

In the case of semi-active replication a forward recovery strategy can be used [BHB$^+$90, Pow91c]. Because all servers in a group receive service request, the follower servers know about all service requests. Furthermore, the service state of the follower servers is up to date until the last non-deterministic decision which was awaited to be taken by the leader. Hence, the new leader can use forward recovery by starting with this decision as the new leader. The timing overhead for forward recovery is much smaller compared to backward recovery. If only the leader server generates outputs outside the group, then, as with backward recovery, there is a window of uncertainty whether an output has been made or not. If all servers in the group generate outputs, then, similar to distributed replication, there is no window of uncertainty.

Distributed replication on the other hand does not require recovery actions. Hence, there is no temporal overhead for recovery. Services are carried out by more than one server in parallel. No server has to take over a service from a failed server since all servers in a group perform the same service. This symmetry of services requires no election which is an advantage compared to the central approach. Yet, a correct solution of the membership problem is very important for the distributed approach. Because failures as well as non-deterministic decisions are resolved in the group by means of a distributed agreement protocol, it is important to know which members are the correct constituents of a group. This problem is less important with central replica control because the leader's decisions do not depend on the results of other servers in the group. Also, the strategy for outputs outside the group or subsystem differs for the distributed approach. Output devices have to consider the results of the whole group or each group member has its own output device. In either case, or for any mixture of both cases, it is irrelevant whether a single server has successfully performed an output or has crashed. Thus, there is no window of uncertainty whether an output has been performed. Output devices are therefore not constrained to have status semantics such as at least once semantics. However, if outputs are made to the real environment, e.g. to actuators, then these actuators have to implement a proper voting scheme. For example, three hydraulic devices may be used to control the position of some lever. If one output is faulty, the power of the other two hydraulic devices can overrule the faulty one which implements a majority vote.

## 5.6 Redundancy preservation

The fault-tolerance of a server group is defined by the maximum number of servers that can fail while providing a correct service. To guarantee a certain degree of reliability, depending on the failure rates of servers and the mission duration, the level of redundancy has to be maintained above a given threshold in a replicated system. Redundancy preservation is thus the task of guaranteeing a certain replication level in spite of server failures. In the following we assume there are $n$ servers in a group, of which $f$ are actually faulty and up to $t$ failures have to be tolerated. The combining condition defines the minimum number of non-faulty servers within a group to guarantee a correct service. For the various failure semantics the following combining conditions have to be guaranteed: For distributed replica control and byzantine failures $n - f > 2t$, for performance failures $n - f > t$ and for crash or omission failures $n - f > 0$. If the combining condition is violated, the group cannot proceed with its service, and instead some servers have to be added to the group. The reconfiguration service has to decide whether there are spare servers that can be integrated. If there are no spare servers available, either less critical services have to be aborted to gain capacity, or the service provided by the server group with the violated combining condition has to be abandoned completely.

For real-time systems it is important to carry out the redundancy preservation strategy in a timely manner. If a group has to halt its service due to a violated combining condition, the duration until new servers are integrated into the group has to be considered for the operational phase. To avoid a group having to halt, the redundancy preservation strategy should add spare servers to a group in advance. Another important aspect which has to be considered for the redundancy preservation strategy are failures of the reconfiguration service. A faulty reconfigurator may remove correct servers or add faulty servers to a group. Therefore, a faulty reconfigurator has the same effect as if its controlled servers were all faulty. Thus $f$ is actually the number of faulty servers plus the number of correct servers that is configured by a faulty reconfiguration service. To avoid single point failures and to achieve a higher degree of fault-tolerance, it is therefore convenient to replicate the reconfiguration service itself.

The underlying necessity for redundancy preservation is the existence of a reliable membership service. To guarantee replica determinism the membership service has to detect departures or joins of servers and must make consistent group membership information system-wide available. For real-time systems this consistency property requires not only agreement on group members, but more important, the delay between a membership change and the consistent reflection of this change in the global membership must be bounded by some a priori fixed time interval. A formal specification of a membership service and its temporal properties which is based on a synchronous system model is given by [Cri88]. Membership information with a bounded delay time may only be guaranteed in a synchronous system.[33] However, this is only possible under the assumption that there is no network partitioning. In asynchronous systems it is possible to build membership protocols which tolerate partition failures, but there is no guarantee for timeliness. Asynchronous membership protocols furthermore have a weak relation between failures occurring in real-time and system internal events. To achieve replica determinism internal and external events have to be reordered according to application specific causality relations. Asynchronous membership protocols are therefore not well suited for real-time systems. Examples of synchronous membership protocols are given by [Cri91a, KG94, Grü93], membership protocols based on asynchronous or partial asynchronous systems are described in [BJ87b, MM89, LP94].

Compared to memberships changes caused by server failures or voluntary departures, joins of servers are much more complicated to handle. The reason for this is

[33]Since a membership service is typically implemented in software, the assumption of a synchronous system also implies guaranteed schedulabilty for hard deadlines which is necessary for a membership service with timeliness guarantees.

that replica determinism requires servers in a group to have corresponding outputs and service states. In case a server departs from a group, this membership change has to be detected and communicated in a consistent and timely manner. But for server joins this is not sufficient. Rather, before adding a server to a group, it has to be initialized to a state that corresponds to other group members. But the state of the other group members is steadily evolving over time. There is a mutual exclusive relation between state initialization of a new server and service progression in the remaining group, which requires group-wide synchronization. When a new server joins a group, it has to be guaranteed that all servers in the group have corresponding states, that no server in the group is actually processing any service request and that all pending service requests are delivered from now on to the new group that includes the joined server. For real-time systems this has the consequence that the temporal overhead of state initialization has to be considered in the operational phase. A consequent treatment of this matter can be found in [KGR89, Grü93, KG94].

# Chapter 6

# Replica determinism for automotive electronics

It has been argued in chapter 2 that systematic fault-tolerance has very desirable properties but requires replica determinism. This chapter is therefore concerned with the problem of replica determinism enforcement for fault-tolerant automotive electronic systems. However, the discussions in this chapter apply also to other fault-tolerant real-time applications where efficiency is important and a short latency period has to be guaranteed. The presented solutions for replica control in the presence of event-triggered service activation and preemptive scheduling have general applicability.

The first section gives a definition for services with a "short" latency period in the context of real-time systems. According to this definition the application area of automotive electronics, as presented in chapter 2, is one that demands a short latency period.

The next section compares the requirements for automotive electronics to known strategies for replica determinism enforcement. To evaluate the applicability of replica determinism enforcement strategies for automotive electronics the two important implementation choices event-triggered service activation and preemptive scheduling are motivated. The replication strategies active replication, semi-active replication and passive replication are compared against the given requirements. It will be shown that none of these strategies fulfills the requirements for short latency periods and high efficiency.

Section three develops a communication protocol that is optimal for replica control of external events with a minimum amount of information. A tight bound for the minimum amount of information exchange that is necessary to achieve agreement is presented. Two easy to implement functions are defined which can be used to achieve agreement with this minimum amount of information.

The next section describes a new methodology which ensures replica determinism for system internal information even if preemptive scheduling is used. This methodology—called timed messages—is based on a simulation of common knowledge to guarantee that replicated services can act as if preemptions are taken simultaneously. With timed messages it is possible to guarantee deterministic preemptions without communication between services. Different message semantics are discussed with regard to implementation considerations and possibilities for optimization.

The final section reevaluates the problem of replica determinism for automotive electronics. The new communication protocol for agreement with a minimum amount of information and timed messages are used for replica determinism enforcement. This reevaluation shows that replica determinism enforcement becomes possible.

## 6.1 Short latency period

It is a misconception about real-time systems to assume that these are systems which have to respond very fast [Sta88] within a short latency period. In fact, real-time systems are required to respond to relevant environmental state changes within a bounded latency period, rather than very fast. The requirement for a short latency period is thus a true additional requirement besides that of real-time. However, there is no exact or formal definitions for "short latency period". Usually, short latency period is used as an informal term which is associated with responding very fast. But since being fast is relative, in the following a notion of a short latency period is introduced that is suitable in the context of real-time systems.

Real-time systems are based on the assumption that there is a minimum interarrival period between relevant state changes.[34] Since the computer system has to respond to each relevant state change, this minimum interarrival period is necessary to bound the processor load that is required to service the relevant state changes [Pol95b]. The worst case latency period between a relevant state change in the environment and the response of the system is called *deadline*. The relevant environmental state changes therefore can be considered as requests to a service which has to deliver service responses. Given the minimum interarrival period between service requests and the deadlines associated with them, a short latency period can be defined as follows:

***Short latency period:*** A latency period is said to be "short" if the deadline that is associated with the service request is short, compared to the minimum interarrival period between consecutive service requests.

The shortest latency period according to this definition is given if the service's computation time is equal to its deadline. In this case there is no additional time left for enforcing replica determinism explicitly; it is impossible to wait for communication services or to execute agreement protocols on non-deterministic states. This extreme

[34]If the assumption coverage of the minimum interarrival period between state changes is insufficient, it is necessary for hard real-time systems to enforce the minimum interarrival period explicitly [Pol93].

case shows that the enforcement of replica determinism becomes the harder—if not impossible—the shorter the latency period. Therefore, replicated systems are currently applied in areas where no requirements for short latency periods exists. Examples are Delta-4 [BHB+90, CPR+92], ISIS [BJ87a, Bir93] and with restrictions MARS [KDK+89]. Delta-4 and ISIS are not well suited for short latency periods due to their asynchronous nature and the resulting overhead for communication protocols and synchronization. MARS, on the other hand, has very little overhead for synchronization and communication. It can therefore guarantee requirements for short latency periods. But due to the static preallocation of resources, MARS has a low resource utilization if it is required to respond to events with an a priori unknown timing within a short latency period [Pol95b].

Automotive electronics and especially control systems for combustion engines have requirements for short latency periods. (The characteristics of this application area together with its requirements have been presented in chapter 2). Besides automotive electronics there is a broad spectrum of applications that also have to respond within a short latency period. However, to avoid overly generic requirements which are either impossible to fulfill or which are too vague to uncover problems, in the following the requirements for automotive electronics have been selected.

## 6.2 Comparing requirements to related work

Many computer systems in the area of automotive electronics have high dependability requirements. For these systems it seems natural to improve dependability by systematic fault-tolerance and replication of critical services. However, to the best of the author's knowledge, up to now no system with short latency period requirements has been developed which uses replication in a systematic manner. One reason for this is the fact that current methodology for replication is not well suited for this requirement. This section compares known methodologies for replication against the requirement for a short latency period, as presented in the previous section.

To evaluate the applicability of known replication strategies for systems with short latency period requirements two important implementation choices first need discussing. The first choice concerns the type of service activation, i.e. whether to use event-triggered or time-triggered service activation. The second implementation choice concerns scheduling decisions, namely whether to use preemptive or non-preemptive scheduling. Under the above given requirements for short latency periods the choices will be strongly biased towards event-triggered service activation and preemptive scheduling. In the following a motivation for both choices will be given since they have strong impact on the applicability of different replication strategies.

### 6.2.1 Event- and time triggered service activation

The reason for using event-triggered activation instead of time-triggered activation to achieve a short latency period is a better resource utilization. By definition, a system has a short latency period if its services have deadlines that are short compared to the minimum interarrival period of service activations. The given requirements demand that some services are activated at certain points in time which are not known a priori. An example for this type of activation are services that have to be activated at certain crank angle positions. To use time-triggered activation exclusively, the task which implements these services needs to be transformed. Such a transformation of event-triggered tasks to time-triggered activation is given by [Mok83]. The transformed task has exactly the same service specification as the original task, but the required processor load is much higher. This relation between latency and the required processor load for event- and time-triggered task activation has been studied in [Pol95b]. The result shows that for time-triggered service activation the processor's resource utilization linearly approaches zero, the shorter the latency period becomes. Since a short latency period is a basic requirement, event-triggered activation should be supported. Note that the support for event-triggered activation does not preclude a system architecture with event- and time-triggered activation. There is rather a mixture of services, where some of them are best supported by event-triggered task activation while others are best supported by time-triggered task activation. Typically, crank angle or rotational speed related services are better supported by event-triggered activation while most of the remaining services are better supported by time-triggered task activation.

The problem of event-triggered activation with replicated systems is its inherently non-deterministic behavior which leads to inconsistent order. Since the activation time with event-triggered activation is under the control of the environment, the real world abstraction limitation applies, i.e. it cannot be guaranteed that an external event is observed by all servers at the same time. Hence, the ordering of events becomes inconsistent in a replicated group of servers. For example server S1 observes an external event $e_1$ at time $t$ while server S2 observes the same event at time $t + 1$. A system internal event $e_2$ occurs at time $t$ in both servers. The temporal order of events will be $e_1$ before $e_2$ in server S1 if ties are broken arbitrarily, and $e_2$ before $e_1$ in server S2. Hence, S1 will select a different order of service execution than server S2.

### 6.2.2 Preemptive scheduling

The second implementation choice is the selection of preemptive scheduling over non-preemptive scheduling. The motivation for preemptive scheduling is given by the broad spectrum of service activation frequencies. According to the requirements there are three orders of magnitude between the highest activation frequency of

10 *kHz* and the lowest activation frequency of 10 *Hz*. This gives a minimum interarrival period of 100 $\mu s$ for the most frequent service. In the case of non-preemptive scheduling it has to be guaranteed that the execution time of all tasks is well below the shortest deadline such that there is remaining computation time for these tasks. Hence, it follows that the maximum task execution time has to be considerably less than 100 $\mu s$. This would imply that all the services with low activation frequencies and larger execution times have to be implemented by task chains where each segment has a sufficiently low execution time. A lot of manual intervention would be required to build such task chains. This would make application programming a tedious process. Preemptive task scheduling should therefore be supported.

Just like event-triggered activation, preemptive scheduling also introduces nondeterministic behavior of servers. There are two possibilities which may lead to inconsistent ordering of schedules. Firstly, if scheduling decisions are based on external events, then possible inconsistent orderings of events can lead to different preemptions as described for event-triggered activation. The second possibility of nondeterministic behavior is caused by the real-world abstraction limitation and by nonidentical replication. Consider the following example: Two identical servers S1 and S2 have to execute the replicated services $T_1$ and $T_2$. But server S2 is additionally used for Service $T_3$ which is not replicated on server S1. This different set of services executed by the servers S1 and S2 can lead to replica non-deterministic behavior as Figure 6-1 shows. Service $T_1$ is requested at both servers, but since server S2 is already executing service $T_3$, service $T_1$ starts a little later at server S2. Some time later service $T_2$ is requested. Service $T_2$ has a higher priority than $T_1$ and hence preempts at server S2. Both servers execute service $T_2$ and finish at about the same time. Server S2 resumes the preempted service $T_1$ and finishes a short time later. But S1 has finished $T_1$ just before $T_2$ was requested. Hence, the order of service delivery is inconsistent. S1 first delivers the response of $T_1$ and then the response of $T_2$ while S2 has the opposite delivery order.

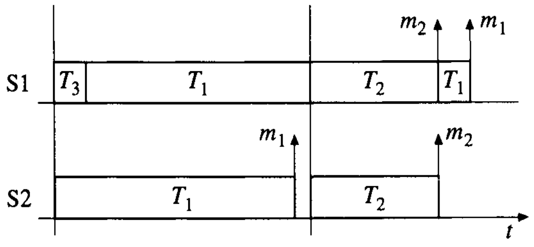

*Figure 6-1: Non-identical replicated services*

Replica non-determinism in the given example was caused by non-identical replication. This is one possibility. The other possibility for non-identical replication are servers with different processing speeds. But even if services and servers are replicated identically, replica non-determinism is introduced by the real world abstraction limitation with preemptive scheduling. The reason is that two servers (which are using replicated clocks) will always have slightly different processing speeds, i.e. since servers are never exactly in synchrony, it is impossible to guarantee that all preemptions occur at both servers at exactly the same instruction[35] cycle.

In the following the suitability of different replication strategies for automotive electronics is discussed. The replication strategy has to be feasible under the given requirements for a short latency period and should also support event-triggered task activation as well as preemptive scheduling. The succeeding subsections discuss active replication, semi-active replication and passive replication.

### 6.2.3 Active replication

With active replication all servers within the group execute the same service requests in parallel. If external inputs are to be handled by some services, then the real world abstraction limitation applies, i.e. the replicated services in the group may observe different external inputs. These differences occur either in the domain of value or in the domain of time. For event-triggered activation, which is an external input, this has the consequence that all servers have to agree on the timing of event occurrence. After the execution of an agreement protocol the agreed upon timing of the event can be used for scheduling decisions.

The same is true for preemptive scheduling with active replication. Before taking a preemption decision all servers in the group have to agree on the exact location where to preempt.[36] Slight differences in the processing speed would otherwise lead to inconsistent preemption points among services. Hence it follows that for active replication the latency is limited by the delay that is needed to attain agreement. If a service is activated by an external event, agreement has first to be achieved on the timing of the event. Secondly, agreement on the location where to preempt the currently executing service is required. The execution time of the agreement protocol limits the frequency of task activations in both cases. Furthermore, the requirement for hard real-time behavior can only be guaranteed if the maximum execution time for the agreement protocol can be bounded. The critical factor for all these requirements is therefore the maximum execution time of the agreement protocol.

[35]Lock-step execution is an exception since all servers are using the same clock service. But due to its limitations lock-step execution is not considered (see subsection "Lock-step execution").

[36]The typical solution to controlling preemption is the insertion of preemption points. These are regularly inserted code pieces that check whether to preempt or not. Preemption points results in a cooperative execution model rather than a truly preemptive one. For this subsection, however, it is assumed that it is possible to control preemption with a negligible preemption latency.

In the following, the maximum execution time for an agreement protocol with active replication is evaluated. This evaluation is based on an actual implementation [MLM+95] where timing parameters for the communication protocol and the operating systems have been derived. The system's fault hypothesis is that servers and communication links experience only omission failures. Servers are implemented by means of 16 *bit* microcontrollers. The lower level communication protocol is CAN [SAE92] which is a bit serial protocol that supports totally ordered atomic broadcast, the bus access strategy is CSMA/CA.[37] The CAN communication protocol [CAN85, SAE92] is based on the assumption of a broadcast degree $b = n$. A message is therefore transported to all servers in a group or to none of them, this allows implementation of atomic broadcast within one round.

Since the effort to agree on the timing of an external event and to agree on a preemption point, or to agree on both together is nearly the same, only the last case is treated. To handle an external event and preempt the currently executing service the agreement protocol has to be executed on behalf of the timing of the external event and on the point where to preempt the execution. Both agreement steps, however, can be packed into one communication message. The agreement protocol can proceed as follows: Each server transmits the timing of its event observation together with the next possible preemption point that can be taken. To tolerate omission failures it is furthermore assumed that messages are retransmitted. In a group of $n$ servers each server will receive $n - 1$ messages if no failures occur. Since the failure semantics are omission failures and CAN supports totally ordered atomic broadcast, it is sufficient that all servers select the same message. The selection must include the latest preemption point because the server which has broadcasted this preemption point cannot preempt earlier. The selection protocol therefore becomes a simple search to pick the largest out of $n$ values. The worst case end-to-end delay between occurrence of the external event and the agreed upon service preemption is given by Table 6-1.

| *action* | *time* [$\mu s$] |
|---|---:|
| maximum latency to react on external event | 20 |
| assemble packet and request transmission | 30 |
| media access control (MAC) latency (111 *bit*) | 444 |
| message transmission $(32 + 47)(r + 1)n$ *bit* | $316(r + 1)n$ |
| get received time stamp and preemption point | $30(n - 1)$ |
| selection of time stamp and/or preemption point | $3n$ |
| context switch to new service | 10 |
| total execution time | $316rn + 349n$ |

*Table 6-1: Agreement execution (active replication)*

[37]Carrier Sensed Multiple Access with Collision Avoidance (CSMA/CA).

The first step of the agreement protocol is to respond to the arrival of the external event. Under the assumption that the event has highest priority a maximum latency of 20 $\mu s$ is guaranteed by the operating system. The next step is to assemble a message that contains the time stamp of the external event and the next preemption point. Message assembly and transfer to the communication controller, which is responsible for transmission, requires 30 $\mu s$. It is assumed that the preemption messages have higher priorities than any other message on the communication bus. This results in a communication media access latency of 111 *bit* at 250 *kbps* = 444 $\mu s$ (this latency time is determined by maximum length of a message which is 111 *bit*). It is assumed that the information for the time stamp as well as for the next preemption point require 16 bit, respectively. The message length with the CAN protocol is 32 *bit* data + 47 *bit* overhead. Since there are $n$ servers and each may retry to send $r$ times, there are $(r + 1)n$ messages with a length of 79 *bits* to send at 250 *kbps*. In the next step the received messages have to be read from the communication controller. Afterwards the message with the latest preemption point has to be selected. This requires a linear search through the proposed preemption points of all received messages. The next step is to switch the context at the agreed upon preemption point to the service which was requested by the external event. For a group with two replicas ($n = 2$) and one retry ($r = 1$) an end-to-end agreement protocol execution time of $pt^{ar} = 316rn + 349n + 474 = 1804\ \mu s$ results. With three servers the total execution time goes up to $pt^{ar} = 2469\ \mu s$.

It is obvious that the performance of active replication does not meet the requirements by far. The system was required to respond within 100 $\mu s$, activation frequencies were required up to 10 *kHz*. Under the above made assumptions, the shortest possible latency period with active replication is 1804 $\mu s$. This is more than one order of magnitude too slow. If services do not have the highest priority, this delay becomes even larger. The reason is the large delay due to the latency and overhead for the communication protocol. Splitting the execution time formula to a part which represents the execution time of the servers and one part that represents the communication time gives $pt^{ar}_{server} = 33n + 30$ and $pt^{ar}_{comm} = 316n(r + 1) + 444$, respectively. Under the assumptions $n = 2$ and $r = 1$ the service execution time is $pt^{ar}_{server} = 96\ \mu s$ and the communication time is $pt^{ar}_{comm} = 1708\ \mu s$. This shows that the servers execution is within the requirements while the communication time is unacceptable. Hence, it follows that active replication is not feasible under the given requirements and assumptions due to the large communication delay.

### 6.2.4 Semi-active replication

Semi-active replication is a central approach where servers within a group act in parallel as long as actions are deterministic. All non-deterministic actions are taken by the leader while the remaining servers in the group await the leader's decisions. Since event-triggered service activation is non-deterministic, all the servers have to agree on the timing of event occurrence. With semi-active replication the leader server determines the timing of the event occurrence and communicates its observation to all follower servers in the group. Compared to the active replication there is no agreement protocol necessary and furthermore only one server needs to broadcast its observation. The same is true for service preemption. If a preemption is necessary, the leader server decides where to preempt and communicates its selection of a preemption point to all followers. This requires that the follower servers systematically lag behind the leader to guarantee that the leader's decisions on preemption points does not arrive late. The execution of the follower servers therefore needs to be delayed for at least the maximum end-to-end delay of the preemption point communication between the leader and its followers. Semi-active replication was explicitly designed to handle preemptive scheduling and event-triggered activation [BHB+90, Pow91c]. To control the location of preemptions it is proposed to insert preemption points into the source code by tool support [Pow91c].

To evaluate the suitability of semi-active replication for fulfilling the given requirements, again, the maximum end-to-end execution time of the information dissemination protocol has to be considered. This delay determines the latency of follower servers. The leader server, however, can take its decision immediately and does not need to wait, i.e. the leader's latency period is much shorter compared to the followers. But still, it has to be guaranteed that the total frequency of non-deterministic decisions can be communicated. This requires that the minimum interarrival period between two non-deterministic decisions taken by the leader needs to be larger than the maximum execution time of the information dissemination protocol. The maximum protocol execution time therefore determines the latency of servers and the maximum frequency at which non-deterministic decisions may be taken.

In the following, the maximum protocol execution time is evaluated under the same assumptions as with active replication. Again, the case of event-triggered service activation and preemption will be described. Upon arrival of the external event the leader has to interrupt its currently executing service. The next action is to broadcast the timing of the event observation and the next possible preemption point to all followers. Upon arrival of that message the followers have to take over the timing of the event observation and the next preemption point. Afterwards they continue their service until the selected preemption is reached, where the actual service is preempted and the event-triggered service is activated. The worst case end-to-end delay between occurrence of the external event and the agreed upon service preemption is given by Table 6-2. The first action of the leader is to respond to the arrival of the external event. It is assumed that the external event has the highest priority among services, the worst case latency is 20 $\mu s$ in this case. In the next step the leader assembles a message containing the time-stamp and next preemption point for the followers. Including the time required to request transmission this action needs another 30 $\mu s$. It is also assumed that the message has the highest priority on the communication bus.

| *action* | *time* [$\mu s$] |
|---|---|
| maximum latency to react on external event | 20 |
| assemble packet and request transmission | 30 |
| media access control (MAC) latency (111 *bit*) | 444 |
| message transmission $(32+47)(r+1)$ *bit* | $316(r+1)$ |
| maximum latency to react to message | 20 |
| get received time stamp and preemption point | 30 |
| maximum preemption point interval | 50 |
| context switch to new service | 10 |
| total execution time | $316r+920$ |

*Table 6-2: Information dissemination (semi-active replication)*

The resulting communication media access delay for the CAN protocol is therefore 111 *bit*, which results in a delay of 444 $\mu s$ at 250 *kbps*. With semi-active replication only the leader needs to broadcast the messages. Under the assumption of 32 *bit* message data and up to at the most $r$ retries the total number of bits is 79 $(r + 1)$. At 250 *kbps* this results in a maximum message transmission delay of $316(r + 1)$ $\mu s$. In the next step the follower servers have to react to the leader's message. In contrast to active replication where all serve's process the external event, the follower servers do not process external events until they are received from the leader. Hence, the maximum latency to react to the leader's message has to be considered. This latency is 20 $\mu s$ since it is assumed that reception of leader messages has highest priority. Following this, the time stamp has to be read from the received message and the requested preemption point has to be passed to the operating system. This requires 30 $\mu s$. The delay to take the next preemption point is defined by the maximum time interval between two consecutive preemption points. It is assumed that the maximum interval between two consecutive preemption points is at the most 50 $\mu s$ (which is optimistic). The last step is to switch the context at the received preemption point to the service which was requested by the external event. In groups of arbitrary size with one send retry ($r = 1$) the end-to-end protocol execution time is $pt^{sar} = 316 + 940 = 1236$ $\mu s$. With two message send retries the protocol execution time goes up to $pt^{sar} = 316\ 2 + 940 = 1552$ $\mu s$.

Again the resulting end-to-end protocol execution time does not meet the established requirements for short latency periods. The resulting 1236 $\mu s$ delay for semi-active replication is far too long to match the required latency period of 100 $\mu s$ and the activation frequency of 10 *kHz*. Compared to active replication, semi-active replication has the advantage that the protocol execution time is independent of the replica group size. Hence semi-active replication scales better. Splitting the total protocol to the execution time for servers and the execution time for communication results in $pt^{sar}_{server} = 160\ \mu s$ and $pt^{sar}_{comm} = 316r + 760\ \mu s$. The server's execution time is fixed, independent of the group size and the number of retries. Under the assumption of one retry ($r = 1$) the communication time is $pt^{sar}_{comm} = 316\ r + 760\ \mu s = 1076\ \mu s$. Compared to active replication the communication time is reduced while the server execution time has increased. With semi-active replication the server execution time of $160\ \mu s$ is alone unacceptable. Furthermore, the recovery time to handle failures of the leader is not considered. It follows that semi-active replication is not feasible under the given requirements and assumptions.

### 6.2.5 Passive replication

With passive replication only one server in the group is active while the other servers are in standby. The active server or primary is responsible for service execution and communicates its service state together with service requests to the standby servers. Event-triggered service activations as well as preemptions are therefore handled by the primary exclusively. Compared to active and semi-active replication there is no agreement or information dissemination protocol necessary. The primary can respond immediately to external events and may preempt services arbitrarily.

There are two possibilities to update the standby servers with service states and service requests. The first possibility is to send the service request together with the service state to the standby servers upon each activation of a service. If services are preempted, the actual service state of the preempted service again needs to be sent to the standby servers. Using this approach the service states are always kept synchronous such that a standby server only needs to start the execution of the last service request. The second possibility is to send all service requests to the standby servers, but to defer sending of service states to periodic intervals. In this case the standby server needs to re-execute all service requests that have arrived since the reception of the last service state. The problem with the first approach is the required communication bandwidth, since the service state needs to be communicated at each service activation and preemption. The second approach reduces the required communication bandwidth but requires backward recovery which causes a significant delay for re-execution of services. Furthermore, backward recovery is inappropriate for many real-time systems. Consider, for example, a control loop: it does not make sense to redo the last calculations and outputs of setpoints. Rather, a forward recovery has to be executed to calculate and output the next setpoint. Whether backward or forward recovery is more suitable depends on the application semantics. In the following, it is assumed that a periodic service state update is sufficient and that proper recovery strategies are implemented.

To evaluate the suitability of passive replication for systems with a short latency period the amount of data and its repetition periods needs to be determined. As mentioned earlier, the latency period is no problem since the primary can act immediately. The problem is to communicate service state data and service requests to the standby servers. This requirement to communicate all service requests is different from active and semi-active replication where only non-deterministic decisions need to be communicated. The critical factor is therefore not the end-to-end communication delay but the rate of information that has to be communicated per time.

For the following estimation of the communication bandwidth it is assumed that service requests arrive with a rate of 10 *kHz* and that the average service request size is 8 *bytes* at that rate. This is an average value since services with lower activation frequencies tend to have larger service requests while services with higher activation frequencies usually have small service requests. Furthermore, it is assumed that the service state is sampled every 100 *ms*. The cumulative size of all service states that is stored within the primary server is assumed to be 4 *kbytes*. The resulting net communication bandwidth is given in Table 6-3. By considering that there is an overhead of about 50% with the CAN's communication protocol, the total required communication bandwidth will be 1936 *kbps*. Compared to the available communication bandwidth of 250 *kbps* the necessary communication rate is too large by a factor of 8. It is obvious that passive replication does not meet the requirements at the given communication bandwidth.

| *data (size and rate)* | *bandwidth* [*bps*] |
|---|---|
| service requests (8 *bytes* at 10 *kHz*) | $640\ 10^3$ |
| service states (4 *kbytes* at 10 *Hz*) | $328\ 10^3$ |
| net bandwidth | $968\ 10^3$ |

*Table 6-3: Communication bandwidth (passive replication)*

For completeness, besides the communication bandwidth the primary's execution time to send service requests and service states has to be estimated as well. The CAN protocol can transmit at the most 8 *bytes* within one message. The transport protocol layer, which is responsible for transfer of larger data amounts, packs 7 *bytes* of data into one message and uses the remaining one byte for sequencing information. The transport protocol's execution time for one segment of 7 *bytes* is 20 $\mu s$. Table 6-4 gives the resulting execution time for the primary.

| *action* | *time* [$\mu s$] |
|---|---|
| service requests (8 *bytes* at 10 *kHz*) | $228.57\ 10^3$ |
| service states (4 *kbytes* at 10 *Hz*) | $117.03\ 10^3$ |
| total execution time | $345.60\ 10^3$ |

*Table 6-4: Communication execution (passive replication)*

The total execution time to broadcast all service requests and the periodic service states updates is 345.60 *ms*. Since this is the execution time within one second, the resulting processor load is therefore 34.56%. The resulting primary's processor load is schedulable although it may be too high in many application areas. The standby servers execution time is not critical since they need not execute services other than receiving service requests and service states.

It follows that passive replication does not fulfill the given requirements for short latency periods either. While the required communication bandwidth is nearly one order of magnitude too high, the required server execution time is feasible. It should be noted, however, that passive replication has important drawbacks which precludes usage of passive replication in many application areas. The are two major points of concern: Firstly, the recovery delay is unacceptable for many real-time applications. Furthermore, the recovery strategy—whether to use forward or backward recovery—has to be implemented application specific which complicates the system design considerably. Secondly, passive replication is typically used in the context of data base systems where serializability is guaranteed as a correctness criterion. This is not the case with real-time systems. Many real-time systems relax serializability to weaker correctness criterias such as similarity [KM92]. This makes backward recovery problematic because the exact execution order with preemptions cannot be repeated. The different execution order will lead to different results for the service executions during recovery.

### 6.2.6 Replication strategy comparison

Of the above presented replication strategies neither active, semi-active or passive replication could satisfy the given requirements. The available communication bandwidth of 250 *kbps* in all three cases was insufficient to synchronize servers on behalf of the non-determinism introduced by event-triggered activation and preemption. With active and semi-active replication the latency period to react to external events and to preempt tasks is unacceptably delayed by the communication time. The problem with passive replication is the inability to communicate all service requests and service states over the limited bandwidth communication media. Furthermore passive replication was shown to be not generally applicable for real-time systems because of its problems with the recovery strategy and recovery delay.

In the following, various approaches to build a system that conforms to the requirements for automotive electronics are discussed. Since the communication bandwidth is the most critical parameter, it is necessary to optimize communication. With passive replication there is no possibility to reduce the communication bandwidth since even the required net bandwidth of approximately 1 *Mbps* is too large by a factor of 4. Also passive replication does not scale well since increases in the size of service requests and service states would raise the communication requirements even more. Passive replication therefore does not have to be considered any further. The remaining replication strategies are therefore active and semi-active replication. Both cannot fulfill the requirements for short latency periods because the communication on non-deterministic decisions introduces unacceptable delays. One reason is the high media access latency of the CAN protocol. Even if the message to be transmitted has the highest priority, the latency is 444 $\mu s$ at a communication speed of 250 *kbps*. Compared to the required latency period of 100 $\mu s$ this is far too long. To reduce the media access latency there are two possibilities. Firstly, one could make messages at the communication bus preemptable which would complicate the communication protocol. And secondly, one could use a special purpose communication bus for resolving non-determinism. The second possibility has a lower complexity compared to the first approach. Furthermore, by having two independent communication buses, one for regular communication and one for replica determinism control, there is additional communication bandwidth available.

Besides the communication media access latency the communication time for a single message is also too long. With CAN the overhead for a single message is 47 *bits*. At a communication speed of 250 *kbps* this requires a transmission time of 188 $\mu s$. Hence, it follows that the communication overhead alone takes more than the required maximum latency period of 100 $\mu s$. The maximum number of bits that can be sent within 100 $\mu s$ at a communication speed of 250 *kbps* is 25 *bits*. It follows that only a very small number of bits can be transferred at the available communication speed. For example, to send a 16 *bit* time stamp of an external event and 16 *bit* preemption point information would require 128 $\mu s$ alone, but this does not include any protocol overhead for framing and error detection. By considering the additional server execution time, the impossibility of transferring 32 *bit* becomes obvious. It is therefore necessary to reduce the number of bits that have to be exchanged to the absolute minimum (although, some redundancy is required to detect message corruption and to support the fault hypothesis).

With the above presented approaches to replica control there were two sources of non-determinism which required communication. Firstly, event-triggered service activations and secondly service preemptions. For event-triggered activation the real world abstraction limitation applies. Hence, it follows that it is impossible to avoid communication for replica determinism enforcement. For service preemption the real world abstraction does not necessarily apply. The real world abstraction limitation only applies if system external events are used for preemption. If, however, preemption decisions are based on system internal events exclusively, it is possible to avoid the real world abstraction limitation. By using a simulation of common knowledge, service preemption can be made deterministic. Under the assumption that there are far more service preemptions than event-triggered service activations, it is possible to reduce the communication time by sending information on external events but not on preemption decisions. An event-triggered service activation can then be implemented in two steps. Firstly, the servers in the group have to agree on external events. By this agreement the event becomes system internal. In the second step servers can base their preemption decisions on the agreed external event and hence need no further replica control by exploiting simulated common knowledge. This approach can be implemented either by active or by semi-active replication. With active replication all servers have to exchange information about the timing of external events. For semi-active replication it is sufficient that only the leader server sends the timing of the external event.

The next section describes a protocol that is optimized for replica control of external events with a minimum number of bits. After this section the following section describes a possible simulation of common knowledge to achieve deterministic service preemptions.

## 6.3 Optimal agreement for replicated sensors

This section describes a protocol to enforce replica determinism on external events or sensor readings with a minimum amount of information exchange. The protocol can be applied to achieve determinism in the value domain, in the time domain or both. The agreement problem for external events is defined by (1) a set of replicated sensors which measure some continuos state variable of a physical process, (2) a set of servers which perform control functions based on these sensor inputs where (3) the results of fault-free servers are required to be identical. For that reason an agreement protocol is required to ensure that replicated servers get identical input values. It is the aim of this protocol to exchange information on the individual sensor readings and present one agreed upon view to the replicated servers. The fault hypothesis for external observations is omission failures, i.e., the probability of more severe failures is considered negligible [Sti78, Pow92].[38] If this failure semantics cannot be implemented by a single sensor there are two possibilities. Firstly, replication of sensors and secondly, application specific reasonableness checks for detection of sensors faults. Both methods can be used to provide a service that is consistent with the omission failure semantics [Mar90, CM91, Pol95b].

The influence of the fault hypothesis on the agreement problem has been treated extensively, e.g., [CASD85, Sch90]. Under assumption of the benign fault hypothesis crash or omission failures it is sufficient to reliably broadcast one correct sensor input to all processors in the group. For more severe fault hypotheses such as timing failures or byzantine failures all sensor inputs have to be exchanged among the processors. By justifying a benign fault hypothesis the required communication

[38]For automotive electronics such benign failure semantics are important to keep the necessary replication level—and thus costs—low.

bandwidth can be reduced by a factor $n$, since it is not necessary to communicate the sensor values of all $n$ replicated sensors. Benign fault hypotheses have been already adopted for automotive electronics to reduce complexity.

The problem of agreement on replicated sensor inputs under resource constraints is not addressed specifically in the published literature. The problem of fault-tolerant sensors [Mar90, CM91] is treated independently from the agreement problem [PSL80, CASD85, Cri90, BM93]. There is no integrated treatment of this problem that would allow an optimization to reduce the cost of communication. In the problem definition for agreement no assumptions are made on the relation between individual input values. For replicated sensors under the assumption of omission failures it is however guaranteed that each pair of non-faulty sensor inputs is within a fixed interval that represents the accuracy of the ensemble of replicated sensors. Therefore, a server which reads a sensor input has a priori knowledge on the maximum deviation of the other sensor inputs. Considering the bounded difference of sensor inputs the question is, what is the minimum amount of information that needs to be exchanged to guarantee agreement. In the following, a formalization of this problem is given and a lower bound is presented.

## The lower bound for information exchange

We are concerned with a set of sensors measuring a continuos state variable of a physical process. The value of the continuos state variable is denoted $v$. Any sensor which is used for the measurement of $v$ maps this value to a discrete representation $d$, $d \in D$ where $D$ is a finite set of discrete numbers. For a set of $n$ replicated sensors the discrete representations are denoted $d_i$, for $1 \le i \le n$. Having defined this mapping we can now define the *discrete replica accuracy*. The discrete replica accuracy is given by the maximum deviation between the discrete representation for a set of sensors observing a state variable $v$. More formally:

***Discrete replica accuracy:*** The discrete replica accuracy $\delta$, is defined such that $\delta$ is the smallest number $\delta \in D$ which fulfills the property $|d_i - d_j| \le \delta$ for each pair $(d_i, d_j)$.

When evaluating the discrete replica accuracy for a "real" system, besides the accuracy of sensors and analog to digital converters, also the *reading error* [KK90] needs consideration: Since it is impossible to achieve simultaneity the continuos state variables are observed at different times. It can be guaranteed in a synchronous system that all replicated observations occur within a given time interval. By taking into account that the slew rate for continuos state variable is limited, it can be guaranteed that the difference between observations in a given time interval is bounded. This bound can be incorporated into discrete replica accuracy. Based on the definition of the discrete replica accuracy it is now possible to define the a priori knowledge available in the system.

> **Theorem 1:** A set of $n$ replicated sensors measures a continuos state variable $v$. If sensor $i$ is correct and its discrete representation for the state variable $v$ is $d_i$ then it is a priori known that all discrete representations of other correct sensors $d_j$ fulfill the property
>
> $$d_j \in [d_i - \delta, d_i + \delta]$$
>
> where $\delta$ is the discrete replica accuracy for the representations $d_i$ and $d_j$ of $v$, $1 \leq i, j \leq n$.

The proof for this theorem, as well as for the following theorems are given in [Pol94]. This theorem defines the possible range for discrete representations of the other sensors with respect to the discrete representation of one particular sensor. Without exchanging the discrete representations each server knows that the discrete representations of the other servers are in the interval $[d_i - \delta, d_i + \delta]$ if $d_i$ is its own discrete representation. In case of sensor or server failures some other discrete representation can be empty. This is the view of any individual server. For an omniscient external observer, however, the interval of possible discrete representations is smaller, since it is known that $|d_i - d_j| \leq \delta$ for all pairs of discrete representations holds. That is, for a single server the uncertainty interval on the other discrete representations is $2\delta$ while the actual uncertainty is only $\delta$. Using the above given theorem a lower bound for the amount of information exchange for agreement can be derived:

> **Theorem 2:** A set of replicated sensors measures a continuos state variable under the assumption of omission failures. The lower bound for the amount of information $I$ that has to be exchanged to guarantee agreement on one discrete representation is
>
> $$I = \lceil \log_2(2\delta + 1) \rceil$$
>
> bits for non-faulty servers and sensors, where $\delta$ is the discrete replica accuracy.

This theorem defines the minimum number of bits that has to be exchanged to guarantee agreement on the discrete representations of replicated sensors. It is interesting to note that the number of bits $I$ depends only on the discrete replica accuracy $\delta$ rather than on the actual number of bits for the discrete representations. This independence of the number of bits for discrete representations reflects the a priori knowledge in the system. It is however important to consider the restrictions of this theorem. Firstly, the failure semantics of sensors are restricted to omission failures or a more benign fault hypothesis. And secondly, only non-faulty servers with non-faulty sensors can reach agreement. This is a restriction compared to classical agreement algorithms where non-faulty servers with faulty sensors can also reach agreement. Both restrictions are acceptable in the context of automotive electronics (and usually for other application areas too). In the following Table 6-5 shows the number of bits $l$ that has to be exchanged to achieve agreement as a function of discrete replica accuracy $\delta$.

| discrete replica accuracy $\delta$ | 1 | [2, 3] | [4, 7] | [8, 15] | [16, 31] | [32, 63] |
|---|---|---|---|---|---|---|
| minimum number of bits $l$ | 2 | 3 | 4 | 5 | 6 | 7 |

*Table 6-5: Minimum number of bits for agreement*

### An optimal agreement algorithm

The above presented lower bound defines the minimum number of bits that has to be exchanged for replica control on sensor inputs. It is now interesting to ask whether there exists algorithms that can achieve agreement with the minimum number of bits and if such an algorithm exists, what its computational complexity is. In the following it will be shown that an algorithm with constant complexity $O(1)$ exists. The existence of this algorithm shows the tightness of the lower bound.

To achieve agreement on discrete representations of replicated sensors with a minimum number of bits the following steps have to be carried out. (1) All servers obtain the discrete representations of the physical state variable via their local connected sensor. (2) One distinguished server, the transmitter, reduces its discrete representation by means of an algorithm to at most $l$ bits and broadcasts its reduced value to all the other servers (if the transmitter fails a new transmitter has to be selected). (3) All servers combine their local discrete representation with the broadcasted value to one agreed discrete representation by means of a second algorithm. Thus, there are two algorithms necessary, one to reduce a discrete representation to $l$ bits and one to combine the reduced value with a discrete representation into one agreed discrete representation.

To formalize this agreement problem the following notation is used. Again, the discrete representations of the sensor measurements are denoted $d_1, d_2, \dots, d_n$ for a set of $n$ replicated sensors. The discrete replica accuracy for $d_1, d_2, \dots, d_n$ is defined by $\delta$. To reach agreement on the discrete representations two functions are required. Firstly, the communication function $C$ which reduces the information of one discrete representation $c_i = C(d_i)$, $c_i \in D$. The communication function has to generate a number $c_i$ of up to $l$ bits which is sent by the transmitter via reliable broadcast to all the other servers (it is assumed that a reliable broadcast protocol is available that guarantees that all non-faulty servers agree on the communicated value $c_i$). Secondly, the agreement function $A$ is necessary to guarantee that all non-faulty servers with non-faulty sensors agree on the same discrete representation. The agreement function is therefore defined for every pair of discrete representations $(d_i, d_j)$ with $1 \le i, j \le n$, such that $d_i = A(d_j, c_i)$. By definition of $c_i = C(d_i)$ it follows that $d_i = A(d_j, C(d_i))$. The communication and agreement functions guarantee that two servers reach the same decision, if $c_i$ is agreed upon. Since there were no assumptions on $d_i$ and $d_j$, except that both are discrete representations of a state variable with discrete replica accuracy $\delta$, the functions $C$ and $A$ can also be used to attain agreement in any group of $n$ servers and sensors. Assume all servers agree on some $c_i$. By using the agreement function then $d_i = A(d_j, c_i)$ is guaranteed for all servers $1 \le j \le n$. Hence, all non-faulty servers with non-faulty sensors agree on $d_i$. The following theorem presents a communication and agreement function which fulfills the requirement that $c_i$ requires at most $I$ bits for information exchange.

**Theorem 3:** For a given discrete replica accuracy $\delta$ the communication function $C{:}D \rightarrow E, E \subset D$ and the agreement function $A{:}D \times E \rightarrow D$ satisfy the properties:

- *Optimality:* $|E| \le 2^I$ and
- *Agreement:* $d_i = A(d_j, C(d_i))$

where $d_i$ and $d_j$ are discrete representations with a discrete replica accuracy of $\delta$. $C$ and $D$ are defined as follows:

$$C(d_i) = d_i \bmod 2^I$$

$$A(d_j, c_i) = \begin{cases} d_j - C(d_j) + c_i & : \quad |C(d_j) - c_i| \le \delta \\ d_j - C(d_j) + c_i + 2^I & : \quad (|C(d_j) - c_i| > \delta) \wedge (C(d_j) > c_i) \\ d_j - C(d_j) + c_i - 2^I & : \quad (|C(d_j) - c_i| > \delta) \wedge (C(d_j) < c_i) \end{cases}$$

By theorem 3 it is shown[39] that $I$ bits of information exchange are sufficient to achieve agreement on non-deterministic discrete representations under the assumption of omission failures and a discrete replica accuracy $\delta$. Furthermore, by theorem 2 and 3 it is shown that the bound of $I$ bits for information exchange is tight. Also, the computational complexity for the communication function $C$ and the agreement function $A$ is $O(1)$ as claimed above. For example, the typical discrete accuracy of time stamps will be the digitizing error of ±1 digits. Hence the discrete replica accuracy $\delta$ is 2. According to theorem 2 and 3 it follows that $\lceil ld(2\delta + 1) \rceil = 3$ *bit* of information have to be exchanged, compared to the 16 *bit* which would typically be necessary if there is no a priori knowledge on $\delta$. At a transmission rate of 250 *kbps* one bit takes 4 $\mu s$ to transmit. Hence, it requires 64 $\mu s$ to send 16 *bit* while it

[39]The proof of this theorem is given in [Pol94].

takes 12 $\mu s$ to send 3 *bit* which saves 52 $\mu s$ in case of the optimal agreement protocol.

A further advantage of the functions $C$ and $A$ is their algorithmic simplicity. There are only simple instructions required, such as logical and, addition, subtraction, and comparison. This allows a very efficient implementation of this functions on any of the relatively simple microprocessors that are used for automotive electronics. Both functions have been implemented as assembler macros for the SAB 80C166 microcontroller [Sie90]. In the following, Table 6-6 gives the timing parameters for the implementations.

| function | *time* [$\mu s$] |
|---|---|
| communication function $C$ | 0.1 |
| agreement function $A$ | 1.1 |

*Table 6-6: Function execution times*

The actual implementations of these functions are given in [Pol94]. At a clock rate of 20 *MHz* the execution timing for the communication function and the agreement function are 0.1 $\mu s$ and 1.1 $\mu s$, respectively. Relating the total execution time of 1.2 $\mu s$ for both functions to the communication duration of 4 $\mu s$ for one bit shows that the functions are effective even if the number of bits for communication would be reduced by only one bit. However, for typical applications the number of bits will be reduced by much more than one bit. This shows that the latency period for replica determinism enforcement can be substantially reduced by sending only the minimum amount of information.

## 6.4 Deterministic preemptive scheduling

This section describes a methodology which allows preemptive scheduling while guaranteeing replica determinism without communication between servers. The basic idea is to use a simulation of common knowledge. In the case of preemptive scheduling the simulation of common knowledge has to guarantee that all servers in a group can act as if preemptions were taken simultaneously. This requires a restriction of the service behavior such that no server can contradict the assumption that preemptions are taken simultaneously. It has been shown in [NT87, NT93] that a simulation of common knowledge can only be attained if the problem has an "internal specification", i.e. the problem can be specified without explicit reference to real-time. Since external events occur in real-time this implies that preemptive scheduling decisions cannot be based on external events. Rather, the servers in a group have to agree on external events. The agreement converts the external events to system internal events (an optimal agreement protocol has been shown in the previous section). In the following it is therefore assumed that this protocol or any other suitable protocol is used for agreement of external events. These agreed upon events can then be used for preemption decisions and for event-triggered service activation.

## 6.4.1 Replica determinism and internal information

To achieve a simulation of common knowledge it is not sufficient that servers agree on external events. In addition agreement on system internal information is also necessary. For the remainder of this chapter it is assumed that information between services is exclusively exchanged by means of messages. A discussion of the message concept and possible semantics will be given later on. Currently it is sufficient to assume that a message can be sent and received whereby the value sent is returned upon receive.

Since the systems considered here have no shared memory, two types of message communication needs to be considered. Firstly, message communication within one processor (intra-processor) where the processors memory is used for message passing. And secondly, message communication between processors (inter-processor) where the communication media is used for message passing. For inter-processor communication it is possible to guarantee replica determinism by the communication protocol. But since the cost of sending a message via communication media is at least one order of magnitude higher than transferring a message to and from memory [Gra88] it is preferable to use inter-processor message communication whenever possible.[40] For intra-processor message communication, however, messages have to be considered as local information, if preemptions are used. It is not guaranteed that all replicated services send and receive intra-processor messages in the same order. There are two sources for this non-determinism. Firstly, due to non-identical replication: Replicated processors have to execute different sets of services which leads to different message send and receive order among individual services. And secondly, due to the real world abstraction limitation: Because processors with replicated clocks do not proceed simultaneously the send and receive order of messages may differ among replicated services. But only the slightest difference in the timing of service executions can lead to a violation of the requirement that all replicas read the same internal information.

To illustrate the violation of a consistent send and receive order consider the following example: Two identical servers S1 and S2 have to execute the replicated services $T_1$ and $T_2$. It is furthermore assumed that service $T_2$ has a higher priority than service $T_1$ and that service $T_2$ receives a message $m$ which is sent by service $T_1$. $T_1$ and $T_2$ are executed on the same processor, hence $m$ is a intra-processor message.

[40]It is a typically design consideration to assign closely coupled services on one processor to reduce the communication effort.

Figure 6-2 shows that the replicas of $T_2$ will receive a different contents of message *m*. Service $T_1$ is requested at both processors and starts execution. But the processing speed of $T_1$ varies at both processors. This leads to a case where service $T_2$ is requested and service $T_1$ has finished execution at processor S1 while it is currently executing at processor S2. Because $T_2$ has higher priority than $T_1$, $T_1$ is preempted at processor S2. Since $T_1$ has finished at processor S1, the message *m* has been sent already. At processor S2 the service $T_1$ was preempted before sending message *m*. Hence it follows that service $T_2$ will receive two different versions of message *m* at processor S1 and S2. Service $T_2$ at processor S2 reads an older message version than $T_2$ at processor S1. The internal information is inconsistent.

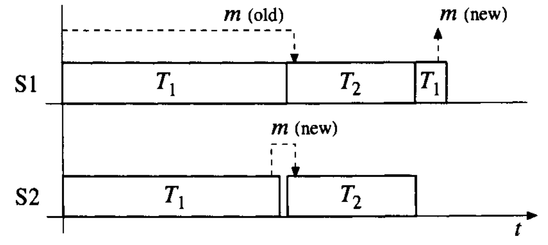

*Figure 6-2: Inconsistent internal information*

## 6.4.2 Simulating common knowledge by timed messages

To allow preemptions it is necessary to avoid this non-deterministic behavior of internal information. For a simulation of common knowledge it has to be guaranteed that all instances of a replicated service receive the same messages regardless of processing speed differences. There are two possibilities of achieving this behavior. Firstly, by using an agreement protocol before receiving messages. This requires all services to communicate for each intra-processor message. The obvious disadvantage of this approach is the extremely high communication and synchronization effort. In the section "Comparing requirements to related work" it has already been shown that this approach is not feasible under the given assumptions and requirements. The reason was that explicit synchronization by means of communication was too slow. This approach therefore needs not be considered any further.

The second approach to guarantee replica determinism is based on the availability of a priori knowledge that is available in synchronous systems. This novel approach uses so called *timed messages* to guarantee deterministic access to intra-processor messages without communication between processors. In a replicated non real-time system the service activation time has to be known by all services in the group and they have to agree on the timing. Real-time systems additionally have the a priori knowledge of when a service has to finish. The *finish time* is defined by the sum of the activation time plus the service's deadline. Since the service start time and the service deadline are agreed upon by all replicated services, it follows that the service finish time is also agreed upon by all services. Hence, there is implicit agreement on service finish times. Alternatively to the finish time the worst case *execution end time* of all replicated services can also be used. The execution end time is defined by the sum of the activation time plus the worst case service execution time (the worst case service execution time has to be smaller than the service deadline). However, the execution end time is not always known since it has to be derived by analytical methods while the finish time is known by the definition of the services deadline $d$. Figure 6-3 shows execution of service $T_1$ with its activation time, execution end time and finish time. Service $T_1$ gets preempted three times before it finishes.

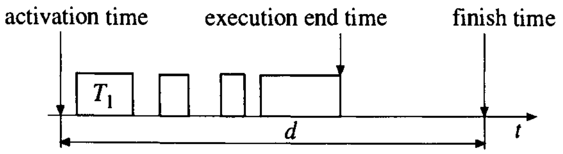

*Figure 6-3: Execution end and finish time*

This a priori knowledge of service finish times (or execution end time) can be used as follows to guarantee replica determinism. Each message is sent by one or more services. All sent messages are associated with the finish time of the sender service. This associated service finish time is called a message's *validity time*. Messages with validity times are called *timed messages*. If a service receives a message, it needs to selected a version such that the associated validity time is less than the receiver's service start time. It can be guaranteed by this approach that all receiver services read identical message contents. Because the service start time as well as the validity times of messages are agreed upon, it is guaranteed that all replicas select the same version of a message. Compared to normal message delivery, where messages become available immediately, timed messages become available at the latest possible moment.

The principle of operation of timed messages can be illustrated by reevaluating the above given example (Figure 6-2) which was used to show the inconsistency of messages. The application of timed messages to this example is shown in Figure 6-4. Again there are two identical processors S1 and S2 which have to execute the replicated services $T_1$ and $T_2$. It is furthermore assumed that service $T_2$ has a higher priority than service $T_1$ and that service $T_2$ receives a message $m$ which is sent by service $T_1$. Service $T_1$ is activated at both processors with the service activation time $t_1$. Some time later, at time $t_2$ service $T_2$ is activated. $T_1$ has finished its execution at processor S1 while it is still executing at processor S2. Since service $T_2$ has a higher priority than $T_1$, service $T_1$ becomes preempted at processor S2. At this time service $T_1$ has sent message $m$ at processor S1 with a validity time of $t_1 + d_1$, where $d_1$ is the deadline of service $T_1$. The slightly slower service at processor S2 has not yet sent the message $m$ since it was preempted by service $T_2$. When service $T_2$ starts to execute at time $t_2$ and it receives the message $m$, it does not read the new version sent at processor S1. This is because the validity time of this message version is $t_1 + d_1$ and $t_1 + d_1 > t_2$. Hence both services read the same (old) version of the message.

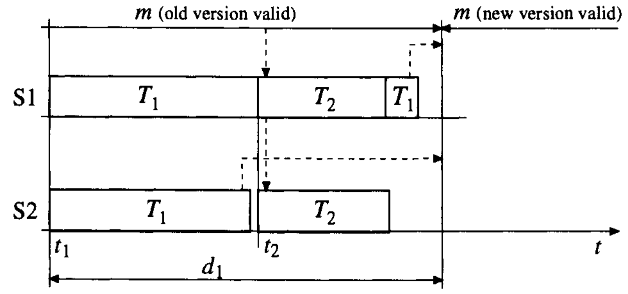

*Figure 6-4: Timed messages*

With timed messages it is possible to guarantee that all replicated services receive identical messages. Timed messages can be implemented at the level of the operating system. This hides implementation details of timed messages, such that servers which send or receive messages do not perceive any difference from the "normal" message semantics. Hence, it is possible to simulate common knowledge by timed messages. Or in other words, timed messages guarantee deterministic behavior of send and receive message operations. Hence, it also possible to guarantee replica determinism for preemptive scheduling by timed messages without communication for intra-processor messages.

## 6.4.3 Message semantics and message versions

The price for replica determinism enforcement with timed messages is that more than one version of a message has to be available simultaneously. This requires additional memory space for messages. The number of simultaneously available message versions depends on the message semantics. Possible semantics are either event messages or state messages which can be described as follows:

***Event-message semantics*** [Tan92]: Event-message semantics are characterized by a consuming receive operation, i.e. if a message is received, it gets consumed such that the following receive operation gets the next message. A message is therefore associated with an event that is processed upon receiving the message. Event-messages are typically implemented by message queues, since it is common that more than one message can be sent before receiving a message. With event-message semantics there is a synchronization between sender and receiver, for each message sent there is exactly one receiver. This 1:1 synchronization relation can be implemented with blocking or non-blocking semantics.

***State-messages semantics*** [KDK+89]: The semantics of state-messages are very similar to that of global variables. If a message is received the last value sent is returned, but the message does not get consumed. It is therefore possible to receive a message more than once. The difference between global variables and state-messages is that global variables can be overwritten at arbitrary points in time while the receiver of a state message gets a message copy that is kept unchanged if no further receive operation is performed. A state-message therefore reflects the last state of some entity which can be read by receiving a message or updated by sending a message. Between sender and receiver there is no synchronization relation with state messages. Rather, they can be used for an unsynchronized 1:*n* or *m*:*n* information exchange between servers, e.g. [Cla89, But93]. State messages are typically implemented by message pools. The sender puts the recent message in the pool while receivers get copies of this message from the pool.

With event-messages the size of the message queue has to be selected such that the data generated by the maximum number of send operations that may occur between two consecutive receive operations fits into the message queue. For timed messages with event semantics the queue size has to be selected differently. Since message receive operations only get messages who's validity time is earlier than the service's start time, the size of the message queue has to be selected such that the data generated by the maximum number of send operations that may occur between two consecutive service activations with receive operations fits into the message queue. However, regardless whether messages are timed or not, the dimensioning of message queues is a difficult problem with event semantics.

The size of the message pool for state messages is equal to the number of senders and receivers plus one, i.e. for each message there has to be one copy that holds the last state and for each sender and receiver there has to be an additional copy. In case of timed messages the number of additional messages depends on the send and receive timing of the services. The precise number of additional message copies per message is given by the maximum number of messages that can be sent within the maximum *replica receive time interval* of all services. The replica receive time interval is defined as follows: Consider a service that is replicated over *n* servers. Assume this service is required to receive a message *m*. The maximum time interval between any two receive message *m* operations is called replica receive time interval.

### 6.4.4 Implementing timed messages with status semantics

Before discussing implementation issues of timed messages a short explanation is given why status-messages should be used under the given requirements. In the area of automotive electronics nearly all of the services have periodic activation patterns, although the period may not be constant. This periodic service activation scheme is based on the fact that many of the services are servo loops or have to perform repetitive control tasks. In complex systems there are hierarchies of servo loops where the individual activation periods (sampling rates) are determined by control theory [Cla89]. The activation periods of individual services are therefore different. It is obvious that the information flow between services cannot be 1:1 since this would require a synchronization of service activation rates. Status-messages are well suited for this application area since 1:*n* and *m*:*n* information exchange can be implemented. Hence, status semantics are considered for implementation.

In the following some implementation issues are discussed. The first question when implementing a message service is whether to use a dynamic memory management scheme with pointers or a static memory management scheme with fixed addresses for message copies. The dynamic memory scheme has advantages if messages are large and pointer passing between services and the operating system is much faster than copying the message contents. With the static memory scheme each server has a fixed memory location which can be accessed directly. If the size of message data is small, then it is faster to copy message data than to manage pointers. Since the message sizes in the application area considered here are typically very small (95% are 2 *byte* messages), a static memory scheme is employed.

The next issue concerning timed messages is how to manage message versions with different validity times. Depending on the maximum number of message versions that have to be stored simultaneously, different implementations have to be selected. Under the assumption that the replica receive time interval of all messages is smaller than the corresponding message send interval at most two message versions have to be stored simultaneously. This assumption is guaranteed in identically replicated systems. But even in non-identically replicated systems the assumption will hold in the application areas considered here. The following implementation is therefore based on the assumption that at the most two message versions have to be stored and that the message size is small. Under this assumption it is sufficient to store an old version of the message, a new version and the validity time of the new message version. See Figure 6-5.

| |
|---|
| msg_val_new |
| val_time_new |
| msg_val_old |

*Figure 6-5: Timed message data structure*

The corresponding message operations send and receive are shown in Figure 6-6. The keywords activation_time and deadline denote the service activation time of the caller service and the deadline of the caller service respectively. Both are known by the operating system and therefore do not have to be passed as function arguments.

```
SendMsg(msg_id, value):
  begin atomic;
  msg_val_old(msg_id):= msg_val_new(msg_id);
  msg_val_new(msg_id):= value;
  val_time_new(msg_id):= activation_time + deadline;
  end atomic;

ReceiveMsg(msg_id):
  begin atomic;
  if (activation_time < val_time_new(msg_id))
    then
      return(msg_val_old(msg_id));
    else
      return(msg_val_new(msg_id));
  endif
  end atomic;
```

*Figure 6-6: Send and receive timed message*

Figures 6-5 and 6-6 show that the overhead for timed messages on memory space and runtime is small. However, there are some optimizations possible by considering the precedence constraints of services. There are two levels of optimization. Firstly, an optimized receive operation may be used if it is a priori known which version of the message has to be selected. In these cases the receive statement becomes a simple memory to memory copy operation. Secondly, the timed message implementation becomes superfluous if it is a priori guaranteed that all replicated services have access to a consistent version of the message. In this case both the send and receive operations becomes memory to memory copy operations. Some sufficient but not necessary criteria for these optimizations are given below.

Assume service $T_i$ sends a message $m$ and service $T_j$ receives the message $m$. There are three possible precedence relations between the two services that allow the use of the optimized receive operation:

| | |
|---|---|
| $T_i \rightarrow T_j$ | task Ti executes once before Tj |
| $T_i \overset{n}{\rightarrow} T_j$ | task Ti executes *n* times before Tj |
| $T_i \underset{n}{\rightarrow} T_j$ | task Ti executes before Tj executes *n* times |

Since service $T_i$ always executes before service $T_j$ it is a priori known that service $T_j$ can select the latest version of message *m*. For example, in a system with fixed priority scheduling there are precedence relations between all services at the same priority level. In this case the service $T_j$ can use the optimized receive message operation as given in Figure 6-7. Other services which have no such precedence relation however have to use the normal receive timed message implementation.

```
ReceiveMsg(msg_id):
  begin atomic;
  return(msg_val_new(msg_id));
  end atomic;
```

*Figure 6-7: Optimized receive timed message*

If a message *m* is sent and received by a set of services and for each pair of services within the set that has a send/receive relation there is a precedence relation, then the second optimization can be applied. In this case it is not necessary to use timed messages. The optimized send and receive operations can be used instead. Also it is sufficient to store the last version of the message only. In this case the optimized send and receive message implementations as given in Figure 6-7 and 6-8 can be used, respectively.

```
SendMsg(msg_id, value):
  begin atomic;
  msg_val_new(msg_id):= value;
  end atomic;
```

*Figure 6-8: Optimized send timed message*

## 6.5 Reevaluating replica determinism for automotive electronics

In the section "Comparing requirements to related work" it has been shown that neither active, semi-active nor passive replication could satisfy the requirements for short latency periods under the given assumptions. The available communication bandwidth of 250 *kbps* was in all three cases insufficient for explicit enforcement of replica determinism with event-triggered service activation and preemptive scheduling. This section reevaluates the problem of replica determinism enforcement under the assumptions that replica control of external events is done by means of the optimal agreement algorithm and by timed messages (see sections "Optimal agreement for replicated sensors" and "Deterministic preemptive scheduling").

Under the assumption that the observations of external events for event-triggered task activation have a discrete replica accuracy $\delta = 2$ and that only omission failures occur, the number of bits that has to be communicated is $\lceil ld(2\delta + 1) \rceil = 3$. It is furthermore assumed that a special purpose serial bus is available for agreement of external events. This bus supports atomic broadcast of messages. The length of message segments is 5 *bit*. This allows sending 3 *bit* of message data within one segment. The remaining 2 bits are used to detect message corruption. Media access control is done by a TDMA (time division multiple access) protocol which allocates every second 5 *bit* slot to the message with highest priority, every fourth 5 *bit* slot for the message with the second highest priority and so on. With a TDMA media control strategy it is not necessary to transmit message identifiers since the message identification is encoded in the transmission time of the message slot. This encoding of message identifiers in time also saves communication bandwidth by using a priori knowledge[41] [KG94]. To handle message corruption a message can be repeated $r$ times. It is assumed that additional TDMA slots are inserted upon transmission failures. This allows immediate repetition of messages. However, the worst case of communication bandwidth requirement for retries has to be considered for the protocol latency calculation. Since timed messages are used to guarantee replica determinism for preemptions, it is not necessary to transmit information about preemption points.

The following slightly modified active-replication strategy can be used to guarantee replica determinism within a short latency period. On occurrence of an external event all replicated servers interrupt the currently executing service. The worst case of latency time is 20 $\mu s$. The next action is to get the time-stamp of the external event, calculate the communication function $C$ of the time-stamp and request transmission. It is assumed that the communication controller is responsible for media access control and for send retries in case of failures. There is a retry ranking order among communication controllers such that the highest ranked controller sends the requested message first. If the message is not received successfully by all servers, then the communication controller with the second highest ranking retransmits the message. This protocol is repeated until at the most $r$ retries have been performed. The worst case medium access latency is 5 *bit* for the message with the highest priority. At the available communication speed of 250 *kbps* this requires 20 $\mu s$. If messages are retried up to $r$ times, $5(r + 1)$ *bits* have to be transmitted, which requires $20(r + 1)$ $\mu s$. The next step is to read the received result out of the communication controller and calculate the agreement function $A$. After this step an agreed upon time-stamp of the external event is available. Finally, the context has to be switched to the new activated service. Since timed messages are used, there is no agreement on preemption decisions necessary. The resulting worst case end-to-end delay between occurrence of the external event and the agreed upon service activation is given in Table 6-7.

[41]The system has a priori knowledge on *which* message is sent *when*.

| *action* | *time* [$\mu s$] |
|---|---|
| maximum latency to react on external event | 20 |
| calculation of communication function $C$ and transmission request | 5 |
| MAC latency (5 *bit*) | 20 |
| message transmission $(2+3)(r+1)$ *bit* | $20(r+1)$ |
| get received communication value and calculate agreement function | 5 |
| context switch to new service | 10 |
| total execution time | $20r+80$ |

*Table 6-7: Agreement execution (agreement function and timed messages)*

The resulting end-to-end protocol execution time is $20r + 80\ \mu s$ where $r$ is the maximum number of message send retries. Under the assumption of one message send retry ($r = 1$) this results in an execution time of $100\ \mu s$. Hence, it is possible to meet the established requirements for short latency periods under the given assumptions. Splitting the optimized protocols execution time to the execution time for servers and execution time for communication results in $pt^{opt}_{server} = 40\ \mu s$ and $pt^{opt}_{comm} = 20r + 40\ \mu s$. The server's execution time is independent of the group size and the number of retries with $pt^{opt}_{server} = 40\ \mu s$. With one retry ($r = 1$) the communication time is $pt^{opt}_{comm} = 20r + 40\ \mu s = 60\ \mu s$.

The basic problem of replica determinism enforcement for automotive electronics is the low available communication bandwidth. Due to this low communication bandwidth it is impossible to send enough information to ensure replica determinism completely on-line without taking restricting assumptions. The alternative is a combination of implicit and explicit replica determinism enforcement. By considering the a priori knowledge available, only a minimum of information has to be exchanged on-line. In the following the a priori knowledge and restrictions that have been used by the presented protocol are listed:

- ***Omission failure semantics:*** To reduce the amount of information that has to be sent, a benign fault hypothesis has been selected.

- ***Bounded accuracy of external observations:*** By considering that the difference between replicated observations is bounded (discrete replica accuracy $\delta$), the number of bits which have to be exchanged for agreement can be reduced.

- ***A priori known service deadlines:*** The assumption that service deadlines are known a priori is used to simulate common knowledge by means of timed messages.

- ***TDMA media access control:*** This media access control strategy was selected because message identifiers need not be sent explicitly. The message identifiers are encoded by the send times of messages.

- ***A priori known retry ranking:*** For message send retries an a priori known ranking is used which defines the server that has to retry after a failed communication.

The restriction to a bounded accuracy for external observations and the a priori known retry ranking can be guaranteed in asynchronous as well as in synchronous systems. However, the remaining protocol characteristics, omission failure semantics, a priori known service deadlines and TDMA media access control can only be guaranteed in a synchronous system. The additionally available a priori knowledge in synchronous systems therefore allows the achievement of performance advantages over asynchronous systems.

# Chapter 7

# Summary

This book is concerned with the problem of replica determinism in the context of fault-tolerant real-time systems. Systematic fault-tolerance is based on the replication of services. The problem of replica determinism thereby is to assure that replicated services show consistent behavior in the absence of failures, i.e. replica determinism is a necessary condition to allow differentiation between correct and faulty services.

After an general introduction, the second chapter of this book gives a brief introduction to automotive electronics which is an important application area for fault-tolerant real-time systems. It was shown that automotive electronics have very stringent efficiency requirements which are dictated by hardware cost. The general application area characteristics are presented followed by typical functional requirements for high-end systems. Additionally, there are demanding dependability requirements. Present problems and future directions to achieve these dependability goals are discussed. It has been shown that systematic fault-tolerance has very desirable properties. Thus, systematic fault-tolerance likely will be adopted for automotive electronics in favor of application-specific fault-tolerance. Systematic fault-tolerance, however, is based on replicated services and requires replica determinism.

In chapter three of this book the basics of replica determinism and non-determinism have been discussed without resort to actual implementations and methodologies. It was shown that a general definition of replica determinism can only be a generic framework. Detailed and formal definitions were shown to be strictly application specific, since it is impossible to achieve total or ideal replica determinism. A novel definition of replica determinism was given. Based on this definition possible modes of non-deterministic behavior were listed. This list shows that replica non-determinism is introduced by standard functions such as on-line scheduling, timeouts or readings of replicated sensors and others.

By definition of a cause/consequence chain on non-deterministic behavior the notation of the atomic source for replica determinism was introduced. Based on this definition a question important for system design was stated: how to characterize *all* the possible atomic sources for replica non-determinism. Such a characterization had been developed consisting of three atomic sources: (1) the real world abstraction limitation, (2) the impossibility of exact agreement and (3) intention and missing coordination. It has been shown that the first atomic source of replica determinism is caused by the continuos nature of the real world which is abstracted by the computer system with a finite set of discrete numbers. The real world abstraction limitation cannot be circumvented. The second atomic source for replica non-determinism is caused by the fact that a certain state of knowledge, namely common knowledge, cannot be attained in a "real" distributed system. This impossibility is of importance since it has been shown that replica determinism, knowledge and simultaneity are closely related to each other. It is therefore impossible to achieve ideal or total replica determinism with "real" systems. The requirements for replica determinism therefore need to be relaxed to achievable cases, which are shown to correspond to simulations of common knowledge and weaker states than common knowledge. Intention and missing coordination is the third atomic source for replica non-determinism. This is the only atomic source that can be circumvented by suitable methodologies. The final question of this part—whether there are systems that do not require enforcement of replica determinism—was concluded negative for non-trivial systems.

The fourth chapter of this book reviewed methodologies and implementations to enforce replica determinism, since it has been shown that replica determinism enforcement is necessary in non-trivial systems. The various strategies for replica determinism enforcement were discussed under special considerations of real-time and fault-tolerance aspects. In a first step the replica determinism enforcement strategies were classified according to the degree of internal and external replica control. Furthermore, the differences between central and distributed approaches were discussed.

The next aspect discussed was that of communication. Because any replica determinism enforcement strategy needs to exchange information, the properties of communication services are of great importance. Depending on the replication strategy, it has been shown that there are communication requirements ranging from simple point-to-point services to complex broadcast protocols with ordering properties or consensus protocols. These protocols have been discussed for synchronous as well as for asynchronous systems under special consideration of real-time constraints. The relation between synchronization and replica determinism is another aspect discussed. It was argued that synchronous approaches are appropriate in the value domain exclusively. In the time domain there is a broad spectrum of possibilities which are based on two different notions of time and temporal precedence, namely virtual synchrony and real-time synchrony. The latter was shown to be superior in replicated real-time systems with fault-tolerance requirements. Also the replica synchronization strategies lock-step execution, active replication, semi-active replication and passive replication were discussed. It was argued that active and semi-active replication are best suited for replicated real-time systems. Since the replication of components is aimed at fault-tolerance, the interdependency between replication strategies and failure semantics was discussed. Only active replication allows arbitrary failure semantics, all other strategies restrict the failure semantics of servers. The different recovery actions in case of failures have also been discussed along with the problem of redundancy preservation.

Having reviewed different replication strategies, the problem of replica determinism enforcement for automotive electronics was treated in chapter six. At first a definition for a short latency period was introduced. From the requirements defined by automotive electronics the important implementation choices for event-triggered service activation and preemptive scheduling were discussed. An evaluation of the replica control strategies active replication, semi-active replication and passive replication has shown that these cannot fulfill the given requirements for short latency periods. The available communication bandwidth was in all three cases insufficient to synchronize servers on behalf of the non-determinism which was introduced by event-triggered service activation and preemption decisions. To circumvent this negative result two optimizations have been introduced.

Firstly, a new agreement protocol for replicated sensor with a minimum amount of information has been introduced. This protocol is based on the assumption that replicated observations of continuos state variables have a bounded deviation and that only omission failures occur. A tight bound for the minimum amount of information that needs to be exchanged to guarantee agreement has been derived. Additionally, two efficiently to implement functions have been defined which achieve agreement with this minimum amount of information.

The second optimization described a new approach—called timed messages—which ensures deterministic access to internal information, even if preemptive scheduling is used. This approach is based on the concept of simulated common knowledge. It has been shown that replicated servers can act under this approach as if preemptions are taken simultaneously without communication between servers. This requires that multiple versions of intra-processor messages are available simultaneously. The number of necessary message versions has been discussed together with the different semantics of messages. For timed messages with status semantics an implementation was presented.

Finally, the problem of replica determinism enforcement for automotive electronics was reevaluated. By using the newly presented communication protocol for agreement with a minimum amount of information together with timed messages it has been shown that replica determinism enforcement can fulfill the given requirements. Systematic fault-tolerance is therefore general applicable even if event-triggered service activation and preemptive scheduling is used to achieve short latency periods.

# References

[AD76] P.A. Alsberg and J.D. Day. A Principle for Resilient Sharing of Distributed Resources. In *Proceedings of the Second International Conference on Software Engineering*. Oct. 1986, pages 627–644.

[ADLW87] M. Ahamad, P. Dasgupta, R.J. LeBlanc and C.T. Wilkes. Fault Tolerant Computing in Object Based Distributed Operating Systems. In *Proc. of the Sixth Symposium on Reliability in Distributed Software and Database Systems*. 1987, pages 115–125.

[AMM+93] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. Fast Message Ordering and Membership using a Logical Token-Passing Ring. In *The 13th International Conference on Distributed Computing Systems*. Pittsburgh, USA, May 25–28, 1993, pages 551–560.

[ARI92] ARINC-178B. Software Considerations in Airborne Systems and Equipment Certification. No. RTCA/DO-178B. RTCA. 1992.

[Avi77] A. Avižienis and L. Chen, On the Implementation of *N*-Version Programming for Software Fault-Tolerance During Program Execution. In *Proceedings Compsac 77*. Computer Society Press of the IEEE. Chicago, IL, USA. Nov. 1977, pages 149–155.

[Bar81] J. Bartlet. A NonStop Kernel. In *Eight Symposium on Operating System Principles*. Dec. 1981, pages 22–29.

[BBC+90] J. Bartlett, W. Bartlett, R. Carr, D. Garcia, J. Gray, R. Horst, R. Jardine, D. Lenoski, and D. McGuire. Fault-Tolerance in Tandem Computer Systems. Tandem Computers Inc., Cupertino, CA., Tandem Technical Report 90.5, May 1990.

[BD84] Ö. Babaoḡlu and R. Drummond. Communication Architectures for fast Reliable Broadcasts. In *Proceedings of the Sixth Symposium on Reliability in Distributed Software and Database Systems*. 1984, pages 2–10.

[Ben83] M. Ben-Or. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. In *Proceedings of the 2nd ACM Annual Symposium on Principles of Distributed Computing*. Montreal, Canada, 1983, pages 27–30.

[Ben87] M. Ben-Or. Randomized Agreement Protocols. In *Fault-Tolerant Distributed Computing. Lecture Notes in Computer Science*, Vol. 448. Springer-Verlag. New York, pages 72–96.

[Ben95] Private communications with Bob Bentley. IFIP WG 10.4 Workshop, Lake Tahoe, 1995.

[Ber88] P. Bernstein. Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing. *IEEE Computer*. Feb. 1988. pages 37–45.

[BG90] D. Barbara and H. Garcia-Molina. The Case for Controlled Inconsistency in Replicated Data. In *Proceedings of the Workshop on Management of Replicated Data*. Houston, Nov. 1990, pages 35–42.

[BG92] D. Bell and J. Grimson. Distributed Database Systems. Addison-Wesley (International computer science series), 1992.

[BGT90] N. Budhiraja, A. Gopal, and S. Toueg. Early Stopping Distributed Bidding and Applications. In *Proceedings of the 4th International Workshop on Distributed Algorithms*, Springer Verlag, Lecture notes in Computer Science 486. Sep. 1990, pages 304–320.

[BHB+90] P.A. Barret, A.M. Hilborne, P.G. Bond, D.T. Seaton, P. Veríssimo, L. Rodrigues, and N.A. Speirs, The Delta-4 Extra Performance Architecture (XPA). In *20th International Symposium on Fault-Tolerant Computing—FTCS 20*. Computer Society Press of the IEEE. Chapel Hill, NC, USA. Jun. 1990, pages 481–488.

[Bir85] K.P. Birman. Replication and Fault-Tolerance in the ISIS System. *ACM Operating Systems Review*. Vol. 19, Nr. 5, pages 79–86.

[Bir93] K.P. Birman. The Process Group approach to Reliable Distributed Computing. *Communications of the ACM*. Vol. 36, Nr. 12, 1993, p ages 37–53.

[BJ87a] K.P. Birman and T.A. Joseph. Exploiting Virtual Synchronity in Distributed Systems. In *Proc. of the 11th ACM Symposium on Operating System Principles*. Austin, Texas. Nov. 1987, pages 123–128.

[BJ87b] K.P. Birman and T.A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*. Vol. 5, No. 1, Feb. 1987, pages 47–76.

[BJRA84] K.P. Birman, T.A. Joseph, T. Raechle, and A. El Abbadi. Implementing Fault-Tolerant Distributed Objects. In *4th Symposium on Reliability in Distributed Software and Database Systems*. IEEE Computer Society in cooperation with ACM. Silver Spring, Maryland. 1984, pages 124–133.

[BKL89] S. Brilliant, J. Knight, and N. Levenson. The Consistent Comparison Problem in *N*-Version Software. *IEEE Transactions on Software Engineering*. Vol. 15, Nr. 11, 1989, pages 1481–1485.

[BL87] J.E. Burns, and N.A. Lynch. The Byzantine Firing Squad Problem. *Advances in Computing Research.* JAI Press. Vol. 4, 1987, pages 147–161.

[BM93] M. Barborak and M. Malek. The Consensus Problem in Fault-Tolerant Computing. *ACM Computing Surveys.* Vol. 25, No. 2, June 1993, pages 171–220.

[BMST92a] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. Optimal Primary-Backup Protocols. In *Proceedings of the Sixth International Workshop on Distributed Algorithms.* Haifa, Israel. 1992, pages 362–378.

[BMST92b] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. Primary-Backup Protocols: Lower Bounds and Optimal Implementations. In *3rd IFIP International Working Conference on Dependable Computing for Critical Applications.* Mondello, Sicily, Italy. Sep. 1992, pages 187–196.

[Bra87] G. Bracha. A $O(\log n)$ Expected Rounds Randomized Byzantine Generals Protocol. *Journal of the ACM.* Vol. 34, Nr. 4, pages 824–840.

[BSD88] Ö. Babaoḡlu, P. Stephenson, and R. Drummond. Reliable Broadcasts and Communication Models: Tradeoffs and Lower bounds. *Distributed Computing.* Springer Verlag. Nr. 2, 1988, pages 177–189.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweigth Causal and Atomic Group Multicast. *ACM Transactions on computer systems.* Vol. 9, No. 3, August 1991, pages 272–314.

[Bur91] A. Burns. Scheduling Hard Real-Time Systems: A Review. *Software Engineering Journal.* May. 1991, pages 116–128.

[But93] G.C. Buttazzo. HARTIK: A Real-Time Kernel for Robotics Application. In *Proceedings of the 14th International Symposium on Real-Time Systems.* IEEE Computer Society Press. 1993. Raleigh-Durham, North Carolina, Dec. 1–3. pages 201–205.

[CAN85] German patent application DE 35 06 118 A1. Verfahren zum Betreiben einer Datenverarbeitungsanlage für Kraftfahrzeuge. (CAN), Filed by *Robert Bosch GmbH,* 22. Feb. 1985.

[Car82] W.C. Carter. A Time for Reflection. In *Proceedings of the 12th International Symposium on Fault-Tolerant Computing* (FTCS-12). IEEE Computer Society Press. Santa Monica, CA. Jun. 1982, pg. 41.

[Car87] W.C. Carter. Experiences in Fault Tolerant Computing, 1947–1971. In *Dependable Computing and Fault-Tolerant Systems.* A. Avižienis, H. Kopetz and J.C. Laprie (Ed). Springer Verlag. Wien, New York. 1987, pages 1–36.

[CASD85] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. In *The 15th Annual International Symposium on Fault-Tolerant Computing* (FTCS-15). IEEE Computer Society Press. Ann Arbor, Michigan, Jun. 1985, pages 200–206.

[CC86] Y.-K. Chin and F.E. Coats. Engine Dynamics: Time-Based versus Crank Angle Based. In *SAE International Congress and Exposition*, Detroit, MI, USA, SAE Special publications, Warrendale, PA, USA. SAE Technical Paper 860412, Feb. 1986, pages 15–35.

[CD86] B.A. Coan and C. Dwork. Simultaneity is Harder than Agreement. In *Fifth Symposium on Reliability in Distributed Software and Database Systems*. IEEE Computer Society Press. Los Angeles, California, Jan. 1986, pages 141–150.

[CDD90] F. Cristian, B. Dancey, and J. Dehn. Fault-Tolerance in the Advanced Automation System. In *20th International Symposium on Fault-Tolerant Computing—FTCS 20*. Computer Society Press of the IEEE. Chapel Hill, NC, USA. Jun. 1990, pages 6–17.

[Cla89] D. Clark. HIC: An Operating System for Hierarchies of Servo Loops. In *IEEE International Conference on Robotics and Automation*. IEEE Computer Society Press. 1989. pages 1004–1009.

[CM84] J.M. Chang and N.F. Maxemchuck. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*. Vol. 2, No. 3, Aug. 1984, pages 251–273.

[CM86] K.M. Chandy and J. Misra. How Processes Learn. *Distributed Computing*. Springer Verlag. Nr.1, 1986, pages 40–52.

[CM91] P. Chew and K. Marzullo. Masking Failures of Multidimensional Sensors. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*. IEEE Computer Society Press. Oct. 1991, pages 32–41.

[Coo84] E.C. Cooper. Circus: A Replicated Procedure Call Facility. In *4th Symposium on Reliability in Distributed Software and Database Systems*. IEEE Computer Society in cooperation with ACM. Silver Spring, ML. 1984, pages 11–24.

[CPR$^+$92] M. Chérèque, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron. Active Replication in Delta-4. In *Proceedings of the 22th International Symposium on Fault-Tolerant Computing* (FTCS-22). IEEE Computer Society Press. Boston, Massachusetts, Jul. 8–10, 1992, pages 28–37.

[Cri88] F. Cristian. Agreeing on Who is Present and Who is Absent in a Synchronous Computer System. In *Proceedings Fault Tolerant Computing* (FTCS-18). IEEE Computer Society Press. Jun. 1988, pages 206–211.

[Cri89] F. Cristian. Exception handling. In *Dependability of Resilient Computers*, T. Anderson (Ed). Blackwell Scientific Publications, Oxford. 1989.

[Cri90] F. Cristian, Synchronous Atomic Broadcast for Redundant Broadcast Channels. *The Journal of Real-Time Systems*. Vol. 2, Nr. 3, Sep. 1990, pages 195–212.

[Cri91a] F. Cristian. Agreement on Processor-Group Membership in Distributed Systems. *Distributed Computing*. Vol. 6, Nr. 4, 1991, pages 175–178.

[Cri91b] F. Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*. Vol. 34, Nr. 2, Feb. 1991, pages 57–78.

[CS93] D.R. Cherington and D. Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*. Asheville, NC., Dec. 5–8, 1993, pages 44–57.

[CSS85] E.W. Czeck, D.P. Siewiorek, and Z. Segall. Fault Free Performance Validation of a Fault-Tolerant Multiprocessor: Baseline and Synthetic Workload Measurements. *Technical Report* CMU-CS-85-117. Carnegie Mellon University, Department of Computer Science. 1985.

[CT91] T. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*. Montreal, Canada, Aug. 19–21, 1991, pages 325–340.

[CW92] J. Cao and K.C. Wang. An Abstract Model of Rollback Recovery Control in Distributed Systems. *Operating Systems Review*. ACM Press. Vol. 26, Nr. 4, Oct. 1992, pages 62–76.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the Minimal Synchronism needed for Distributed Consensus. *Journal of the ACM*. Vol. 34, Nr. 1, Jan. 1987, pages 77–97.

[DHS86] D. Dolev, J.Y. Halpern and H. R. Strong. On the Possibility and Impossibility of Achieving Clock Synchronization. *Journal of Computer and System Science*. Vol. 32, 1986, pages 230–250.

[DLS88] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*. Vol. 35, Nr. 2, Apr. 1988, pages 288–323.

[Dij74] E.W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*. Vol. 17, 1974, pages 643–644.

[Dol82] D. Dolev. The Byzantine Generals Strike Again. *Journal of Algorithms*. Vol. 3, Nr. 1, 1982, pages 14–30.

[DR85] D. Dolev and R. Reischuck. Bounds on Information Exchange for Byzantine Agreement. *Journal of the ACM*. Vol. 32, Nr. 1, Jan. 1985, pages 191–204.

[DRS90] D. Dolev, R. Reischuck, and H.R. Strong. Early Stopping in Byzantine Agreement. *Journal of the ACM*. Vol. 37, Nr. 4, 1990, pages 720–741.

[DS83] D. Dolev and H. Strong. Authenticated Algorithms for Byzantine Agreement. *SIAM Journal on Computing*. Vol. 12, Nr. 4, 1983, pages 656–666.

[Edw67] P. Edwards (Ed). The Encyclopedia of Philosophy. *Volume 2*, Determinism. Collier – Macmillan Limited, London, 1967, pages 359–378.

[FDLP88] J. Fabre, Y. Deswarte, J.C. Laprie and D. Powell. Saturation: Reduced Idleness for Improved Fault Tolerance. In *Proceedings Fault Tolerant Computing* (FTCS-18). IEEE Computer Society Press. Jun. 1988, pages 200–205.

[Fid89] E.J. Fidrich. Draft Directive Concerning §30 Straßenverkehrs-Zulassungs-Ordnung "Testing of Vehicle Systems Including Electronic Components". In *VDI Berichte*. Verein Deutscher Ingenieure, VDI-Gesellschaft Fahrzeugtechnik, Electronic Systems for Vehicle, Bericht 1009. 1992. pages 399–412.

[FL82] M. Fischer, and N. Lynch. A Lower Bound for the Time to Assure Interactive Consistency. *Information Processing Letters 14*. Nr. 4, Apr. 1982, pages 183–186.

[FLP85] M. Fischer, N. Lynch and M. Paterson. Impossibility of Distributed Consensus with one Faulty Processor. *Journal of the ACM*. Vol. 32, Nr. 2, Apr. 1985, pages 374–382.

[GL91] T. Goelzer and R. Leonhard. Robert Bosch GmbH. In *International Symposium Vehicle Electronics Integration*. ATA-EL 91. Turin, 8–9. Apr. 1991, pages 17–29.

[GM82] H. Garcia-Molina. Elections in a Distributed Computing System. *IEEE Transactions on Computers*. Vol. 31, Nr. 1, Jan. 1982, pages 48–59.

[GM95] J.A. Garay and Y. Moses. Fully Polynomial Byzantine Agreement for $n > 3t$ Processors in $t + 1$ Rounds. To appear in *SIAM Journal on Computing*. 6/1995.

[Gop92] A. Gopal. Fault-Tolerant Broadcasts and Multicasts: The Problem of Inconsistency and Contamination. Ph.D. Dissertation, Computer Science Dept., Cornell Univ., Ithaca, New York. 1992.

[Gra78] J. Gray. Notes on Database Operating Systems. In *Operating Systems – An Advanced Course*. R. Bayer, R.M. Graham, and G. Seegmuller, (ed.), Lecture Notes on Computer Science. Vol. 66, Springer Verlag, 1978, pages 393–481. Also Available as *IBM Research Report* RJ 2188. IBM, Aug. 1978.

[Gra88] J. Gray. The cost of messages. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*. Toronto, Canada, Aug. 15-17, 1988, pages 1–7.

[Grü93] G. Grünsteidl. Dezentrales Redundanzmanagement in verteilten Echtzeitsystemen. Ph.D. Dissertation, Technische Universität Wien, Wien, Österreich, Jän. 1993.

[GS89] H. Garcia-Molina and Annemarie Spauster. Message Ordering in a Multicast Environment. In *The 9th International Conference on Distributed Computing Systems*. IEEE Computer Society Press. 1989, pages 354–361.

[GS90] F. Di Giandomenico and L. Strigini. Adjudicators for Diverse-Redundant Components. In *Proceedings of the Ninth Symposium on Reliable Distributed Systems*. IEEE Computer Society Press. Huntsville, AL. 1990, pages 114–123.

[GT91] A. Gopal and S. Toueg. Inconsistency and Contamination. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*. Montreal, Canada, Aug. 19–21, 1991, pages 257–272.

[Hag90] G. Hager. Task-Directed Sensor Fusion and Planning—A Computational Approach. Kluwer Academic Publishers, Norwell, Massachusetts, 1990, pages 15–51.

[HK89] A. Herzberg and S. Kutten. Efficient Detection of Message Forwarding Faults. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, Edmonton, Alberta, August 1989.

[HM84] J. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. Vancouver, Canada, Aug., 1984 (Revised Version dated Nov. 21, 1985), pages 50–61.

[HM90] J. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment. *Journal of the ACM*. Vol. 37, Nr. 3, Jul., 1990, pages 549–587.

[Hoa85] C.A.R. Hoare, Communicating Sequential Processes, *Prentice Hall International*, 1985

[HS92] H. Higaki and T. Soneoka. Fault-Tolerant Objects by Group-to-group Communication in Distributed Systems. In *The second international Workshop on Responsive Systems*. Kamifukuoka, Japan. Oct. 1992, pages 62–71.

[Hun92] T. Huntsberger. Sensor Fusion in a Dynamic Environment. In *Proceedings on Sensor Fusion V.* SPIE—The International Society for Optical Engineering. Vol. 1828, Nov. 15–17, 1992, pages 175–182.

[HT93] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. *Distributed Systems.* Second Edition, S. Mullender, (ed.). ACM Press. 1993, pages 97–145.

[IBM87] IBM International Technical Support Centers. IBM/VS Extended Recovery Facility (XRF) Technical Reference. Technical Report GG24-3153-0, IBM, 1987.

[KDK+89] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, C. Senft and R. Zainlinger. The MARS approach. *IEEE Micro.* Vol. 9, Nr. 1, Feb. 1989, pages 25–40.

[KG94] H. Kopetz and G. Grünsteidl. TTP - A Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer.* Vol. 27, No. 1, Jan. 1994, pages 14–23.

[KGR89] H. Kopetz, G. Grünsteidl, and J. Reisinger. Fault-Tolerant Membership Service in a Synchronous Distributed Real-Time System. In *Dependable Computing for Critical Applications*, Vol. 4 of Dependable Computing and Fault-Tolerant Systems, A. Avižienis and J.C. Laprie (ed.). Springer Verlag, 1991, pages 441–429.

[KK90] H. Kopetz, and K. Kim. Temporal Uncertainties in Interaction among Real-Time Objects. In *Proceedings of the 9th Symposium on Reliable Distributed Systems.* IEEE Computer Society Press, Huntsville, AL. Oct. 1990. pages 165–174.

[KKG+90] H. Kopetz, H. Kantz, G. Grünsteidel, P. Puschner, and J. Reisinger. Tolerating Transient Faults in MARS. In *Proceedings Fault Tolerant Computing FTCS-20.* IEEE Computer Society Press. Newcastle upon Tyne, UK. 1990, pages 466–473.

[KKMS95] H. Kopetz, A. Krüger, D. Millinger and A. Schedl. A Synchronization Strategy for a Time-Triggered Multicluster Real-Time System. To appear in *14th Symposium on Reliable Distributed Systems.* Bad Neuenahr, Germany, Sept. 1995.

[KKM+92] K. Kim, H. Kopetz, K. Mori, E. Shokri, and G. Grünsteidl. An Efficent Decentralized Approach to Processor-Group Membership Maintenance in Real-Time LAN Systems: The PRHB/ED Scheme. In *Proceedings of the 11th Symposium on Reliable Distributed Systems.* Houston, USA, Oct. 1992, pages 74–83.

[KM92] T.-W. Kuo and A. Mok. Application Semantics and Concurrency Control of Real-Time Data-Intensive Applications. In *Proceedings of the 13th International Symposium on Real-Time Systems.* IEEE Computer Society Press. 1992, pages 35–45

[KM93] T.-W. Kuo and A. Mok. SSP: A Semantics-Based Protocol for Real-Time Data Access. In *Proceedings of the 14th International Symposium on Real-Time Systems*. IEEE Computer Society Press. 1993, pages 76–86.

[KMP91] M. Koutny, L.V. Mancini, and G. Pappalardo. Formalising Replicated Distributed Processing. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*. IEEE Computer Society Press. Pisa, IT. 1991, pages 108–117.

[KMP94] M. Koutny, L.V. Mancini, and G. Pappalardo. Two Implementation Relations and the Correctness of Communicating Replicated Processes. Technical Report Nr. 491. University Newcastle upon Tyne. Oct. 1994.

[KO87] H. Kopetz and W. Ochsenreiter. Clock Synchronization in Distributed Real-Time Systems. *IEEE Transactions on Computers*. Vol. 36, Nr. 8, Aug. 1987, pages 933–940.

[Kop86] H. Kopetz. Scheduling in Distributed Real Time Systems. In *Proceedings on the Advanced Seminar on Real-Time Local Area Networks*. INRIA. Bandol, France. Apr. 1986, pages 105–126.

[Kop91] H. Kopetz. Event-Triggered versus Time-Triggered Real-Time Systems. In *Proceedings of the Workshop: Operating Systems of the 90ties and Beyond*. Springer Lecture Notes on Computer Science. 1991, pages 87–101.

[Kop92] H. Kopetz. Sparse Time versus Dense Time in Distributed Real-Time Systems. In *12th International Conference on Distributed Computing Systems*. IEEE Computer Society Press. Yokohama, Japan. Jun. 9-12, 1992, pages 460–467.

[Kop95] H. Kopetz. Automotive Electronics—Present State and Future Perspectives. In *25th International Symposium on Fault-Tolerant Computing*. Special Issue. Pasadena, CA. 1995, pages 66–75.

[KT87] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*. Vol. 13, Nr. 1, 1987, pages 23–31.

[KT91] M.F. Kaashoek and A.S. Tanenbaum. Group Communication in the Amoeba Distributed Operating System, In *Proc. 11th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, Los Alamitos, Calif., May 1991, pages 222–230.

[KTW+88] R.M. Kieckhafer, P.M. Thambidurai, C.J. Walter, and A.M. Finn. The MAFT Architecture for Distributed Fault-Tolerance. *IEEE Transactions on Computers*. Vol. 37, Nr. 4, 1988, pages 394–405.

[LA90] P.A. Lee, and T. Anderson. Fault Tolerance. In *Dependable Computing and Fault-Tolerant Systems*. A. Avižienis, H. Kopetz and J.C. Laprie (Ed), chapter 7, Error Recovery. Springer Verlag. Wien, New York. 1990, pages 143–185.

[Lam78a] L. Lamport. The Implementation of Reliable Distributed Multiprocessor Systems. *Computer Networks*. Vol. 2, 1978, pages 95–114.

[Lam78b] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communication of the ACM*. Vol. 21, Nr. 7, 1978, pages 558–565.

[Lam81] B. Lampson. Atomic Transactions. In *Distributed Systems—Architecture and Implementation*, B. Lampson, M. Paul and H. Siegert (eds.), Lecture Notes in Computer Science, Vol. 105, Springer-Verlag, pages 246–265.

[Lam84] L. Lamport. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Transactions on Programming Languages and Systems*. Vol. 6, Nr. 2, Apr. 1984, pages 254–280.

[Lap92] J.C. Laprie (Ed). Dependability: Basic Concepts and Terminology. *Volume 5 of Dependable Computing and Fault-Tolerant Systems*, chapter 5.2, Fault-Tolerance. Springer Verlag. Wien, New York. 1992, pages 23–28.

[LF82] L. Lamport and M.J. Fischer. Byzantine Generals and Transaction Commit Protocols. SRI International, Technical Report 62.

[LH94] J.H. Lala and R.E. Harper. Architectural Principles for Safety-Critical Real-Time Applications. *Proceedings of the IEEE*. Jan. 1994, pages 25–40.

[LLS+91] P. Lauvin, A. Löffler, A. Schmitt, W. Zimmermann, and W. Fuchs. Electronically Controlled Highpressure Unit Injector System for Diesel Engines. In *SAE International Congress and Exposition*, Detroit, MI, USA. SAE Special publications, Warrendale, PA, USA. SAE Technical Paper 911819. Sep. 1991, pages 1–13.

[LM85] L. Lamport and P.M. Melliar-Smith. Synchronizing Clocks in the Presence of Faults. *Journal of the ACM*. Vol. 32, Nr. 1, Jan. 1985, pages 52–78.

[LP94] C.A. Lingley-Papadopoulos. The Totem Process Group Membership and Interface. Master's Thesis, University of California, Santa Barbara, Aug. 1994.

[LSP82] L. Lamport, R. Shostak and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*. Vol. 4, Nr. 3, Jul. 1982, pages 382–401.

[Lyn89] N.A. Lynch. A Hundred Impossibility Proofs for Distributed Computing. In *Proceedings of the 8th Symposium on Principle of Distributed Computing*. ACM press. Edmonton, Canada. Aug. 14–16, 1989, pages 1–27.

[MAM+93] L.E. Moser, Y. Amir, P.M. Melliar-Smith, and D.A. Agrawal. Extended Virtual Synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems*. Poland, 1994, pages 56–65.

[Mar90] K. Marzullo. Tolerating Failures of Continuos-Valued Sensors. *ACM Transactions on Computer Systems*. Vol. 8, No. 4, Nov. 1990. pages 284–304.

[MB76] R. Metcalfe, and D. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communication of the ACM*. Vol. 19, Nr. 7, Jul. 1976, pages 396–403.

[MLM+95] H.-J. Mathony, H. Lohner, M. Mutter, J. Unruh, and S. Poledna. Echtzeitfähige Kommunikationssoftware für vernetzte Steuergeräte im Kfz. *ATZ Automobiltechnische Zeitschrift*. Vol. 97, Nr. 4, April 1995, pages 208–214.

[MM89] P.M. Melliar-Smith, and L.E. Moser. Fault-Tolerant Distributed Systems Based on Broadcast Communication. In *Proceedings of the 9th International Conference on Distributed Computing Systems*. IEEE Computer Society Press, 1989, pages 129–134.

[MMA90] P.M. Melliar-Smith, L.E. Moser, and V. Agrawala. Broadcast Protocols for Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*. Vol. 1, Nr. 1, Jan. 1990, pages 17–25.

[Mok83] A.K. Mok. Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment. Masterthesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May. 1983.

[MP88] L. Mancini and G. Pappalardo. Towards a theory of replicated processing. In *Formal Techniques in Real-Time and Fault-Tolerant Systems. Lecture Notes in Computer Science*, Vol. 331. Springer-Verlag. New York, pages 175–192. 1988.

[MPS89] S. Mishra, L.L. Peterson and R.D. Schlichting. Implementing Fault-Tolerant Replicated Objects Using Psync. In *Eight Symposioum on Reliable Distributed Systems*. Computer Society Press of the IEEE. Seattle, Washington. Oct. 1989, pages 42–52.

[MPS91] S. Mishra, L.L. Peterson and R.D. Schlichting. A Membership Protocol Based on Partial Order. In *2nd International Working Conference on Dependable Computing for Critical Applications*. Tucson, Arizona, USA, Feb. 1991, pages 137–145.

[NEC86] NEC. Data Book Microprocessors and Peripherals. µPD70616 (V60) CMOS 32-Bit Microprocessor. Oct. 1986, pages 3.229–3.279.

[Nei88] G. Neiger. Knowledge Consistency: A Useful Suspension of Disbelief. In *Proceedings of the 2nd Conference on Theoretical Aspects of Reasoning about Knowledge*. San Maeto, Calif. 1988, pages 295–308.

[Neu56] J. von Neumann. Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components. *Automata Studies*. C.E. Shannon and J. McCarthy (Ed). Princeton University Press. 1956, pages 43–98.

[NT87] G. Neiger and S. Toueg. Substituting for Real Time and Common Knowledge in Asynchronous Distributed Systems. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*. Vancouver, Canada, Aug. 10–12, 1987, pages 281–293.

[NT93] G. Neiger and S. Toueg. Substituting for Real Time and Common Knowledge in Asynchronous Distributed Systems. *Journal of the ACM*. Vol. 40, No. 3, Apr. 1990, pages 334–367.

[PB86] D.L. Palumbo and R.W. Butler. A Performance Evaluation of the Software-Implemented Fault-Tolerance computer. *AIAA Journal of Guidance, Control, amd Dynamics*. Vol. 9, No. 2, Mar.–Apr. 1986. pages 175–180.

[Per92] L. Perlovsky. Model-based Sensor Fusion. In *Proceedings on Sensor Fusion V*. SPIE—The International Society for Optical Engineering. Vol. 1828, Nov. 15–17, 1992, pages 197–200.

[PL90] C. Pu and A. Leff. Epsilon-serializability. Technical Report CUCS-054-90, Department of Computer Science, Columbia University, Dec. 1990.

[Pol93] S. Poledna. Reliability of Event-Triggered Task Activation for Hard Real-Time Systems. In *Proceedings of the 14th International Symposium on Real-Time Systems*. IEEE Computer Society Press. 1993. Raleigh-Durham, North Carolina, Dec. 1–3, pages 206–210.

[Pol94] S. Poledna. Replica Determinism in Fault-Tolerant Real-Time Systems. Ph.D. Thesis, Technical University Vienna, Austria, Apr. 1994.

[Pol95a] S. Poledna. Fault-Tolerance in Safety Critical Automotive Applications: Cost of Agreement as a Limiting Factor. In *25th International Symposium on Fault-Tolerant Computing*. Pasadena, CA, June 27–30, 1995, pages 73–82.

[Pol95b] S. Poledna. Tolerating Sensor Timing Faults in Highly Responsive Hard Real-Time Systems. *IEEE Transactions on Computers*. Special issue on fault-tolerant computing. Vol. 44, Nr. 2, 1995, pages 181–191.

[Pöp93] Ch. Pöppe. Irreführung durch Software. In *Spektrum der Wissenschaft*. Nov. 1993. pages 18-23.

[Pow91a] D. Powell (Ed). Delta-4: A Generic Architecture for Dependable Computing. *Volume 1 of ESPRIT Research Reports*, chapter 6.3, Models of Distributed Computation. Springer Verlag. Wien, New York. 1991, pages 99–100.

[Pow91b] D. Powell (Ed). Delta-4: A Generic Architecture for Dependable Computing. *Volume 1 of ESPRIT Research Reports*, chapter 6.4, Replicated Software Components. Springer Verlag. Wien, New York. 1991, pages 100–104.

[Pow91c] D. Powell (Ed). Delta-4: A Generic Architecture for Dependable Computing. *Volume 1 of ESPRIT Research Reports*, chapter 6.7, Semi-Active Replication. Springer Verlag. Wien, New York. 1991, pages 116–120.

[Pow91d] D. Powell (Ed). Delta-4: A Generic Architecture for Dependable Computing. *Volume 1 of ESPRIT Research Reports*, chapter 6.6, Passive Replication. Springer Verlag. Wien, New York. 1991, pages 111–115.

[Pow91e] D. Powell (Ed). Delta-4: A Generic Architecture for Dependable Computing. *Volume 1 of ESPRIT Research Reports*, chapter 10.6, Two-Phase Accept Protocol. Springer Verlag. Wien, New York. 1991, pages 282–284.

[Pow92] D. Powell. Failure Mode Assumptions and Assumption Coverage. In *Proceedings of the 22th International Symposium on Fault-Tolerant Computing*. Computer Society Press of the IEEE. Boston, Massachusetts. Jul. 1992, pages 386–395.

[PSL80] M. Pease, R. Shostak and L. Lamport. *Reaching Agreement in the Presence of Faults*. Journal of the ACM. Vol. 26, No. 2, Apr. 1980, pages 228–234.

[PT86] K.J. Perry and S. Toueg. Distributed Agreement in the Presence of Processor and Communication faults. *IEEE Transactions on Software Engineering*. Vol. 12., No. 3, Mar. 1986, pages 477–482.

[PT88] P. Panangagen and K. Taylor. Concurrent Common Knowledge: A New Definition of Agreement for Asynchronous Systems. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*. Toronto, Canada, Aug. 15–17, 1988, pages 197–209.

[PT92] P. Panangaden and K. Taylor. Concurrent Common Knowledge: Defining Agreement for Asynchronous Systems. *Distributed Computing*. Springer Verlag. Nr. 6, 1992, pages 73–93.

[Rea86] J.F. Ready. VRTX: A Real-Time Operating System for Embedded Microprocessor Applications. *IEEE Micro*. Vol. 4, Nr. 6, Jun. 1986, pages 8–17.

[Rei89] J. Reisinger. Failure Modes and Failure Characteristics of a TDMA Driven Ethernet. *Research Report* 8/89, Institut für Technische Informatik, Technische Universität Wien, Austria, Dec. 1989.

[SAE92] 1992 SAE Handbook Volume 2, Controller Area Network (CAN), An In-Vehicle Serial Communication Protocol—SAE J1583 MAR90. Society of Automotive Engineers, 400 Commonwealth Drive PA, USA, 1992, pages 20.342–20.355.

[SB89] S.S.B. Shi and G.G. Belford. Consistent Replicated Transactions. In *Eight Symposioum on Reliable Distributed Systems*. Computer Society Press of the IEEE. Seattle, Washington. Oct. 1989, pages 30–41.

[Sch84] F.B. Schneider. Byzantine Generals in Action: Implementing Fail-Stop Processors. *ACM Transactions on Computer Systems*. Vol. 2, Nr. 2, May. 1984, pages 145–154.

[Sch90] F.B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*. Vol. 22, Nr. 4, Dec. 1990, pages 299–319.

[Sch93a] M. Schneider. Self-Stabilization. *ACM Computing Surveys*. Vol. 25, Nr. 1, Mar. 1993, pages 45–67.

[Sch93b] M.D. Schroeder. State-of-the-Art Distributed System: Computing with BOB. *Distributed Systems*. Second Edition, S. Mullender, (ed.). ACM Press. 1993, pages 1–16.

[Sch93c] W. Schütz. The Testability of Distributed Real-Time Systems. Kluwer Academic Publishers. 1993.

[SGS84] F. Schneider, D. Gries, and R. Schlichting. Fault-Tolerant Broadcasts. *Science of Computer Programming*. Vol. 4, No. 1, 1984, pages 1–15.

[Sie90] Siemens. Microcomputer Components SAB 80C166/83C166 16-Bit CMOS Single-Chip Microcontrollers for Embedded Control Applications. User's Manual. Edition 6.90.

[SK92] S.H. Son and S. Kouloumbis. Replication Control for Distributed Real-Time Database Systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems*. IEEE Computer Society Press. Yokohama, Japan. Jun. 9-12, 1992, pages 144–151.

[SKK+86] K. Shinoda, H. Koide, F. Kobayashi, M. Nagase, S. Ikeda, M. Takata, and J. Nakano. Development of New Electronic Control System for a Diesel Engine. In *SAE International Congress and Exposition*, pages 197–206, Detroit, MI, USA, Feb. 1986. SAE Special publications, Warrendale, PA, USA. SAE Technical Paper 860597.

[SLL86] K.G. Shin, T.-H. Lin, and Y.-H. Lee. Optimal Checkpointing of Real-Time Tasks. In *Proceedings on the Fifth Symposium on Reliability in Distributed Software and Database Systems*. Computer Society Press of the IEEE. Los Angeles, California. Jan. 1986, pages 151–158.

[Sof91] Software Components Group, Inc. *pSOS+/68K User's Manual,* Aug. 8, 1991. KX68K-MAN, Version 1.2.

[SR88] J.A. Stankovic and K. Ramamritham. *IEEE Tutorial: Hard Real-Time Systems.* IEEE Computer Society Press, Washington, D.C., USA, 1988.

[SS83] R.D. Schlichting and F.B. Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computing Systems.* Vol. 1, Nr. 3, Aug. 1983, pages 222–238.

[Sta88] J.A. Stankovic. Misconceptions about Real-Time Computing: A Serious Problem for Next-Generation Real-Time Systems. *IEEE Computer,* Oct. 1988, pages 10–19.

[Sti78] J.J. Stiffler. Fault-Coverage and the Point of Diminishing Returns. *Journal of Design Automation and Fault-Tolerant Computing.* Vol. 2, Nr. 4, 1978, pages 289–301.

[Tan90] A.S. Tanenbaum et al. Experiences with the Amoeba Distributed Operating System, *Communication of the ACM,* Vol. 33, Dec. 1990, pages 46–63.

[Tan92] A.S. Tanenbaum. Modern Operating Systems. section 2.2.8, Message Passing. Prentice-Hall. 1992

[TKB92] A.S. Tanenbaum, M.F. Kaashoek, and H.E. Bal, Parallel Programming Using Shared Objects and Broadcasting, *IEEE Computer,* Aug. 1992, pages 10–19.

[TPS87] S. Toueg, K.J. Perry, and T.K. Srikanth. Fast Distributed Agreement. *SIAM Journal on Computing.* Vol. 16, Nr. 3, Jun. 1987, pages 445–457.

[TS90] A. Tully and S.K. Shrivastava. Preventing State Divergence in Replicated Distributed Programs. In *9th Symposium on Reliable Distributed Systems.* Computer Society Press of the IEEE. Huntsville, Alabama. 1990, pages 104–113.

[TM93] M. Tomasevic, and V. Milutinovic. The Cache Coherency Problem in Shared-Memory Multiprocessors. IEEE Computer Society Press, Jul. 1993.

[TW89] D. Taylor and G. Wilson. The Stratus System Architecture. In *Dependability of Resilient Computers.* T. Anderson, (Ed.), Blackwell Scientific Publications, Oxford, 1989.

[Ver90] P. Veríssimo. Real-Time Data Management with Clock-less Reliable Broadcast Protocols. In *Proceedings of the Workshop on Management of Replicated Data.* Houston, Nov. 1990, pages 20–24.

[VR89] P. Veríssimo and L. Rodrigues. Order and Synchronism Properties of Reliable Broadcast Protocols. Technical Report No. RT/66–89, INESC, Lisboa, Portugal, Dec. 1989.

[VR92] P. Veríssimo and L. Rodrigues. Group Orientation: A Paradigm for Distributed Systems of the Nineties. In *3rd IEEE Workshop on Future Trends of Distributed Computing Systems*. Taipe, Taiwan. Apr. 1992, pages 1–7.

[VRB89] P. Veríssimo, L. Rodrigues, and M. Baptista. AMp: A Highly Parallel Atomic Multicast Protocol. In *SIGCOMM Symposium*, ACM, Austin, USA. Sept. 1989, pages 83–93.

[WBS89] Supply Dynamics in Automotive Electronics. Booz, Allen and Hamilton, Inc. United Kingdom. In *Proceedings of the 20th International Symposium on Automotive Technology & Automation*. Vol. II. Florence, Italy. 29. May–2, Jun. 1989, pages 1355–1369.

[WLG+78] J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostack and C.B. Weinstock. SIFT: The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. In Proceedings of the IEEE. Vol. 66, Nr. 10, Oct. 1978, pages 1240–1255.

[WYP91] K.L. Wu, P.S. Yu, and C. Pu. Divergence Control for Epsilon-Serialisability. Technical report CUCS-002-91, Department of Computer Science, Columbia University, Feb. 1991. Also available as IBM Tech Report No. RC16598.

# Index