Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Selection and Hardware-Implementation of an Efficient Consensus Algorithm for a Mesochronous System

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Alexander Heinisch, BSc.
Matrikelnummer 0627820

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Thomas Polzer

Wien, 4.12.2014 _______________________ _______________________
(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Selection and Hardware-Implementation of an Efficient Consensus Algorithm for a Mesochronous System

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Alexander Heinisch, BSc.
Registration Number 0627820

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Assistance: Univ.Ass. Dipl.-Ing. Dr.techn. Thomas Polzer

Vienna, 4.12.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Alexander Heinisch, BSc.
Hauptstraße 148, 2124 Oberkreuzstetten

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Abstract

This thesis explores the possibility of using a consensus algorithm to replace the traditional, triple modular redundancy scheme for fault tolerance. We start with the presentation of state of the art fault tolerance mechanisms and discuss the advantages and drawbacks of TMR systems. To be able to compare these mechanisms to our approach, we outline the basics of distributed algorithms and discuss different consensus algorithms found in literature. Based on this theoretical background a reduction of TMR and replica determinism to consensus has been created.

A theoretical evaluation of the well known Exponential Information Gathering protocol, the Phase King protocol and the Phase Queen protocol as well as modifications of these protocols to implement them directly in hardware is given. The theoretical results are analyzed in detail and augmented by results from fault injection experiments performed using a software simulator. As the simulator was specifically tailored to incorporate the properties of the target platform, we were able to evaluate their fitness for direct hardware implementation solely based on the simulation results. As the fault injection experiments were designed to violate the fault hypothesis in some cases, the degradation properties of the algorithms could also be analyzed.

The Phase King and the Exponential Information Gathering protocol were implemented on a Field Programmable Gate Array (FPGA) network. To circumvent the single shared clock tree in synchronous systems which introduces a single point of failure we decided to implement the system based on a mesochronous clocking system, namely the Distributed Algorithms for Robust Tick-Synchronization (DARTS) protocol and a fully parametrizable communication buffer supporting metastability free communication in this setting. The implementation of these two illustrate that Byzantine fault tolerance using distributed algorithms is achievable in VLSI circuits.

Kurzfassung

Diese Arbeit analysiert die Möglichkeit die traditionellen, dreifach redundanten Fehlertoleranzmechanismen durch Konsensus-Implementierungen zu ersetzen. Nach einem Überblick über gängige Fehlertoleranzansätze und einer Erörterung der Probleme bei der Verwendung von Triple Modular Redundancy und Replikadeterminismus werden das Consensus Problem und die zugrundeliegenden Modelle erörtert. Anhand der theoretischen Modelle werden verwandte Probleme wie Byzantine Agreement und Reliable Broadcast vorgestellt und eine Reduktion von Agreement mit TMR und Replikadeterminismus zu Consensus präsentiert.

Eine theoretische Analyse bekannter Konsensusprotokolle, nämlich des Exponential Information Gathering, des Phase King und des Phase Queen Algorithmus, sowie deren für Hardware-Implementierung modifizierte Versionen, wird durchgeführt. Die theoretischen Eigenschaften der Protokolle werden im Detail analysiert und mit zusätzlichen Daten aus Fehlerinjektions-Simulationen erweitert. Diese Zusatzinformationen ermöglichen Einblicke in das Verhalten der Protokolle, wenn diese außerhalb der festgelegten Spezifikation ausgeführt werden.

Das Phase King Protokoll und das Exponential Information Gathering Protokoll werden auf einem Field Programmable Gate Array (FPGA) Netzwerk implementiert. Da ein einzelner Clock Tree, wie er in synchronen Systemen gängig ist, ein gewaltiges Fehlerpotenzial aufweist, wurde zugunsten einer Implementierung mittels verteiltem System entschieden. Dies wurde durch Zuhilfenahme des Distributed Algorithms for Robust Tick-Synchronization (DARTS) Protokolls und eines parametrisierbaren Kommunikatonsbuffers, der metastabilitätsfreie Kommunikation in dieser Konfiguration erlaubt, erreicht. Die Implementierung dieser zwei Protokolle zeigt, dass verteilte Algorithmen dafür geeignet sind, ein byzantinisch fehlertolerantes Hardwaresystem zu entwickeln.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Shrinking feature sizes and the continuous increase of clock frequency in current very large scale integrated (VLSI) circuits increase their susceptibility to faults caused by external influences such as electromagnetic interference as well as to internal failures like electromigration [13, Fue12]. The single, shared clock tree inherent to most state of the art systems constitutes a single point of failure [29, SSS07]. Due to the high clock speeds, the distribution of the clock signals has become very challenging [13, Fue12]. Alternate approaches, like globally asynchronous and locally synchronous (GALS) systems [28, Roy03] try to solve this problems by introducing smaller synchronous islands whose design is much easier. Unfortunately the lack of a global clock signal makes communication and fault detection harder to implement. By adding at least some precision guarantees to the clocking system, like in DARTS [14, Fue06] [29, SSS07], the problem of communicating between the synchronous islands becomes easier again.

In [25, Pol09] a metastability free communication scheme for mesochronous systems has been developed to further increase fault tolerance. Since most critical systems must guarantee safe operation over periods of time, transient and even permanent faults of single nodes have to be tolerated by these systems without generating erroneous outputs.

To achieve such a fault tolerant behavior, we use mechanisms known in Distributed Computing and apply them to VLSI circuits. Therefore, we have to map electronic circuits to distributed systems. Since such a mapping is not straight forward [13, Fue12], further investigation using benchmarking and prototype implementations is required.

To achieve fault tolerance, we use consensus algorithms. Distributed computing has a vast set of tools for handling Byzantine faults. [21, Lam82] [4, Bar87] [23, Pea80]) [7, Ber89b] [12, Fis85] [1, Att04]

In the scope of this thesis we implement two consensus protocols on a Field Programmable Gate Array (FPGA) and evaluate their behaviour when subjected to fault injections (both violating and respecting their fault hypothesis). The hardware implementation utilizes the metastability free, mesochronous, point to point communication framework developed by [26, PHS09]. As currently no consensus algorithm optimal in all complexity measures (resilience, message and time complexity) is currently known [1, Att04], we have to analyze their hardware implementation overhead before deciding which one is the most suitable one for our needs. This is achieved by theoretically evaluating the protocols and by simulating them on a PC with a simulation framework exactly capturing the restrictions of the communication subsystem available in our hardware framework.

1.2 Aims and Methodology

The aim of this thesis is to evaluate consensus algorithms known in distributed computing concerning their fitness for hardware implementation. We have selected the *Exponential Information Gathering*, the *Phase King*, the *Phase Queen* algorithm and their *Early Stopping* variants for our analysis. To be able to create a fair analysis, we have placed certain restrictions on the selected algorithms, namely a lock step synchronous execution model as well as using single bit messages only. Related problems to consensus like *Byzantine Agreement* or *Atomic Broadcast* are also presented and their relations to the consensus problem are analyzed. We use this analysis to show that ensuring replica determinism is reducible to consensus (for crash and for Byzantine faults).

Besides the theoretical analysis a simulation based fault injection analysis is performed. We focus on random fault injection experiments which both violate and respect the fault hypothesis, respectively. This enables us to confirm the correct execution in case the fault hypothesis is respected and to get data on the algorithms degradation when violating the hypothesis. Based on the theoretical analysis and the fault injection experiments we can select the algorithm best suited for hardware implementation. Utilizing an FPGA prototype of the selected one, we repeat the fault injection experiments and therefore verify the accuracy of the simulation results.

1.3 Contribution

We have created an exhaustive evaluation of three consensus algorithms and their early stopping variants concerning implementability as VLSI circuits. Besides the theoretical work, a simulation environment capturing the main properties of our hardware model was created from scratch. Using this simulator we were able to experimentally evaluate the resilience and the degradation properties of the algorithms before selecting the one best suited for hardware implementation. The selected algorithm as well as the Exponential Information Gathering algorithm were implemented on an FPGA prototyping platform and evaluated in hardware using fault injection experiments. We were able to show that consensus is a viable alternative to Triple Modular Redundancy and replica determinism in environments where Byzantine faults may occur.

1.4 Structure of the Thesis

The thesis is structured into 4 parts. The first part (Chapter 2) discusses state of the art approaches to increase fault tolerance as well as their advantages and disadvantage. It starts with the definition of basic properties necessary to understand fault tolerance in Section 2.1 and outlines common failure models in Section 2.2. Throughout the Sections 2.3 and 2.4 replica determinism and triple modular redundancy are described, and the problems arising when implementing replica deterministic behavior are outlined. In Section 2.6 methods to create an abstraction of time and the problem of ordering different events are discussed, while Section 2.7 closes the chapter by discussing problems created by variations in the input events.

Chapter 3 constitutes the second part. Our approach of creating highly dependable VLSI circuits using consensus algorithms is outlined. The system model founding the basis for our later evaluation is presented in Section 3.1. Based on this model, the problem of reaching consensus is defined in Section 3.2. Afterwards, enforcing replica determinism is reduced to consensus using the state machine approach in Section 3.3. Section 3.4 gives an overview of related broadcasting problems, like Atomic Broadcast, and their relation with the consensus problem. Afterwards, solutions to reach Byzantine Agreement, a closely related problem to consensus, is considered in Section 3.5. The analysis of selected consensus algorithms is performed in Sections 3.6 and 3.7. While we first concentrate on the basic protocols, Sections 3.8 - 3.12 outline techniques to optimize the protocols for certain scenarios.

The third part of the thesis describes the simulation environment used for the empirical evaluations of the algorihms in Chapter 4. While the Sections 4.1 and 4.2 describe the simulation environment and the framework used to gather the results, Section 4.3 presents the transformation of the Exponential Information Gathering protocol to single bit messages required by our hardware framework. The evaluation of the simulation results and the selection of the protocols to be implemented in hardware is done in Section 4.4.

The fourth part describes common problems when designing hardware and the hardware framework used for the implementation of the protocols (Chapter 5). The design of the implementation of the selected protocols is given in Chapter 6, while Chapter 7 presents the results of the hardware experiments. Finally, Chapter 8 summarizes and concludes the thesis and gives an outline on possible future work.

CHAPTER 2

Fault Tolerance

One of the major characteristics of a safety critical system is the ability to tolerate partial failures. To guarantee correct behaviour at the system level certain parts of the system have to be implemented redundantly and computations have to be split into smaller units operating at different times or in different space domains.

This chapter summarizes the design principles used in safety critical applications as well as the classification of faults and possible solutions for handling them. At the end of this chapter we will outline the major drawbacks of state of the art solutions and we will use the next chapter to present a new solution for satisfying the requirements of highly fault tolerant systems.

2.1 Properties of Fault Tolerant Systems

In this section we present the key concepts and properties of fault tolerant systems. Since it is impossible to guarantee a completely fault free system, certain techniques must be applied to increase robustness and therefore its dependability. Considering non reparable systems, e.g. an autonomous Mars rover or a satellite, a high dependability is of vital importance. Additionally, elaborate mechanisms for diagnosing and debugging have to be designed. Based on their results repair strategies can be executed to return to an operational state.

First, we introduce the partitioning into faults, errors and failures. A fault can either be caused by out of specification operation (e.g. an invalid input signal) or by an erroneous implementation of the specification (e.g. an programming error). Errors are caused by faults and are their manifestation in the system's state. A failure on the other hand is an event at a certain instant of time, where the erroneous system state causes a deviation of the system's specified operation. [19, Kop97] Their relation is also outlined in Figure 2.1.

The design of fault tolerant systems is mostly based on fault containment regions. A fault detected inside such a region is hindered to propagate to other fault containment regions and

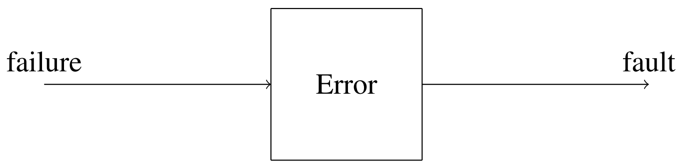


Figure 2.1: Relation between fault, error and failure

therefore a single fault can not affect the whole system. If a detection or removal of a fault is not possible at this level it can be handled at the next higher one, namely the error containment region. Similar to the fault containment region the error containment region detects errors and hinders them to propagate. The highest level is the failure containment region. [19, Kop97]

With these concepts it is possible for a system to handle, tolerate or remove certain faults, errors and failures. The so called fault hypothesis gives assumptions on the type of fault the system can cope with as well as the frequency it occurs. The number of faults that can be detected is measured as the fault detection coverage. All faults not manageable by the system have to be rare events (probability of occurrence has to be below a given threshold) to guarantee dependability. How well the assumptions made in the fault hypothesis capture reality is given by the assumption coverage and it is therefore a qualitative measure of the fault hypothesis. [19, Kop97]

Faults can be grouped into *transient faults*, *intermittent faults* and *permantent faults*. The first class of faults are *transient faults* which occur for short periods of time and then disappear again. Such faults mainly manifest themselves by single bit flips and are often caused by external sources such as electromagnetic interferences, or disturbances in the power supply. Depending on the system, the rate of *transient faults* which can be tolerated can be 10 to 100,000 times higher than the rate of *permanent faults*. [19, Kop97]

The second class, namely *intermittent faults*, contain faults occurring for arbitrary periods of time and then disappear again. These faults are often caused by changes in the system's environment, e.g. a change in temperature. It is quite hard to reproduce and identify this type of fault as the external factors leading to the malfunction may cause the problem only in case they appear in combination. [33, Tan06]

The last class of faults are *permantent faults*. These faults can not be repaired online and maintenance is required. Nevertheless, masking these faults using redundant hardware is still possible in the field.

If a fault can not be repaired online, counter measures depend on the type of systems. We distinguish between fail-safe systems and fail-operational systems. A fail-safe system is a system providing a safe shutdown state which can always be reached. If a fatal error occurs, the system will be switched to the shutdown state. An example of this kind of system is the railway signaling mechanism. If an error occurs, all signals are switched to stop and no train will move any more (= safe shutdown state)

The second class of systems are called fail-operational. In such systems no safe state is present and therefore no possibility of shutdown exists. An example for such a system would be an airplane, where at least basic operation has to be provided at any given point in time to prevent it from crashing.

2.2 Failure Models

The properties presented in Section 2.1 are significant in choosing the right failure model for a system. In general three classes of failures are distinguished in literature [1, Att04]:

1. Crash failures
2. Omission failures
3. Byzantine failures

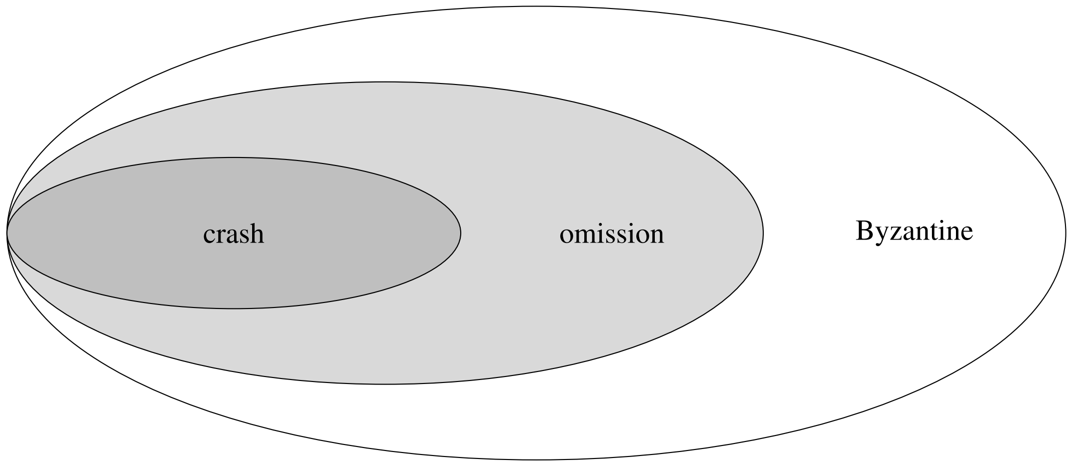


Figure 2.2: Failure models and their relation

In case of crash failures, all subsystems deliver correct results at all times. However, some of them may crash at an arbitrary point in time and stay silent from this point onward. While in a synchronous or partially synchronous setting this class of faults can easily be detected, they can not be handled by asynchronous circuits due to their inherent lack of information delivery deadlines. [12, Fis85]

When considering omission failures the service simply does not respond properly. This can appear in several ways: First the reception of incoming requests may be faulty, thus, the computation unit does not even recognize that it has to respond. In the second case, the unit has computed the results but is not able to communicate it to other units, e.g. because of a broken pin in the communication interface. The third variant of this failure class are all kinds of delays that are not related to the communication subsystem, e.g. a long lasting calculation or an infinite loop.

The strongest and most malicious class of failures is called arbitrary failures or Byzantine failures. Failures of this class can manifest themselves in any imaginable way. Processors or computational units suffering from Byzantine faults may not behave according to their specification or algorithm at all. They can transmit arbitrary messages at any point in time, produce random values or it can even happen that multiple, Byzantine faulty subsystems work in tandem to increase their effect on the whole system.

As outlined in Figure 2.2, the crash failure model is the weakest one and can be seen as a subset of the omission failures that again constitute a subset of the Byzantine failure model.

2.3 Replica Determinism

As discussed before, an increase in dependability of a system is possible if we replicate nodes in certain domains. This can be done at the hardware as well as the software level. Methods to redundantly execute an operation can be established in the time domain, by executing an algorithm multiple times, or in the space domain, such that a computational task takes place on multiple nodes. When it comes to multiple executions, different implementations of a given specification are often used to detect mistakes in the specification in an early stage and to avoid common implementation errors within the system.

A key property of a replicated system is its type of standby operation. Basically we distinguish between active and passive redundancy. Both types are used to enable the system to operate correctly in case of failures by using additional hardware. When using active redundancy all replicas of a component compute the same functions in parallel and their results are subjected to voting and classified as correct or incorrect (e.g. Triple Modular Redundancy). Components computing incorrect results are shut down by the voting logic. Passive redundant designs also offer replicated components but the operations are only computed by one of them. If faults are detected within this component the system switches to another fault free replica. A vital challenge in this case, is to update the replica to the last correct state of the faulty one. A mix of these two are standby-redundant systems. These, like the designs presented before, use replications of their components. All replicas compute the same operation, and therefore have the same history state. In contrast to an active system the output is only taken from one replica. In case of faults the output is taken from another, correct replica. Another special case of replicaton is the N+1 redundancy. Here only one spare component is available to take over the operation of a faulty component out of a total number of N components. Thus, in the N+1 design one fault can be tolerated.

The most common forms of redundancy are *hardware redundancy*, *software redundancy*, *information redundancy* and *time redundancy*. Hardware redundancy addresses mechanisms like the use of multiple sensors monitoring the same entity where the sensors values are subject to a voting process to set an actuator. Here, if one sensor fails, other sensors' values can be used to calculate correct results. Another type of hardware redundancy are methods to spacially distribute replicas. Software redundancy is established by computing results with certain implementations of an algorithm generated from the same specification. With this concept we can develop designs prone to errors introduced by unclean programming, or by missleading assumptions of the specificaton. Therefore, several independent teams develop similar functionality according to the specification with different program constructs, algorithms as well as different programming languages, thus reducing the probability of identical software faults propagating throughout the system. For evaluation and comparison of the different versions it is crucial to specify the input and output formats, the comparison algorithm used as well as the functionality of the used algorithm. This N-Version concept introduced by software redundancy can also be applied to hardware concepts, namely distinct functional redundancy. In this case similar functionality is achieved by using different physical concepts, like mechanical and hydraulic braking mechanisms within a car e.g. Information redundancy addresses replication of transferred data, checksums and error detection or error correction used.

Typical codes range from simple parity codes over CRC codes to cryptographic hash functions. The selection of the used code depends on the available computational resources, the length and format of data as well as the capacity of the communication system itself, and the type of faults to be detected or even corrected at the receiver. The major drawback of all of the above mentioned kinds of redundancy are the increased hardare overheads. In systems where hardware capacity is quite limited e.g. by spacial constraints or unit costs, these replication strategies do not suite well. Time redundancy on the other hand computes the same functionality upon the same input data multiple times on the same component. The computations are, however, shifted by a defined offset in time. This, by definition, doesn't increase the robustness of the component with respect to permanent faults since the same component is used for each time replicated computation, but suits well in environments suffering from frequent transient faults. Another aspect of time redundancy is to classify a failure as transient or permanent, and thus supporting a quite strong indication whether a component has to be disabled permanently or can be kept online for later use. Which kind of replication has to be used and further if one type of replication is sufficient to cover the fault hypothesis depends on the system requirements and its constraints, and has to be decided and evaluated in an early stage of the system design.

The methodology to accomplish correct behaviour by replicating operations in a deterministic system is called *replica determinism*. According to the definition given in [24, Pol93] correct replicas within the same group of replicated components result in the same output in the value as well as the time domain if and only if the computation is started from the same initial state on by executing the same input requests. Same output in the value and time domain hereby means within the specificaton. Thus, differing input and output formats are tolerated within defined ranges. Also the timeliness does not mean that outputs have to be generated at exactly the same instant of time, but within a specified range of time. Problems the designer of a distributed system has to face are those leading to inconsistent behaviour or even non-determinism. Such undesirable behaviours can be forced by *inconsistent inputs*, *inconsistent order*, *non-deterministic* *constructs*, *inconsistent scheduling decisions*, *inconsistent interpretation of time* and *inconsistent interpretations of values* [24, Pol93].

1. Inconsistent inputs: When our system has to cope with inconsistent inputs, such as analogue sensor values, the outputs can be different. Assuming two temperature sensor, it is quite realistic that one sensor reports $20°C$ while another reports $21°C$ due to different calibration, spacial distribution or restrictions in the representation of the measured value.

2. Inconsistent order: Assume two non-commutative messages, node A receives message 1, which says activate the thrust reverser for landing, before message 2 which says increase engine power for a go around, while node B receives them in the opposite order.

3. Non-deterministic constructs: Also constructs which cannot be assumed to deliver equivalent responses on different nodes, such as a true random number generators or complicated high level constructs depend on internal node information. The problem here is, that at any two replicas we cannot predict what the exact state of the other replica is after execution of such constructs.

4. Inconsistent scheduling decisions: Assume two tasks which can not be commutatively executed on two nodes being preemted on one of them, while executed on the other. Assuming two nodes A and B. For simplicity A is adding a natural number x to a value v stored in internal memory and B is multiplying v by a natural number y. Replica 1 schedules A and then B while replica 2 schedules B and then A. It is easy to see that this leads to $(v + x) * y$ on replica 1 while it leads to $v * y + x$ on replica 2.

5. Inconsistent interpretation of time: Since we have limited computational and storage capacity we cannot provide a dense time base like real time. Assuming a decision dependent on the local clock of each node while one node's clock will say it's 12:00 another node's clock will pretend it's 11:59. If the decision is something like shut down the engine at midday but run the engine before, the two nodes will compute different decisions whether the engine should run or shut down.

6. Inconsistent interpretation of values: The same number has to be interpreted equally on all nodes regardless whether the architecture used is big or little endian. Further, since irrational numbers cannot be represented right in the computer rounding and exponent/mantissa interpretation have to be consistent among the nodes.

While some of these inconsistencies can be eliminated at design time others have to be handled online. Depending on the system two possible approaches can be distinguished to solve non-determinism at runtime. On one hand a *strictly centralized or asymetric* approach can be used. In this case each group of nodes chooses a dedicated leader. This leader accepts incoming requests and distributes them among the other nodes in its group. It is its responsibility to correctly pass the requests, to resolve inconsistencies and to respond to external clients. The drawbacks of this approach are that the leader-node is a single point of failure and a clear hierarchical structure within the group has to be enforced. Further, it is quite hard to guarantee

Byzantine fault tolerance in this model, since a faulty leader can fool the other nodes within its group easily. The second approach is the *distributed or symmetric* scheme. In this approach no dedicated leader exists and therefore no single point of failure. Inconsistencies have to be resolved distributedly with some kind of agreement protocol. [24, Pol93]

In the following sections we will discuss the handling of order, time and inputs in more detail.

2.4 Triple Modular Redundancy

Dependability can be increased by using more reliable (which is closely related to more expensive) components or by designing the system in a redundant way. One well known approach is *Triple Modular Redundancy* (TMR), where each node is replicated three times. Each replica is followed by a voter network which calculates the output value by, e.g. choosing the majority of the node values. The complexity of this voter network depends on the dependability constraints and the chosen fault hypothesis. In the simplest implementation, a two out of three majority voter, its complexity is much lower than the complexity of the nodes (e.g. microcontrollers) themselves. Based on this observation it gets apparent that in such settings a single voter per stage may become insufficient to satisfy the dependability criteria. However, if more ambitious voting mechanisms are required their complexity can force the designer to replicate the voters themselves eliminating the single point of failure introduced by the single voter. The number of required replications for each component and the number of required voters per stage mainly depends on the systems dependability requirements, as well as the robustness of the used components. An introduction on how to calculate the required probability measures based on the components dependability can be found in Section 2.5.

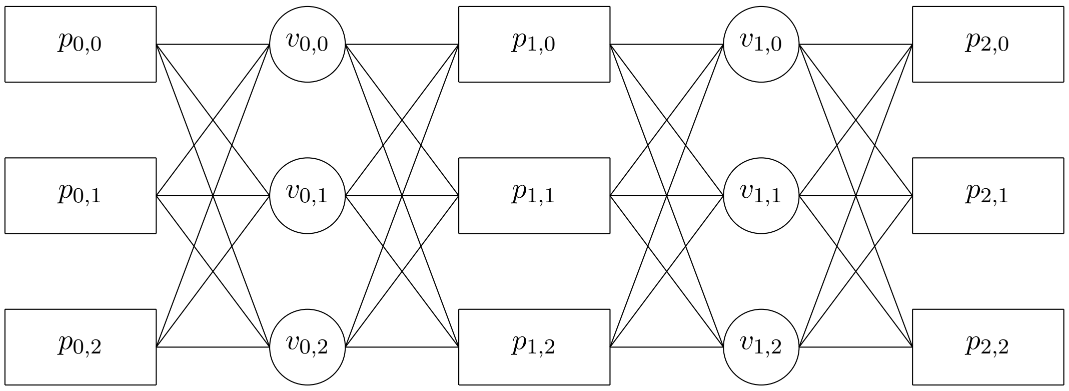


Figure 2.3: TMR network

An example for a TMR network is given in Figure 2.3. In this work we focus on the commonly used TMR design which uses three nodes and three voters. While the first level of nodes ($p_{0,*}$) works with the respective input values (the result of a single temperature sensor, e.g.),

Component	Value to propagate		Descriptions
	binary	decimal	
$p_{0,0}$	1000 0000	128	
$p_{0,1}$	0111 1111	127	
$p_{0,2}$	0000 0000	0	sending to $v_{0,0}$
$p_{0,2}$	1111 1111	255	sending to $v_{0,1}$
$p_{0,2}$	1111 1111	255	sending to $v_{0,2}$
$v_{0,0}$	0000 0000	0	
$v_{0,1}$	1111 1111	255	
$v_{0,2}$	1111 1111	255	

Table 2.1: Inexact input data with one faulty node $p_{0,2}$

the second level nodes ($p_{1,*}$) use the already replicated and voted results of the first level. The last, in our example the third level, calculates the final result usable for directly manipulating an actuator, e.g. As between each two levels of nodes a voter is used, single faults in a lower level are masked out safely before the values are forwarded to and processed by the next higher one. While the complexity of the nodes is dependent on the implemented application, the complexity of the voters is mainly influenced by the value domain they are operating on. The easiest way of voting is by comparing the input values bit for bit and propagating the majority value for each single one. A more challenging task is to compute the majority on a multi bit value like a temperature, if small deviation of the values have to be tolerated. In this case a simple bitwise comparison is insufficient, since a single bit missmatch in the value domain could lead to large deviations in the outputs of the voters (as shown in Table 2.1).

In the example setting introduced in Figure 2.3 the use of an exact voter on non replica deterministic inputs may enable an error to propagate through the whole system. As outlined in Table 2.1 a single fault can cause $v_{0,0}$ to believe the correct value to be $00000000_b = 0_{10}$. At the same time the faulty node $p_{0,2}$ can manipulate $v_{0,1}$ and $v_{0,2}$ into believing that the correct value is $11111111_b = 255_{10}$. Both results are far from the correct inputs $10000000_b = 128_{10}$ and $01111111_b = 127_{10}$. Further, the erroneous value may propagate through the whole TMR network even if $p_{0,2}$ remains the only faulty component.

To avoid this scenario certain voting techniques have been developed. In the scope of this thesis, we will discuss different voting techniques as well as their advantages and drawbacks. The first technique is called *exact voting*. When we use *exact voting* we have to guarantee exactly equal outputs of all replicas at the time of voting. Thus, replica determinism has to be enforced among the replicas to guarantee correct results. The major benefit of exact voting is the simple implementation of the voter itself, since it only has bitwise comparisons of its inputs. Nevertheless, as we will see in Section 2.3, in some cases replica determinism is quite hard or even impossible to achieve, thus, another voting technique has to be applied in these cases. The second technique presented here is *inexact voting*. In contrast to exact voting, *inexact voting*

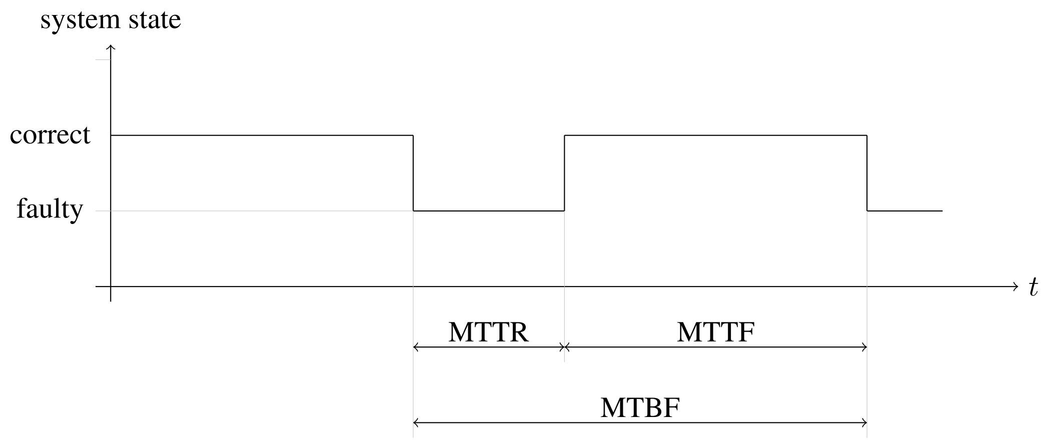


Figure 2.4: Relation between MTTF, MTTR and MTBF

only forces the deviation of the correct values to be at most a specified range apart. Compared to exact voting, the voter needs more information about the value domain and what semantics they represent.

2.5 Dependability Measures

To quantitatively evaluate the dependability of a system we need to introduce statistical measures. Most literature distinguishes between *Availability*, *Reliability* and *Maintainability* [27, Rou99] [19, Kop97] as the key concepts of building dependable systems. While the availabilty gives an estimate on the probability that a system is usable at any given time, the reliability describes the estimation on how probable it is that a system is still operational after a given timespan. Beside these two, maintainability is a measure on how easy it is to repair a corrupted or a failed system.

Reliability reflects the component robustness and the quality of design. The *Mean Time Between Failures (MTBF)* is the timespan between the occurrence of two failures, which is given by the sum of the *Mean Time To Failure (MTTF)* and the *Mean Time To Repair (MTTR)* as depicted in Figure 2.4. As we can see the effort taken to guarantee a certain reliability measure is dependent on the time without any failures on the one hand and the possibility for repair on the other hand. The MTTF gives the average time a component can be operated correctly. It is assumed that the component is fully operational at the begin and after each repair action. The MTTR gives the mean time needed to repair a component by divison of the total time needed for repair actions by the number of components repaired. The reliability $R(t)$ can be directly calculated from the MTBF for repairable components and from the MTTF for unmaintainable components. The inverse of the MTBF is the failure rate λ which specifies the frequency of failures occurrences. [19, Kop97]

$$R(t) = e^{-\lambda t} = e^{-\frac{t}{MTBF}} \tag{2.1}$$

$$R(t) = \sum_{i=k}^{n} \binom{n}{i} (e^{-\lambda t})^i (1 - e^{-\lambda t})^{n-i} \tag{2.2}$$

The Equation 2.1 specifies how to calculate the reliability of a system without replicated components, while Equation 2.2 describes the reliability of a system where k replicas are assumed to work correctly at any given point in time.

Another very important property is the systems availability which describes the confidence that a system is operational at any given time. The availability therefore is dependent on the *Mean Time To Failure (MTTF)* and the *Mean Time To Repair (MTTR)* as outlined in Equation 2.3. The longer the maintenance action lasts, the higher is the MTTR and thus the lower the availability of the system under consideration becomes. Thus, for a system which can not be repaired, the availability is 0.

$$A = \frac{MTTF}{MTTF + MTTR} \tag{2.3}$$

The MTTF is mainly affected by are *Design Failures*, *Infant Mortality*, *Random Failures* and *Wear Out*.

1. Design failures are failures due to deficiencies during the system design phase. In a well designed system this type of failures should have very low influence on the systems availability. The influence of this class of failures can be minimized by appropriate testing, validation, simulation and certain review phases during product development. [31, Sin06]

2. Infant mortality addresses faulty behaviour introduced by manufacturing problems like leaking capacitors or poor soldering. This kind of faults can be detected by certain tests such as burn-in testing to emulate normal operation before the system is in a productive environment (e.g. with a testbed developed by an independent team), power cycling to simulate intermittend states and memory effects at startup, temperature cycling and other physical impacts to stress the mechanical limitations of the hardware. [31, Sin06]

3. Random failures can occur at any time of the systems lifetime, and thus fault-correction methods like replication have to be considered to increase the systems availability. [31, Sin06]

4. Wear out addresses degradation of hardware components leading to faults. This class of failures can only be managed by replicating the hardware and introducing standby systems or by continuous maintenance of the components. [31, Sin06]

All of these classifications can be visualized by the bathtube curve. Other than the reliability, the bathtube distribution determines the probability that a component fails exactly at a given time t. The curve outlined in Figure 2.5 shows a sample plot of the bathtube curve and shows the relations to the above classifications. Design and manufacturing failures occur in the early stages of the systems lifetime, while random failures can occur at any time. The longer the system is in use the higher the chance for components to wear out more and more becomes and thus the probability of faults increases again at the end of the systems lifetime.

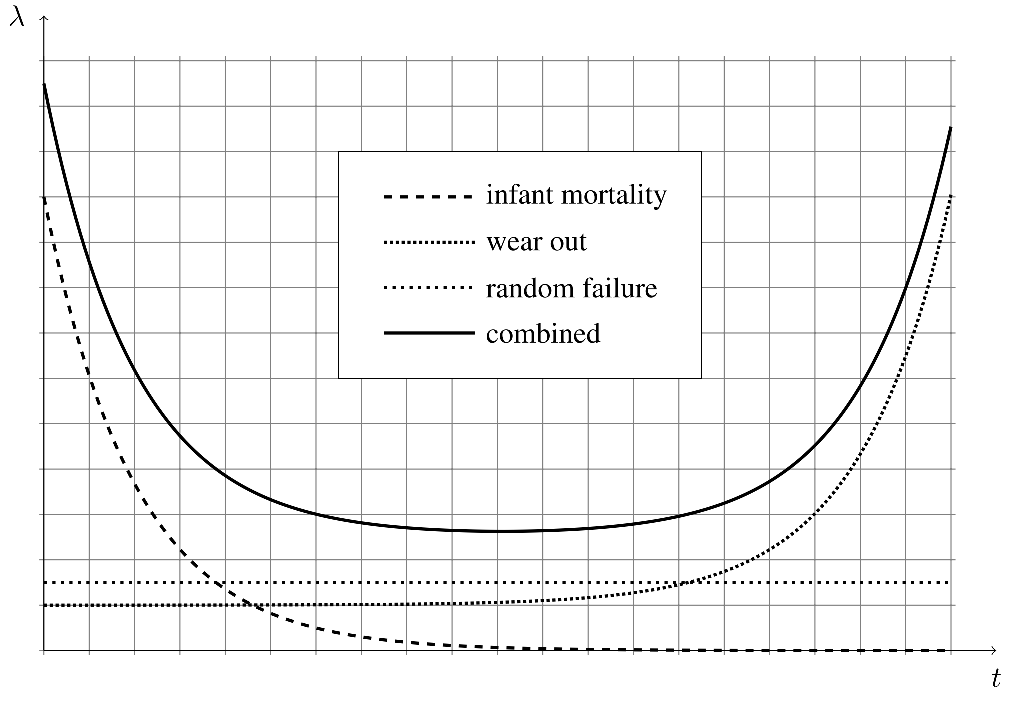


Figure 2.5: Bathtube curve: Showing the failure rate λ of a hardware system over time

According to the reliability of the different subcomponents, the probability on how many faults have to be tolerated by the system can be estimated. A system withstanding k faults is called k-resilient or k-fault tolerant. By replicating the nodes of the system the resiliency can be increased. According to the classification in Section 2.2 we can distinguish three major cases for TMR systems. To handle crash-failures the system has to provide $f + 1$ replicated components whereof at most f components are allowed to get faulty. Trivially this constraint is needed, since otherwise in the case of faults no output could be generated at all. For omission faults at least $2f + 1$ replicated components are needed. Since in the case of omission faults the timeliness of the components values cannot be guaranteed we need at least a majority of correct values at the voter to differentiate an outdated value from a correct value. Lastly, the most challenging class, the Byzantine faults, require $3f + 1$ replicas. [19, Kop97]

2.6 Event Order

As already pointed out inconsistent interpretation of the event order can destroy replica determinism. Therefore, it is important to guarantee a consistent event order in a distributed system. To achieve this, several mechanisms are known and summarized in this section. The two most important concepts for achieving event order in distributed systems are physical clocks and vector clocks.

Physical Clocks

A common knowlege of time simplifies many problems in a distributed computing system. When talking about time, however, in most cases some kind of clock or timer is meant. These devices fire an interrupt or increment the value of a register after a given timespan of real time t. Due to variations in the underlying physical system, like material or temperature variations, the value clock(t) is only an approximation of real time. A measure of this approximation's quality is given by the drift rate of the clock ρ which is described in Equation 2.4. An example for the clock drift is given in Figure 2.6

$$(1 - \rho)t \leq clock(t) \leq (1 + \rho)t \tag{2.4}$$

According to this Equation the value of any two clocks $clock_i$ and $clock_j$ may diverge over time by a factor of $\rho_i + \rho_j$, where ρ_i and ρ_j specify the drift rates of the two clocks, respectively. The maximum drift of all clocks in the ensemble at any given time is called the precision of the system Π. To guarantee a non infinite precision we have to define a convergence function to counteract the missalignment of the clocks. This convergence function Φ specifies the maximum offset of any two clocks after the resynchronization while the drift offset $\Gamma = \max_{\forall i,j}(\rho_i + \rho_j)\delta_{sync}$ describes the maximum drift of any two clocks within a given resynchronization interval δ_{sync}. To guarantee a system wide precision Π Equation 2.5 must hold. [19, Kop97]

$$\Phi + \Gamma \leq \Pi \tag{2.5}$$

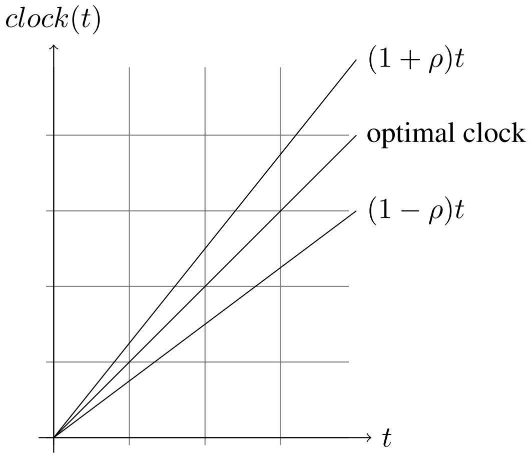


Figure 2.6: Drift rate

Since the synchronization of the clocks in a distributed system can only be achieved via communication of the nodes, another important role on the precision of a clocking system is the maximum latency jitter ϵ of the communication system. The lower bound on the precision (see Equation 2.6) of a system consisting of N nodes with a worst case latency of ϵ was given by [22, Lun84].

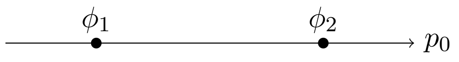
$$\epsilon(1 - \frac{1}{N}) \leq \Pi \tag{2.6}$$

On a single node the ordering of events can be handled quite easily by combining each event with a local timestamp. In a distributed system this is however not the case since, even with an accurate clock synchronization, an event may have timestamps differing by Π on different nodes, which makes ordering events a complex challenge.

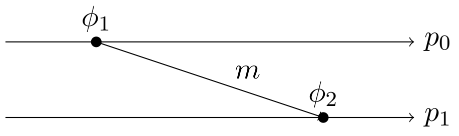
One method to order events on different nodes is to apply an agreement protocol whenever an event occurs, which has the drawback of additional communication and processing overhead. Another approach is to introduce a sparse time base, splitting the real time (which is a dense time base) into intervals of silence where no event is fired and intervals where all occurring events are handled as concurrent. While the first approach is quite unrestrictive, the second approach can only handle events in the field of system control.

Vector Clocks

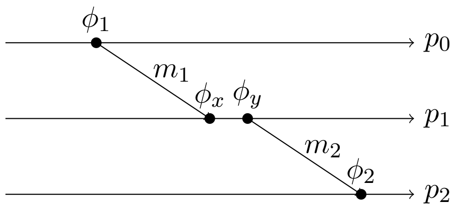
Intuitively an event ϕ_1 is said to happen before an event ϕ_2 if it occurs earlier in the context of real time. As we have seen in the previous section consistently capturing time in a distributed system is quite hard. To circumvent some of the problems we rely on Lamport's happened before



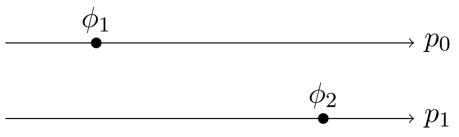
(a) ϕ_1 and ϕ_2 occur on the same node and ϕ_1 occurs first



(b) ϕ_1 is a send event of a message m and ϕ_2 is the corresponding receive event of m



(c) There exists an sequence of events ϕ_1, ϕ, ϕ_2 such that $\phi_1 \overset{\alpha}{\Rightarrow} \phi$ and $\phi \overset{\alpha}{\Rightarrow} \phi_2$



(d) There is no happens before relation between ϕ_1 and ϕ_2

Figure 2.7: Examples for the happens before relation

relation [20, Lam78], which is even valid in fully asynchronous system. The relation defines:

An event ϕ_1 happens before ϕ_2 in an execution α denoted by $\phi_1 \overset{\alpha}{\Rightarrow} \phi_2$ if either

1. ϕ_1 and ϕ_2 occur on the same node and ϕ_1 occurs first
2. ϕ_1 is a send event of a message m and ϕ_2 is the corresponding receive event of m.
3. or if there exists a sequence of events ϕ_1, ϕ, ϕ_2 such that $\phi_1 \overset{\alpha}{\Rightarrow} \phi$ and $\phi \overset{\alpha}{\Rightarrow} \phi_2$

Examples of the happens before relation as described above are depicted in Figure 2.7. Subfigure 2.7d specifically outlines that two events on different nodes cannot be ordered with the happens before relation if there is no communication between the nodes where ϕ_1 and ϕ_2 happen.

Implementing logical clocks is one way to realize the happens before relation. Thereby each node in the system has its own logical clock LC_i which is updated with every single event occurrence. To every message the logical clock of the sending node is appended. If a local event takes place at node i its logical clock LC_i is incremented by one. Otherwise, if a message from node j has been received at node i, LC_i is set to the maximum of both clocks incremented by one. [20, Lam78]

Theorem 1. *Assume two events ϕ_1, ϕ_2 and let α be an execution segment such that $\phi_1 \overset{\alpha}{\Rightarrow} \phi_2$ holds, it follows $LC_*(\phi_1) < LC_*(\phi_2)$. [20, Lam78]*

In contrast to Theorem 1 the inverse arguement is not valid because we mapped the partial order of the happens before relation to the total order of the natural numbers. By establishing a partial ordering of logical clocks, vector clocks are a tool supporting both directions. [1, Att04]

In the case of vector clocks each node maintains a vector $\vec{v_i}$ holding n natural numbers. The i^{th} entry of vector $\vec{v_i}$ represents the logical clock of node i. Initially the logical clocks are set to 0 and are incremented on each event occurrence, such as the reception of a message, the sending of a message or a local event, occurring at node i. The vector $\vec{v_i}$ is appended to every message sent from node i to node j. Node j updates its local vector $\vec{v_j}$ with the informations of $\vec{v_i}$ by simply setting each entry of $\vec{v_j}$ to the maximum of the corresponding pairs of logical clocks as depicted in Equation 2.7.

$$\forall x, 0 \leq x < n : \vec{v_j}[x] = \max(\vec{v_i}[x], \vec{v_j}[x]) \tag{2.7}$$

Theorem 2. *For every admissible execution, for all nodes i and j, $\vec{v_j}[i] \leq \vec{v_j}[j]$ holds. [1, Att04]*

To achieve partial ordering we have to define how to compare two vector clocks. $\vec{v_i}$ is said to be *smaller or equal* compared to $\vec{v_j}$ ($\vec{v_i} \leq \vec{v_j}$) if $\vec{v_i}[x] \leq \vec{v_j}[x]$ for all $0 \leq x < n$. Further, $\vec{v_i}$ is said to be *smaller* compared to $\vec{v_j}$ ($\vec{v_i} < \vec{v_j}$) if $\vec{v_i} \leq \vec{v_j}$ and $\vec{v_i} \neq \vec{v_j}$.

Theorem 3. *Assume two events ϕ_1, ϕ_2 and let α be an execution segment such that $\phi_1 \overset{\alpha}{\Rightarrow} \phi_2$ holds, it follows $\vec{v_*}(\phi_1) < \vec{v_*}(\phi_2)$. [1, Att04]*

Theorem 4. *Assume two events ϕ_1, ϕ_2 and let α be an execution segment such that $\vec{v_*}(\phi_1) < \vec{v_*}(\phi_2)$ holds, it follows $\phi_1 \overset{\alpha}{\Rightarrow} \phi_2$. [1, Att04]*

Theorem 5. *If $\vec{v_*}(\phi_1)$ and $\vec{v_*}(\phi_2)$ are incompareable in execution α, that is $\vec{v_*}(\phi_1) \nleq \vec{v_*}(\phi_2)$ and $\vec{v_*}(\phi_2) \nleq \vec{v_*}(\phi_1)$, ϕ_1 and (ϕ_2) are said to be concurrent, denoted by $\phi_1 ||_\alpha \phi_2$.[1, Att04]*

According to Theorems 3, 4 and 5 the happens before relation can be completely captured with vector clocks.

2.7 Input

We have shown how system dependability is linked to the reliability of single units of the system. As sensors are quite unreliable compared to e.g. microcontrollers, these parts are implemented redundantly in most cases. The analog values of many of these sensors originate from a continuous value domain, and the resolution of values in the system is limited. Thus, the values have to be converted from a continuous value domain to a discrete one. This transformation can lead to deviations in the measured values called digitalization or quantization error. Besides, the deviations of the captured values can lead to different measurements of the same object by two sensors. Thus, even in the fault free case, some kind of agreement strategy has to be applied to achieve replica determinism.

Since a per bit comparison is not possible in all cases, voting strategies have to be applied to reach agreement on the sensor values. The selected voting strategy mainly depends on the type of agreement, where *syntactic agreement* and *semantic agreement* can be distinguished. We say the system agrees syntactically on the sensor values, if the decision takes place without interpretation of the values and its possible contexts according to the system state. An example for voting would be to take the average over all input values. A single fauly value can however influence the solution very heavily in this case. When talking about semantic agreement, the agreement algorithm interprets the values passed to the voters. Context information can help the algorithm to filter out erroneous values, e.g. values exceeding their natural bounds. The algorithm can furthermore combine the inputs of different sensors and check their validity according to the laws of nature. Thus, it is not necessary that every sensor of the network measures the same entity. [19, Kop97]

CHAPTER 3

Consensus

In the previous chapter we have seen that for reaching a certain reliability in a distributed system, protocols are needed that allow the system to function properly even if a limited number of computational units act faulty. To achieve this goal, we have to guarantee cooperation of the non faulty nodes of the system to detect and handle the misbehaviour of the faulty ones and commonly agree on the same information. At a first glance this can be implemented quite easily by a majority voting over all the known values. If more powerful fault models are used, however, the problem can not be solved so trivially. Since in this case even one single fault can lead to disagreement and therefore in close elections more intricate mechanisms have to be used. How to solve these kinds of problems is subject of this chapter with the main focus on the consensus problem and a selected subset of its solutions. [11, Fis00]

In the following we will present algorithms and primitives providing agreement among a group of nodes. We will use the system model introduced in Section 3.1 for the algorithms and the analysis. The algorithms presented are designed to run on distributed systems consisting of self-contained nodes communicating via a fully connected message passing system. The impact of faults is modeled using an adversary, an omniscient, imaginary entity capable of injecting faults into the system.

3.1 Formal Model

In our model, a system consists of a set Π of n nodes p_i ($i \in \{0, 1, ..., n-1\}$) each executing an instance A_i^n of algorithm A and a fully connected communication system (for details see Figure 3.1).

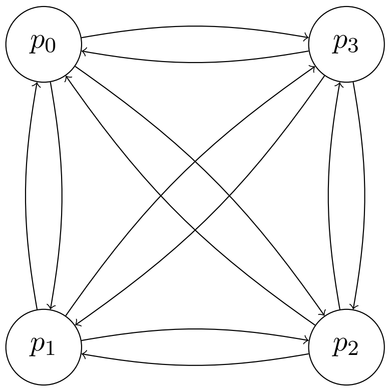


Figure 3.1: Fully connected communication network

Since communication between the nodes takes place via message passing only (realized by $n(n-1)$ distinct channels) the algorithm A_i^n has only access to the nodes p_i internal state and the input and output buffers of p_i. $outbuf_i[j]$ thereby is the output buffer of node p_i, containing all sent but not yet delivered messages from p_i to p_j. On delivery, a message is copied from $output_i[j]$ to $inbuf_j[i]$ and therefore becomes accessible for node p_j. When node p_j reads (consumes) a message, it will be removed from its input buffer. For details on the message buffers see Figure 3.2.

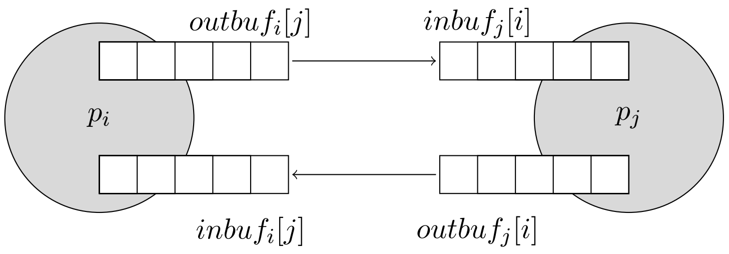


Figure 3.2: Message buffers for incoming and outgoing messages

Each node p_i can be modeled as a state machine Q_i where $q_i \in Q_i$ represents the nodes current state including all message buffers ($inbuf_i[*]$ and $outbuf_i[*]$). While q_i is the local state for node p_i a configuration $C = \{q_0, ..., q_{n-1}\}$ represents a consistent state of the whole system. The transition between two different configurations can either be a computation event ($comp_i$) corresponding to a state change of p_i's local state machine Q_i or a message delivery event ($del(i, j, m)$) forwarding a message from $outbuf_j[i]$ to $inbuf_i[j]$. An alternation of configurations C and events ϕ is called an execution α as depicted in Figure 3.3. [1, Att04]

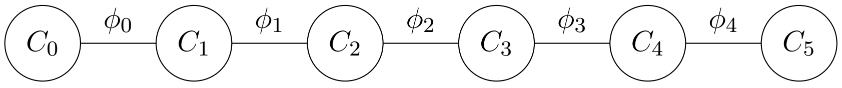


Figure 3.3: Execution α

An execution is admissible, if it satisfies a given set of liveness and safety properties. A safety property guarantees that nothing bad will happen, while on the other hand, a liveness property states that eventually something good will happen. [1, Att04]

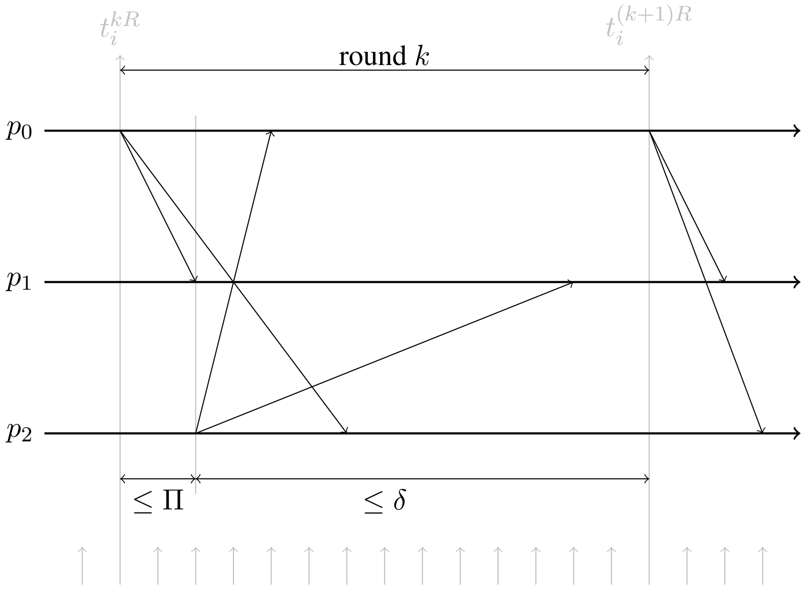


Figure 3.4: Example for the execution of a distributed algorithm in the lockstep synchronous model

Possibly, the most restricting property of a formal model is the way in which communication may take place. The most prominent models in this regard are the asynchronous, the synchronous and the lockstep synchronous model. In a fully asynchronous system the timely behaviour of the system is completely unrestricted. With respect to message passing systems, this means no assumptions on the time needed to send a message m from one node to another can be given. It is not possible to distinguish between two configurations where a message m sent by node p_i is not yet delivered at time t or p_i has, due to faults, not sent m at all. The synchronous model assumes that there exists some kind of event, namely a tick, after which a certain process of computation will always be executed. The transfer of a message m from one node to another is bounded by a given latency Δt_m and therefore its delivery can be guaranteed to be in the interval $[0, \Delta t_m]$. If a message m has not arrived at time Δt_m, it is save to assume that it has never been sent or has been lost. Our system model is assumed to be lockstep synchronous. This means that a single computation step takes place in zero time and that all communication is delivered between two successive impulses triggered by a distributed synchronization mechanism accessible by all nodes. The interval between the r^{th} and the $r+1^{st}$ impulse is called the r^{th} round. We assume in the lockstep synchronous model that all messages sent by node p to node q in round r are delivered at the beginning of round $r+1$. An example for an execution of a distributed algorithm in the lockstep synchronous model is given in Figure 3.4. As we can see, the lockstep synchronous model can be implemented on top of the synchronous model by selecting sufficient clock ticks to ensure the save delivery of all messages within one round. Therefore, the beginning of two rounds must be at least $R \leq \frac{\Pi+\delta}{1-\rho}$ ticks apart, where Π gives the precision of the clocking system as described in Section 2.6 and δ defines the known upper bound on the transmission delay. Therefore, we can savely trigger round k when the kR^{th} tick t_i^{kR} occured at node p_i.

Beside our distributed system, there exists an omniscient, imaginary adversary controlling the whole system. It is only restricted by the fault hypothesis and the system model. Using these powers, the adversary can alter message scheduling, e.g. the time a message is delivered, node scheduling, e.g. when certain events are triggered, and can select nodes which will suffer from faults and can define when these faults occur. For example in a Byzantine fault aware system, tolerating f-faulty nodes, the adversary may cause up to f nodes to crash or execute arbitrary code and even worse the adversary can cause the faulty nodes to work together emerging their mightiness. In systems using cryptography it is assumed that messages are not altered arbitrarily by the adversary. In this model the computational power of the adversary is restricted and thus it can be assumed that spurious messages can be identified by correct nodes. Therefore, protocols using cryptography can perform better than protocols not using cryptography. In the scope of this thesis the further investigations are restricted to protocols not using cryptography.

We say node p_i to be correct if it executes according to A_i^n. Otherwise p_i is called faulty. Note that the classification of a node as faulty is assumed only valid during a single execution of the protocol, since otherwise temporary faults would be classified like permanent faults. Protocols withstanding f faulty nodes are called f-resilient.

To be able to qualitatively compare protocols, we introduce *resilience*, *message complexity*, *bit complexity* and *time complexity* as measures. We will always evaluate them in worst case executions to get the practically relevant bound for the protocols. We say a bound is tight, if there exists an execution hitting the bound while proving that no lower bound for this scenario can be found. The measures used to compare the protocols are defined as follows:

1. Resilience states how many faulty nodes f a protocol can tolerate in a system consisting of a total of n nodes, while all liveness and safety properties must be satisfied.

2. Message complexity gives an upper bound on the number of messages sent during a single execution of the protocol. The size of a single message has no influence on this measure.

3. Message size gives an upper bound on the size of a single message.

4. Bit complexity gives an upper bound on the number of bits that have to be transferred during an execution of the protocol. Note: Since the number of messages and the size of the messages may vary from round to round, the bit complexity gives, in contrast to the message complexity, the exact number of bits transferred.

5. Time complexity gives an upper bound on the number of communication rounds the protocol needs until each node successfully terminates, while ensuring the liveness and safety properties.

3.2 The Consensus Problem

In a distributed system there are many scenarios in which all the nodes p_i, each holding a private value x_i, have to agree consistently on a single value y (agreement) within finite time (termination). For the operation to be considered correct, y must be one of the values $(x_0, ..., x_{n-1})$ (validity). This is quite simple in a reliable system where all nodes and the communication system behave correctly. As soon as our assumption (which is quite strong) does not hold, reaching consensus in arbitrary execution scenarios is not that easy anymore. Depending on the underlying fault model, (for details about these classifications see Section 2.2) protocols have been developed in the last decades to solve this challenge. In this section we will outline the common properties of consensus algorithms, and some impossibility results already proved in literature. Protocols solving the consensus problem under Byzantine failures are presented throughout the Sections 3.6 - 3.13.

Lets recall the categorization of the nodes as correct and incorrect (faulty):

1. a node p_i is called correct or non faulty if it behaves timely and according to its specification given by A_i^n.

2. a node p_i is called incorrect or faulty if and only if it is not correct.

In the following, the set of faulty nodes is labeled as F, while the set of correct nodes is abreviated as $\Pi \setminus F$. The maximum number of faulty nodes is given by $f = |F|$. Since a particular execution can have fewer faults, the total number of faulty nodes during a single execution is given by $t \leq f$. Therefore, any algorithm $A(P)$ satisfying the consensus problem P has to guarantee the following three properties:

1. *Termination*: In every admissible execution the decision value y_i of all nodes $p_i \in \Pi \setminus F$ is eventually assigned. [1, Att04]

2. *Agreement*: If the decision values y_i and y_j for every pair of correct nodes $p_i, p_j \in \Pi \setminus F$ is assigned then $y_i = y_j$. [1, Att04]

3. *Validity*: For every execution where $\forall p_i \in \Pi : x_i = v$ the decision value of all correct nodes y_i is v. [1, Att04]

The first property, *termination*, states that the protocol has to execute within finite time, otherwise an infinite loop would perfectly solve the consensus problem, though never reaching consens as we intend. The second property, *agreement*, has to be established, to prevent each node deciding on its own initial value x_i. The last property, *validity*, guarantees that trivial solutions like simply assigning $y_i = 0$ on all nodes, regardless of their initial value x_i, is prohibited. [1, Att04]

As described above we compare the protocols satisfying consensus according to their worst case performance characteristics. A lot of research and lower bound proofs have been investigated in the last decades leading to the following results. [12, Fis85] proved that consensus cannot be reached in an asynchronous system, even with a single crash failure. Further, the lower bounds of $n \geq f + 1$ for crash and $n \geq 3f + 1$ for Byzantine fault tolerant protocols have been developed. The lower bound on the number of rounds is given by $r \geq f + 1$, which is valid for crash as well as Byzantine failure tolerant consensus protocols. For systems withstanding Byzantine failures, the message complexity has been proved to be at least polynomial in n. [1, Att04]

3.3 A Reduction of Replica Determinism to Consensus

As already described in Section 2.3, replicated nodes have to guarantee replica determinism, which is quite a challenging task. Schneider has given a quite easy but powerful abstraction of replica determinant nodes, based on a state-machine approach. Therefore, a client server model is applied, where clients request the execution of a task on a server. The idea is that tasks executed on these servers are built as state machines S consisting of states Q and transitions ϕ, where a state represents the history of the execution and a transition represents the atomic execution of a deterministic program extending the history and thus lead to a state transition. [30, Sch90]

Lemma 1. *By the atomicity of the transitions the output of a state machine S is determined only by its initial state $q \in Q$ and the requests issued. [30, Sch90]*

Assuming a point-to-point communication primitive (as given in Figure 3.5), by the atomicity of transitions, it is quite easy to guarantee that requests originating from client c_0 to the state machine sm_0 are executed in the same order they were issued by the client. Thus, in this simple case order can be easily guaranteed.

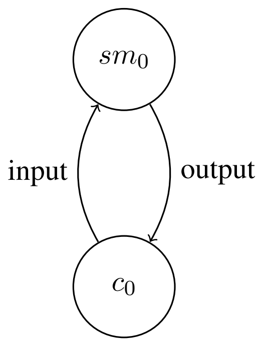


Figure 3.5: Simple example of state machine approach

Unfortunately, in the case of faults, it is not sufficient to rely on the output value of a single instance of the state machine, and thus many instances of the same state machine have to be used. To ensure that each replica executes the requests in exactly the same sequence, the replicas have to be coordinated. [30, Sch90] To guarantee replica determinism on the order of events and input values some kind of agreement protocol has to be applied (see Figure 3.6). In the Byzantine case, [23, Pea80] showed that agreement can only be guaranteed with $n \geq 3f + 1$ nodes.

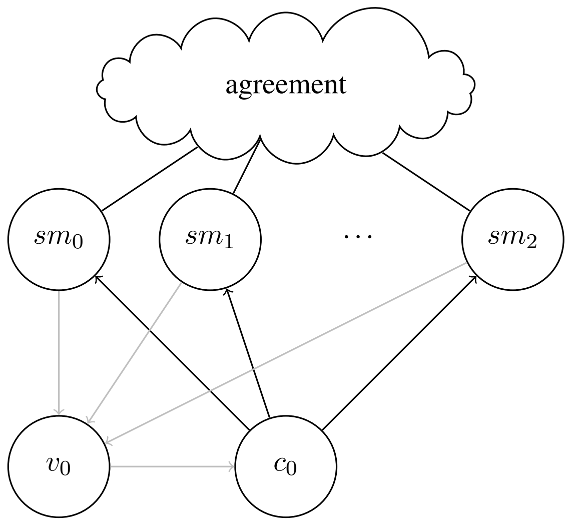


Figure 3.6: State machine with agreement on inputs

Assuming that we can handle replica coordination between the state machines, the problem of choosing a correct output still remains. Since it is not possible to guarantee a f-fault tolerant system by applying a single voter (representing a single point of failure) we have to distinguish between two possible implementations of output usage:

1. Outside the system: If the output is used outside the system and the component can be replicated, we can replicate the voter too, increasing the system resiliency.

2. Within the system: If the output is used as the input value of a client, we can reduce the single point of failure of the voter and the single point of failure of the client to one single point of failure, since we can implement the voter within the client.

As the clients represent a single point of failure, they have to be replicated for the same reason as for the state machines. Leading to the same constraints on their input values (the state machines output values). Therefore, each state machine instance has to agree on the input value chosen among the clients' proposals, and thus agreement among these has to be enforced, as shown in Figure 3.7.

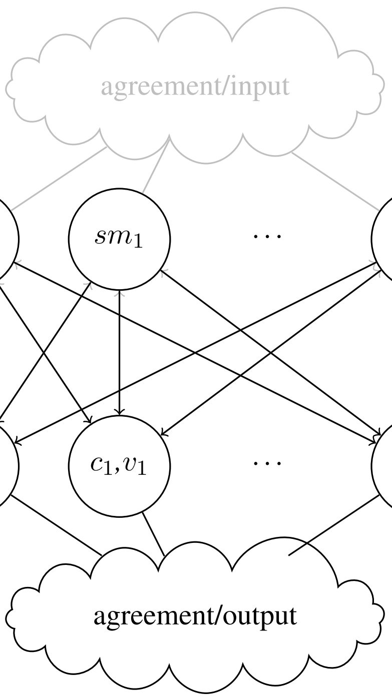


Figure 3.7: State machine with agreement on outputs

Since an instance of a state machine and a client instance as well as two client instances (the one responsible for the input value and the one responsible for the output value) can be combined, we can reduce the problem of ensuring replica determinism with values used inside the sphere of the system to the consensus problem (as stated in Theorem 6).

Theorem 6. *The problem of ensuring replica determinism within the sphere of the system guaranteeing f-resiliency can be reduced to consensus (as well for the crash as for the Byzantine fault model).*

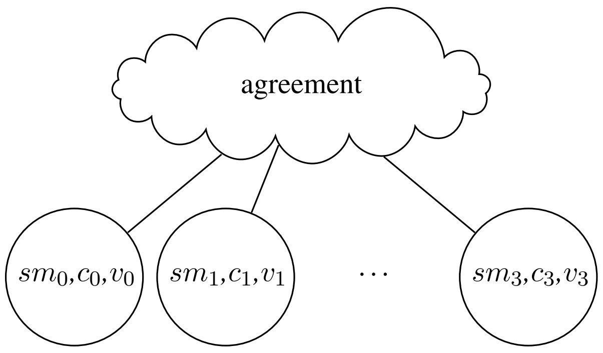


Figure 3.8: Simplification of the state machine approach to consensus

3.4 Relation between Problems

Before we start our discussion about consensus protocols we clarify the relation between certain problems similar to the consensus problem.

Besides the severity of fault classes described in Section 2.2 we can distinguish between deterministic protocols as well as randomized protocols. In the case of deterministic protocols, drawn on the state machine approach (see Section 3.3), the state resulting from a transition is uniquely given by the current state and the event triggering a transition. This is not the case when randomized protocols are used. In the case of randomized protocols the same event can result in different successor states. Given a common current state, it is possible for the protocol to decide between different transitions with a predefined probability. [16, Had93]

For our further analysis we restrict ourselves to deterministic protocols.

Further, we restrict our elaboration of certain problems to those which communicate by broadcasting messages via a fully connected point-to-point message passing subsystem. As the loss of a single message can already cause inconsistencies within the system a distributed system using unreliable broadcasts cannot be assumed to be fault tolerant. *Reliable Broadcast*, *FIFO Broadcast*, *Causal Broadcast*, *Atomic Broadcast*, *FIFO Atomic Broadcast* and *Causal Atomic Broadcast* are common protocols to solve this problem. We describe the kind of problems they are solving and their properties in the remainder.

The properties of reliable broadcast are *validity*, *agreement* and *integrity* ensuring no message m is generated or altered during a broadcast, and ensuring that all correct nodes eventually deliver the message originally sent by the sender. In the case the sending node fails during execution reliable broadcast offers two possible outcomes on the delivery of a message m. Either no correct node delivers m or all correct nodes deliver m.

1. *Validity*: In every admissible execution all correct nodes $p_i \in \Pi \setminus F$ eventually deliver a message m broadcasted by a correct node $p \in \Pi \setminus F$ in advance.

2. *Agreement*: If a correct node $p \in \Pi \setminus F$ delivers message m all correct nodes $p_i \in \Pi \setminus F$ eventually deliver m.

3. *Integrity*: Every correct node $p \in \Pi \setminus F$ delivers message m exactly once if and only if m has been broadcasted by a correct node $p \in \Pi \setminus F$ before.

If our system relies on messages delivered in the same order they have been broadcasted by a node we have to refine our problem description with *first in first out* semantic, leading to *FIFO Broadcast*, ensuring that a message m broadcasted by a correct node p before p broadcasts m' is not delivered after m'.

4. *FIFO Order*: If m has been broadcasted before m' by a correct node $p \in \Pi \setminus F$, then no correct nodes $p_i \in \Pi \setminus F$ delivers m' before m.

As an example, this can be an important property in the case a node broadcasts its current state, or the value of a sensor. Consider a system monitoring an autonomous flight system and two states. The first state reporting a non critical system state and the second state reporting a loss of altitude, probably leading to a collision. If the messages are not delivered in FIFO order this can lead to catastrophic consequences.

In many scenarios, the causality of events (introduced by [20, Lam78] and outlined in Section 2.6) is crucial for their correct interpretation. Protocols able to ensure causal order are called *Causal Broadcast* and are extentions to *Reliable Broadcast* by the *causal order* property. Besides *FIFO Broadcast*, *Causal Broadcast* guarantees causal relationships not only for messages sent by a single sender p, but for messages broadcasted by any correct node.

5. *Causal Order*: If the broadcast of m causally influences m' no correct node $p \in \Pi \setminus F$ delivers m' before m.

As we already depicted in Section 2.6, causality defined by the *happened before relation* accomplishes a partial order of messages. Lets think about two messages: (i) m sent by node p_i (managing the fuel consumption) which says the speed of our airplane has to be decreased by $10m/s$ and (ii) a second message m' sent by node p_j (calculating the vertical acceleration according to m) that says that the speed of the airplane has to increase by 10% to keep the desired altitude.

If the two messages causally relate to each other, the order of the messages is crucial. Assume two nodes p and q controlling the engines of the airplane and p receives m before m', while q receives m' before m. Given, the speed of the airplane was $x\ m/s$, node p outputs an actual speed of $(x - 10) * 1,1$ while q outputs $x * 1,1 + 10$. Therefore, in such a system the causal order has to be supported. To establish *total order* of messages among all nodes, we introduce the following property based on the properties given by *Reliable Broadcast*, namely *Atomic Broadcast*.

6. *Total Order*: If two correct nodes $p, q \in \Pi \setminus F$ deliver the messages m and m', p delivers m before m' if and only if q delivers m before m'.

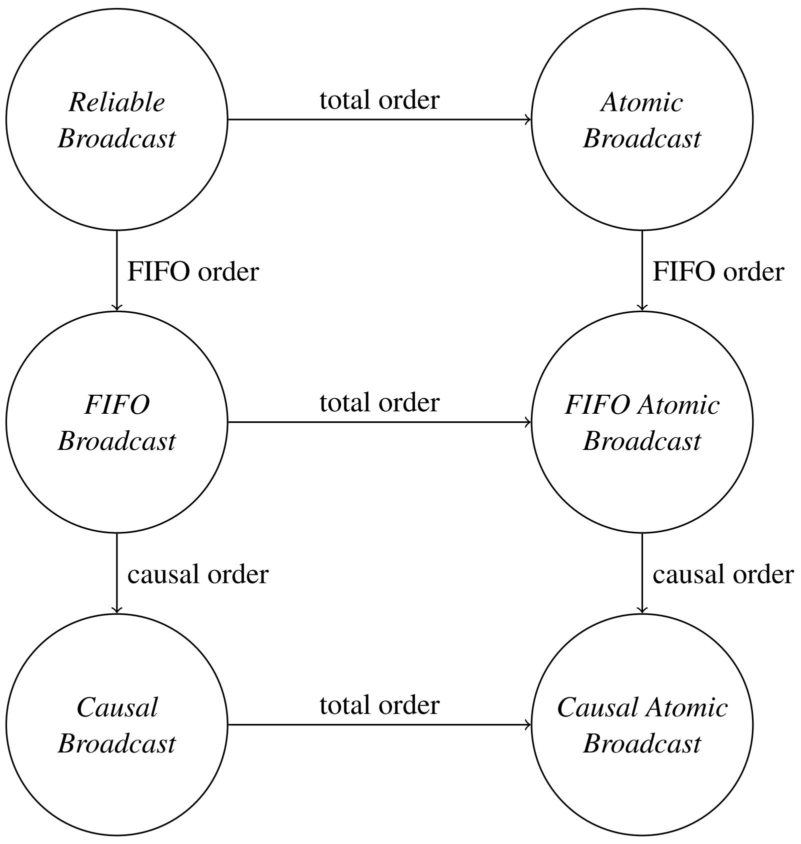


Figure 3.9: Relations among broadcasting problems

FIFO Atomic Broadcast and *Causal Atomic Broadcast* are defined as *FIFO Broadcast* and *Causal Broadcast* but are extended by the *total order* property. An overview of certain variations and relations of the presented broadcasting primitives is given in Figure 3.9.

To analyze the relations between the given problems as well as the consenus problem we apply a well known method from the field of formal methods, namely reduction. Reduction allows us to state that a problem A is not harder than a problem B under certain assumptions about the system model, if we can find a transformation $T_{B \leftarrow A}$ transforming every instance of a problem B to a instance of problem A. We abbreviate this with $A \preceq B$. If the reduction holds in both directions, thus $A \preceq B$ as well as $B \preceq A$ we say the problems A and B are equivalent. [16, Had93]

In the following, we will sketch the idea of a reduction between *Atomic Broadcast* and *Consensus*, leading us to the fact that Consensus is not a harder problem than Atomic Broadcast.

1. Termination (of Consensus): Since every correct node must know in which round it has to decide after the protocol starts, termination of consensus is given.

2. Validity of *Reliable Broadcast* states that a correct processor p_i only delivers a message m previously broadcasted by a correct processor p_j. Constructing an algorithm executing *Atomic Broadcast* for all nodes $p \in \Pi$. We know that $n \geq 3f + 1$ holds in the Byzantine fault model (or $n \geq f + 1$ holds in the crash fault model), for a total number of nodes n and a maximum number of faulty nodes f. Thus, at least $n - f$ nodes deliver a message initially broadcasted by one of the $n - f$ correct nodes. By *Total Order* it is supported that even if multiple executions of the protocol are executed on the nodes back to back, all messages delivered and used for the consensus protocol are in the same order they were previously broadcasted. This has the same effect like running multiple consensus instances one after another. Therefore, validity (of Consensus) follows immediately by validity of *Atomic Broadcast*.

3. Agreement of *Reliable Broadcast* states that if a correct processor $p_i \in \Pi \setminus F$ delivers message m all correct processors $p_j \in \Pi \setminus F$ deliver m. Again, by constructing an algorithm executing *Atomic Broadcast* for all nodes $p \in \Pi$ we know that at least $n - f$ nodes deliver the same message m for each execution of *Reliable Broadcast*. Applying a majority voter $f(m_0, \ldots, m_{n-1})$ every node $p_i \in \Pi \setminus F$ will decide on the same value $y_i = f(m_0, \ldots, m_{n-1})$. Assume a node p_i decides another majority value than node p_j. Since all correct nodes deliver exactly the same values this means either p_i or p_j are faulty. Therefore, all correct processors decide on the same value y. Proving that agreement of Consensus follows directly from agreement of *Atomic Broadcast*.

The other direction can be argumented similarly and thus we can state that the two problems are equivalent. By this we can adopt all results from one of the problems to the other and vice versa. [16, Had93]

3.5 Byzantine Agreement

In literature *Atomic Broadcast* is often referred to as *Byzantine Agreement*. Thus, we will outline some common representatives of these algorithms. *Byzantine Agreement* is the problem of reaching agreement on a single value proposed by a dedicated node p_i in a Byzantine faulty environment. The aim here is to propagate the message m of the sender p_i to all other nodes in the system in such a way that all correct nodes (i) either deliver the message m if the sender p_i is correct (ii) or deliver no message at all. A draft for a reduction of consensus to Atomic Broadcast has been given in Section 3.4. Therefore, the agreement on a single value x_i propagated by a dedicated sender p_i is the major difference to the Consensus problem, where agreement on a set of values $(x_0, \ldots, x_{n-1})$, one value x_i per node is required. The *Byzantine Agreement Algorithms* sketched are given in Table 3.1.

The algorithm shown in [23, Pea80] was the first protocol for *The Byzantine Generals Problem* proposed in [21, Lam82], where the handling of malfunction and inconsistent communication of system components was described. The Byzantine Generals Problem is about a group of generals camping around their enemy city. The generals of the Byzantine army can only communicate via messengers to agree on a common battle plan. The problem given is how to ensure

Protocol	n	rounds	communication	computation
[23, Pea80]	$n \geq 3f+1$	$f+1$	$exp(n)$	$exp(n)$
[10, Dol82]	$n = 3f+1$	$2f+3$	$f^3 log(f)$	1

Table 3.1: Representatives of Byzantine agreement algorithms

all loyal (non faulty) generals decide for the same plan. The algorithm communicates in each round which informaton all the nodes heard so far. This information is stored in a tree on each node (the so called information tree). Given that the number of nodes exceeds $n \geq 3f+1$, a correct solution can be found by majority voting based on the information tree. In contrast to other implementations, the algorithm is quite simple, but requires a huge communication and computation effort. Since, this kind of agreement is similar to the Exponential Information Gathering protocol presented in the next section, we won't go into more detail now.

Another algorithm has been developed by [10, Dol82]. It uses an asymmetric approach where the identifiers of a node are sent to state their witness to 1 while the trust in 0 is given by omitting to send a message. In the first round the dedicated transmitter broadcasts an initiation message with its value to all nodes (including itself). On receipt of an initiation message containing 1 each correct node broadcasts its node id. Dependent on a round dependent threshold the nodes being witnesses to the propagated value can support the message received during initiation. The propagation of spurious messages is prohibited by restricting the correct nodes to only broadcast messages if enough supporters for the message are known. If the number of supporters exceeds $2f+1$ the nodes agree on 1. Otherwise they agree on 0. (A detailed description of the algorithm can be found in [10, Dol82].)

Revisit that in the Byzantine Agreement Problem all correct nodes agree on a value or at least decide that the originator of the value is faulty. [10, Dol82] In the Consensus Problem all correct nodes have to decide for a single value out of a set of values distributed throughout the system. In the following Sections 3.6-3.13 we will discuss the most famous algorithms solving the consensus problem assuming the Byzantine fault case.

3.6 An Exponential Algorithm

Exponential Information Gathering (EIG)

One of the most famous Byzantine consensus algorithms is the EIG algorithm, given by an extention to the Byzantine agreement protocol presented by [23, Pea80]. The algorithm requires $n \geq 3f+1$ nodes and runs for $f+1$ rounds, which are optimal results for Byzantine consensus, but forces an exponential communication effort with respect to the number of nodes n.

The algorithm is split into two parts. The first part is used to gather and distribute information among the nodes. The second part is used to calculate a decision based on the results of the first part.

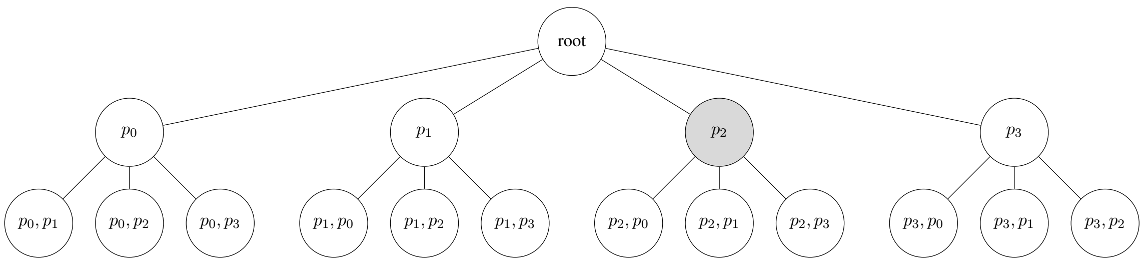
1. Gathering Information: The information gathered is stored in a tree locally on each processor p_i ($tree_i$). The root node of the $tree_i$ holds the initial value of p_i. It has $n - 1$ children one for each node p_j, where $p_j \in P \setminus p_i$. And each of them again has successors for each node p_q, where $p_q \in P \setminus \{p_i, p_j\}$ and so forth.

   All edges in the tree are labeled with the name of the succeeding node, while each node is labeled by the sequence of edge labels starting from the root ("p_0,p_1,p_2" e.g.). The node values, namely $tree_i(p_0, p_1, p_2)$, are filled during the information gathering phase with the values received from the other nodes. Initially they hold the algorithms default value. The tree is constructed such that the meaning of each entry is the perception of node p_i of the value transmitted from p_j. Therefore, e.g., the value $tree_i(p_0, p_1, p_2)$ is the perception of p_i of p_2's view of p_1's view of p_0's value.

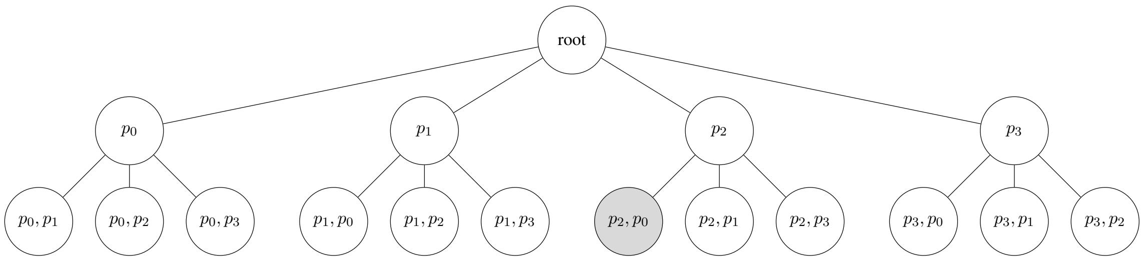
   In each round r of the algorithm the level $l = r + 1$ constructed in the last round of the tree is broadcasted to all other nodes and filled into their tree. This procedure is continued for f+1 rounds after which the tree is complete and the second part of the protocol can calculate the decision value of node p_i based on $tree_i$.[23, Pea80] [1, Att04]

2. Calculation of the Decision Value: To calculate the decision value for node p_i the resolve function is applied recursively to $tree_i$, starting with its root node.

   Therefore, $resolve(\pi)$ returns the value of $tree(\pi)$ if the node labeled with π is a leaf node. Otherwise it returns the majority of all values computed by the resolve functions of the direct children of the node labeled π, or the default decision value $\perp$ of the protocol, if no majority exists. [1, Att04] [15, Gar98] [5, Ber89a]



(a) $tree_0$



(b) $tree_1$

Figure 3.10: Example trees for nodes p_0 and p_1, $p_0, p_1 \in P \setminus F$

The algorithm for node p_i is outlined in Algorithm 1. The function $labels(d)$ returns all the node labels at distance d from the root in the tree.

Algorithm 1 *Exponential Information Gathering algorithm, code for node p_i,* $0 \leq i \leq n$

```
   Initialization:
 1: $x := \{1|0\}$ {initial value for $p_i$}
 2: $y$ {decision value}
 3: $\perp := 0$ {default value}

 4: $f$ {upper bound on nr of faulty nodes}
 5: $n$ {nr of nodes}

 6: $tree$ {the decision tree}

   Information Gathering:
 7: for round $k$, $1 \leq k \leq f + 1$: do

 8:     for each $\pi$ in labels($k - 1$): do
 9:         send $tree(\pi)$ + $\pi$ to all nodes, where $i \notin \pi$

10:     on receipt of $tree_j(\pi')$ + $\pi'$ from node $p_j$ append $tree_j(\pi')$ to $tree(\pi', j)$
11:     if no message was received for a leaf add $\perp$ to that leaf

   Calculation of Decision
12: $y :=$ resolve($root$)
```

In Figure 3.10 the trees for two correct nodes p_0 and p_1 are outlined. The two grey shaded nodes are an example of the information exchange between the nodes. The node value p_2 of $tree_0$ is copied to the node p_2,p_0 in $tree_1$. During round 2 p_0 tells p_1 about the values it received in the previous round, the value of p_2 in our example. Based on this mechanism, it can be shown that, for all non-faulty processors, the resolve function $resolve_i(\pi)$ equals the node value $tree_j(\pi')$ and that at least one common node on each path from the root to the leaves must exist. This is the basis for the algorithms correctness proof (which is omitted here).

3.7 A Polynomial Algorithm with Constant Message Size

As we have seen in the previous section, the Exponential Information Gathering protocol provides optimal resiliency and is also optimal in the number of rounds required, but has an exponential increase in message complexity with respect to the number of nodes. In this section we present algorithms with constant message complexity. We will see the tradeoff between the complexity criteria (resiliency, round complexity and message complexity). Unfortunately, by now no algorithm is known to be optimal in all three criteria. [1, Att04]

Phase Queen

The *Phase Queen* algorithm is the first algorithm with polynomial communication effort that will be discussed in more detail. It executes in $f + 1$ phases, where each phase consists of two rounds of communication. The message size required by the *Phase Queen* algorithm is 1 bit. Compared to the *Exponential Information Gathering* protocol, beside the slightly increased number of rounds it additionally requires $n \geq 4f + 1$ nodes to tolerate f faulty ones.

The *Phase Queen* algorithm is given in Algorithm 2. Each node has a private preference value V which is initially set to X (the input value of the node). As outlined above, the algorithm executes in $f + 1$ phases. Each phase consists of two rounds of communication, namely the universal exchange and the queens broadcast, as shown in Figure 3.11. During the universal exchange every node broadcasts its private preference V to all other nodes in the system. On receipt of the messages, the number of received 1s and 0s are counted in $C[0]$ and $C[1]$, respectively. Missing messages are counted as if they would have the default value $v_\perp = 0$. If the count of 1s exceeds the threshold of $\frac{n}{2}$, the node sets its preference to 1 otherwise to 0. In the second round of the phase, namely the queens broadcast, a dedicated node is chosen to broadcast its private preference to all other nodes. On receipt of the queens broadcast, the receiving nodes set their preference value V to the received message, if the belief in V was too low, namely, if the count of received 1s respectively 0s is below $\frac{n}{2} + f$, V is overwritten by the queens broadcast. The queen of each phase is chosen at design time. It has to be a uniquely chosen node for each phase. Thus, the same node can only be queen in at most one phase of the protocol.

By the thresholds of both parts of a phase, it is ensured that all correct nodes will prefer a certain value V at the end of a phase if they already preferred it at the start of the round. When

Algorithm 2 *Phase Queen algorithm, code for node p_i, $0 \leq i \leq n$*

```
    Initialization:
 1: $x := \{1|0\}$ {initial value for $p_i$}
 2: $y$ {decision value}

 3: $f$ {upper bound on nr of faulty nodes}
 4: $n$ {nr of nodes}

 5: $V := \{x\}$ {preference value}
 6: $C[0] := 0$ {counter for received 0s}
 7: $C[1] := 0$ {counter for received 1s}

    Computation:
 8: for round $k$, $1 \leq k \leq f + 1$: do

       universal exchange
 9:    send $V$
10:    $C[0] :=$ the number of received 0s
11:    $C[1] :=$ the number of received 1s
12:    $V := C[1] > \frac{n}{2}$

       queens broadcast
13:    if $k = i$ then
14:       send $V$
15:    if $C[V] < \frac{n}{2} + f$ then
16:       $V :=$ the message received

    decision
17: $y := V$
```

the first non-faulty queen broadcasts its value, the believe of all non-faulty nodes will be fixed, if it was not already.

The belief, however, can not be different for different, non-faulty nodes, as it requires a minimum of $\frac{n}{2} + f$ equal values and therefore there can be no majority for a different value on any other node. Therefore, if one non-faulty node has a majority for a value, all other non-faulty nodes must also prefer this value or do not prefer a value at all (and will use the queen's suggestion).

Phase King

The second polynomial algorithm we discuss here is called Phase King and follows a similar philosophy as the Phase Queen algorithm discussed previously. Like Phase Queen, it breaks ties by allowing a dedicated node per phase to propagate its current preference. The Phase King algorithm uses an extra round of communication where the nodes can tell others whether they have already a strong preference or not. This additional information reduces the number of required nodes to $n \geq 3f + 1$. The algorithm requires $3(f + 1)$ rounds when using 2 bit messages or $4(f + 1)$ rounds when using 1 bit messages. [8, Ber92a] [5, Ber89a]

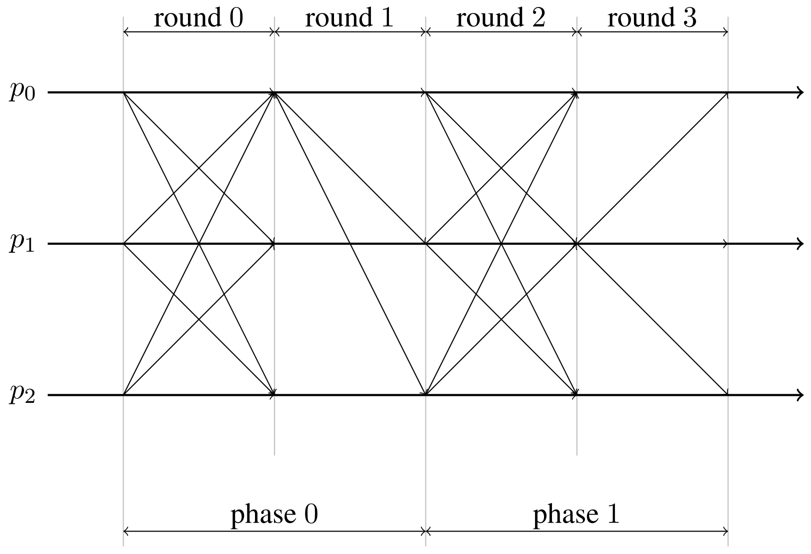


Figure 3.11: Execution of Phase Queen algorithm

A pseudocode implementation of the *Phase King* algorithm is given in Algorithm 3. To establish single bit messages, the second universal exchange has to be split in two separate communication rounds. In the first universal exchange (first round of phase) the protocol broadcasts its preference value to all other nodes.

As in the *Phase Queen* algorithm, in the second round of the phase the number of received 0s and 1s are counted in $C[0]$ and $C[1]$, respectively. Therefore, $C[0]$ and $C[1]$ give an indicator which value $v \in \{0, 1\}$ is strongly preferred. This preference is exchanged during the second universal exchange. If $C[v]$ exceeds the threshold $n - f$, we can safely assume that at least more than half of all correct nodes would decide $Y = v$. Further, we can state that no two correct nodes p_i and p_j $i \neq j$ will ever evaluate $C_i[0] \geq n - f$ while $C_j[1] \geq n - f$. Assume this could happen. By the thresholds of $\geq n - f$ both nodes would have received at least $n - f$ messages stating 0 or 1, respectively. Summing up at least a total of $(n - f) + (n - f) = 2n - 2f$ messages (counting the spurious messages twice) must have been sent. By $n \geq 3f + 1$ this gives a total of $4f + 2$ messages. By the pigeonhole principle, given a maximum of 3f+1 messages per round $f + 1$ messages must be spurious, and therefore $f + 1$ nodes must be faulty. Leading to a contradiction. Therefore, we can conclude that as soon as one correct node signalises a strong preference for v, no other correct node p_j will have a strong preference for $\bar{v} \neq v$. After the second exchange $D[v]$ holds the number of nodes that detected a strong preference to decide to v during the first universal exchange. A node p_i can set its local preference value V_i to v, if $D[v]$ exceeds f, namely, if at least one correct node detected a strong preference for v.

During the third round of phase k, namely the kings phase, a dedicated node p_k broadcasts its preference value V_k. Every node p_i with $D_i[V_i] < n - f$ sets its preference to the value propagated by the king. Since $C[v] \geq n - f$ for at least $n - f$ nodes, and therefore $D[v] \geq n - f$ we can conclude that, if all correct nodes prefer a value v at the beginning of a phase they will prefer the same value at the end of the phase. A similar argument as for Phase Queen can be given to state that if one correct node evaluates $D[v] \geq n - f$ to true no other correct node can evaluate $D[\bar{v}] \geq n - f$. Assuming a correct king, either a correct node already holds the value v during the kings broadcast with $D[v] \geq n - f$ or sets $V = v$ during the kings broadcast. Therefore, by applying $f + 1$ phases we can guarantee correctness of the protocol. [7, Ber89b] [8, Ber92a]

Algorithm 3 *Phase King algorithm, code for node p_i,* $0 \leq i \leq n$

```
    **I**nitialization:
 1: $x := \{1|0\}$ {initial value for $p_i$}
 2: $y$ {decision value}

 3: $f$ {upper bound on nr of faulty nodes}
 4: $n$ {nr of nodes}

 5: $V := \{x\}$ {preference value}
 6: $C[0] := 0$ {counter for received 0s}
 7: $C[1] := 0$ {counter for received 1s}
 8: $D[0] := 0$ {counter for strong preference of 0}
 9: $D[1] := 0$ {counter for strong preference of 1}

    **C**omputation:

10: **for p**hase $k$, $1 \leq k \leq f + 1$: **do**

      **u**niversal exchange 1
11:   **if** $D[V] < n - f$ AND $k > 1$ **then**
12:     $V :=$ the message received
13:   send $V$

      **u**niversal exchange 2 part 1
14:   $C[0] :=$ the number of received 0s
15:   $C[1] :=$ the number of received 1s
16:   send $C[0] \geq n - f$

      **u**niversal exchange 2 part 2
17:   $D[0] :=$ the number of received 1s
18:   send $C[1] \geq n - f$

      **k**ings broadcast
19:   $D[1] :=$ the number of received 1s
20:   $V := D[1] > f$

21:   **if** $k = i$ **then**
22:     send $V$

23: **if** $D[V] < n - f$ AND $k > 1$ **then**
24:   $V :=$ the message received

    **d**ecision
25: $y := V$
```

3.8 Early Stopping

In this section we present a technique called *Early Stopping*. *Early Stopping* enables the algorithm to detect when a preference for a decision value can safely be finalized. This is the case, when the algorithm detects that all correct nodes would reach agreement, and thus termination at an earlier phase is possible. [2, Bar95]

Early Stopping Phase King

No matter if faults have occured or not, the number of rounds required by the *Phase King* algorithm is constant. The *Early Stopping Phase King* variant, however, yields $r = \min(3(f+1), 3(t+2))$ rounds of communication for $n \geq 3f+1$ nodes and a total bit complexity of $b = O(nt \min(f+2, t+1))$ using two bit messages and $r = \min(5(f+1), 5(t+2))$ for single bit messages, if f is the maximum number of faults the algorithm can handle and t is the actual number of faults occurring during a single execution. [2, Bar95]

As we can see in Algorithm 4 the universal exchanges are similar to the exchanges used in Algorithm 3. The main difference to the previously discussed *Phase King* algorithm can be found in the king's broadcast. In contrast to *Phase King*, the king's broadcast in this variant is performed by every node. During this stage every node sends a tuple consisting of a value preferred by itself and an indicator determining the believe in the strong preference for this value. By line 19, if less than $f+1$ nodes signal that they strongly believe in its preference value, the preference value of the phase's king is set as preference value. This execution is similar to the unmodified algorithm.

To optimize the algoritm we have to detect when it is save to stop. In the unmodified *Phase King*, a node evaluating $D[v] \geq n - f$ would not change the value $v \in \{0, 1\}$ during this phase. This is signalised by sending $\Pi = 1$ during the kings phase. The number of nodes which signalised $\Pi = 1$ and therefore won't change their decision values in the current execution of the protocol is counted by S. If more than $2f$ nodes signalise this ($S > 2f$), at least $f+1$ correct nodes have a strong preference $D[v] \geq n - f$ for a value v. Therefore, at least $f+1$ correct nodes have notified that at least $n - f \geq 2f + 1$ correct nodes already prefered $V = v$ at the beginning of phase k. Lets call this set P' in the remainder. In the unmodified version the nodes $p_i \in P'$ would again send their values $V = v$ to each other during the next phase. Every correct node would evaluate $C[v] \geq n - f$ during the next phase and therefore all correct nodes would decide to v. By this, we can terminate the execution of the protocol for a node safely, if it encounters $S > 2f$. Therefore, the other nodes that have not terminated yet must compensate the missing messages from the nodes already terminated. If no message from p_i is received during a phase by a node p_j, p_i assumes to receive its own preference value V_i, except for Π_j it assumes 1. By line 30 of Algorithm 4 all correct nodes (including the terminated ones) have set $V = (D[1] > f) = v$ in the kings phase (and in any later one) when the first processor terminates. Thus, all correct nodes will decide $Y = v$ too. [6, Ber92b]

Algorithm 4 *Early Stopping Phase King algorithm, code for process p_i, $0 \le i \le n$*

```
Initialization:
1: $x := \{1|0\}$ {initial value for $p_i$}
2: $y$ {decision value}

3: $f$ {upper bound on nr of faulty nodes}
4: $n$ {nr of nodes}

5: $V := \{x\}$ {preference value}
6: $C[0] := 0$ {counter for received 0s}
7: $C[1] := 0$ {counter for received 1s}
8: $D[0] := 0$ {counter for strong preference of 0}
9: $D[1] := 0$ {counter for strong preference of 1}
10: $\Pi := 0$ {strength indicator for preference value}
11: $A[0..n-1] := \{0\}^n$ {input memory for king's phase}
12: $B[0..n-1] := \{0\}^n$ {input memory for king's phase}
13: $S := 0$ {number of nodes known with a permanent decision}

Computation:

14: for phase $k$, $1 \le k \le f+1$: do

      universal exchange 1
15:   if $k > 1$ then
16:     $B[i] :=$ the value received from $p_i$
17:     $S := \sum_{i=0}^{i<n} B[i]$
18:     if $S \le f$ then
19:       $V := A[k-1]$
20:     else if $S > 2f$ then
21:       $y := V$
22:       TERMINATE
23:   send $V$

      universal exchange 2 part 1
24:   $C[0] :=$ the number of received 0s
25:   $C[1] :=$ the number of received 1s
26:   send $C[0] \ge n - f$

      universal exchange 2 part 2
27:   $D[0] :=$ the number of received 1s
28:   send $C[1] \ge n - f$

      kings broadcast part 1
29:   $D[1] :=$ the number of received 1s
30:   $V := D[1] > f$
31:   $\Pi := D[V] \ge n - f$
32:   send $V$

      kings broadcast part 2
33:   $A[i] :=$ the value received from $p_i$
34:   send $\Pi$

35: $B[i] :=$ the value received from $p_i$
36: $S := \sum_{i=0}^{i<n} B[i]$
37: if $S \le f$ then
38:   $V := A[k-1]$
39: else if $S > 2f$ then
40:   $y := V$
41:   TERMINATE

decision
42: $y := V$
```

Protocol	resiliency	message size	time complexity
EIG	$n \geq 3f+1$	exp	$f+1$
EIG	$n \geq 3f+1$	1	msgsize$(n, f+1)$
Phase Queen	$n \geq 4f+1$	1	$2(f+1)$
Phase King	$n \geq 3f+1$	1	$4(f+1)$
Early Stopping EIG	$n \geq 3f+1$	exp	min((f+1), (t+2))
Early Stopping EIG	$n \geq 3f+1$	1	msgsize$(n,$ min((f+1), (t+2)))
Early Stopping Phase Queen	$n \geq 4f+1$	1	min(3(f+1), 3(t+2))
Early Stopping Phase King	$n \geq 3f+1$	1	min(5(f+1), 5(t+2))

Table 3.2: Properties of algorithms and their Early Stopping variants

Other Early Stopping Results

An early stopping variant of the *Phase Queen* algortihm was presented in [6, Ber92b]. It uses a similar approach as the already presented one for the *Phase King* algorithm.

The *Early Stopping Phase Queen* variant achieves a round complexity of $r =$ min(2(f+1), 2(t+2)) for two bit messages and $r =$ min(3(f+1), 3(t+2)) for single bit messages, given that $n \geq 4f+1$.

An implementation for the early stopping EIG algorithm (presented in [6, Ber92b] achieves a round complexity of $r =$ min((f+1), (t+2)) using $n \geq 3f+1$ nodes. While these measures are optimal, the exponential increase in message size, however, still exists.

Table 3.2 contains a comparison of all discussed algorithms (including their early stopping variants). Since the further analysis and the implementations are based on algorithms using 1 bit messages, the function msgsize(n, R) is introduced here, to make the theoretical results more comparable. It evaluates the number of messages to be sent per round, if the message size does not exceed 1. In Table 3.2 it is used to transform the exponential message size used by the Exponential Information Gathering algorithm to its time complexity if single bit messages are used. Therefore, the result of msgsize(n, R) is exponential in n and R, where R gives the number of rounds the protocol executes.

As shown in Tables 3.3 - 3.5, applying an early stopping variant of an algorithm is not always beneficial. Since all of the early stopping variants do increase the round complexity of the algorithm, their use has to be carefully evaluated first.

First, we evaluate the benefits of the *Early Stopping Exponential Information Gathering* protocol. We therefore analyze the rounds needed by the normal protocol and compare it to the *Early Stopping* variant. We thereby restrict the protocols to single bit messages. The *EIG* protocol needs $r =$ msgsize$(n, f+1)$ rounds in this case, where msgsize(n, R) is given by Equation 3.1. It can be seen that in case of single bit messages the EIG protocol has an exponentially increasing round complexity.

n	f	t	rounds needed for ESEIG	rounds needed for EIG
4	1	0	4	4
4	1	1	4	4
7	2	0	7	37
7	2	1	37	37
7	2	2	37	37
16	5	0	16	396076
16	5	1	226	396076
16	5	2	2956	396076
16	5	3	35716	396076
16	5	4	396076	396076
16	5	5	396076	396076

Table 3.3: Evaluation of benefit using *Early Stopping EIG*

$$\text{msgsize}(n, R) = 1 + \sum_{d=1}^{d<R} \prod_{i=1}^{i\leq d} (n - i) \tag{3.1}$$

Table 3.3 also shows the rounds required by the early stopping variant of *EIG* for $f = 1$, $f = 2$ and $f = 5$ for the minimum number of required nodes ($n = 3f + 1$). For $f = 1$ we have to use a system consisting of $n = 4$ nodes and the unmodified *EIG* as well as its early stopping variant require 4 rounds, even if no fault occurs. In case of $f = 2$, the unmodified algorithm needs exactly 37 rounds, when using the single bit variant independent of the number of actually occurring faults t. The early stopping version, however, only requires seven rounds, if there is no fault actually occurring ($t = 0$). If at least one fault occurs ($t > 0$), however, the early stopping version also requires 27 rounds. For $f = 5$ the unmodified algorithm even requires 396.076 rounds of communication. Therefore, the benefit of the early stopping variant for $t \in \{0, 1, 2, 3\}$ is enormous.

Table 3.4 summarizes the round complexity of the Phase Queen algorithm (using $f = 1$ $f = 2$ and $f = 5$). Again the benefit of early stopping gets relevant for higher values of f. We can see that for $f = 1$, $r = 6$ rounds of communication are required in the early stopping case. Therefore, the standard protocol with $r = 2(f + 1) = 4$ is faster in this case and therefore using early stopping would be disadvantageous under this failure hypothesis.

If we have to assume a very high number of faults ($f = 20$, e.g.), the execution time in the fault free case will be much better in the early stopping variant (6 rounds vs. 42 rounds). As the table shows, even for lower numbers of tolerable faults ($f = 5$, e.g.), the early stopping variant can lead to medium reduction in execution time, if the overall fault probability is low. Nevertheless, the Phase Queen algorithm is a good example on how the early stopping variant

f	t	rounds needed for ESPQ	rounds needed for PQ
1	0	6	4
1	1	6	4
2	0	6	6
2	1	9	6
2	2	9	6
5	0	6	12
5	1	9	12
5	2	12	12
5	3	15	12
5	4	18	12
5	5	18	12
20	0	6	42

Table 3.4: Evaluation of benefit using *Early Stopping Phase Queen*

f	t	rounds needed for ESPK	rounds needed for PK
1	0	10	8
1	1	10	8
2	0	10	12
2	1	15	12
2	2	15	12
5	0	10	24
5	1	15	24
5	2	20	24
5	3	25	24
5	4	30	24
5	5	30	24

Table 3.5: Evaluation of benefit using *Early Stopping Phase King*

may worsen the time complexity of the algorithm instead of improving it.

In Table 3.5 we have summarized the single bit variant of the *Phase King* algorithm for $f = 1$, $f = 2$ and $f = 5$. According to Table 3.2 it requires $r = \min(5(f+1), 5(t+2))$ rounds. Since, if $t \leq f = 1$ the algorithm takes 10 rounds no matter if a fault occurs or not, the normal variant of the *Phase King* algorithm with $r = 4(f+1) = 8$ rounds is the better choice in this case. Nevertheless, the benefit of early stopping increases for bigger f as we can see for $f = 5$ as long as the number of actually occurring faults is low.

To summarize the results of this section, the use of *Early Stopping* has to be evaluated very carefully with regard to the number of tolerable and actually occurring faults. In some cases, the application of early stopping may even worsen the time complexity of an algorithm. For systems with a large number of nodes and a high value of f, early stopping shows the biggest benefits if the probability of actual occurring faults is low. Further, the early stopping variants will only improve the average case and not the worst case runtime. If the execution time of the algorithm has to be bounded, the application of early stopping will be futile, even if it improves the average performance. Therefore, in many cases the usage of the normal, non early stopping variants may be favorable.

3.9 Applying Committees to Phase King

As presented in Section 3.7, the *Phase King* algorithm is optimal in the required number of nodes n and in its bit complexity. But its round complexity is at least 4 times the optimum one.

To reduce the round complexity [8, Ber92a] adapted the idea of a divide and conquer strategie (as proposed in [7, Ber89b]) and applied the committee technique to the *Phase King* algorithm to reduce the round complexity.

As we have already presented, the correctness of the Phase King algorithm is based on the existence of a single, fault free King's broadcast. Running the algorithm for $f + 1$ phases guarantees the existence of such a broadcast. The idea applied by [8, Ber92a] is to partition the set of nodes in R disjoint sets Q_k. Each set Q_k represents the preference of the King in phase k, $0 \leq k \leq R$.

Algorithm 5 *Excerpt of Phase King algorithm with committees, code for node* p_i, $0 \leq i \leq n$

```
1: ...

   kings broadcast
2: if $p_i \in Q'_k$ then
3:     run $ICP$ with $V$ and broadcast all messges sent by $ICP$ to all nodes
4: if $C[V] < \frac{n}{2} + f$ then
5:     $V :=$ the consensus value of $Q_k$

6: ...
```

As shown in Algorithm 5, we apply a so called intra-committee protocol, abbreviated with ICP, to assure consistency among the nodes of a set Q_k. ICP can be any algorithm, deterministically solving the consensus problem [8, Ber92a]. Since we try to reduce the round complexity of the algorithm it is advantageous if ICP is round optimal. The tradeoff here is clearly the need for ICP which requires multiple rounds of communication per phase. By carefully choosing R, ICP can be chosen to be round optimal while the other complexity measures are marginal, since the committees are quite small.

During the execution of ICP all messages communicated inside Q_k are broadcasted to all $p \in P$. By this, every $p \in P \setminus Q_k$ can compute the preference value of the k^{th} committee and thus the kings broadcast. To guarantee consistency for the King's broadcast, respectively the committees' broadcast, at least more than a third of the committees nodes have to be non-faulty. By this, correctness of this protocol follows the correctness of Phase King, as long as a committee can be found where non-faulty nodes dominate. [8, Ber92a]

A similar technique was used in [7, Ber89b] to reduce the round complexity of *Phase Queen* as we will discuss in Section 3.11. In the following Section 3.10 we present an example for the use of committees with the *Phase King* protocol.

3.10 Sample of Phase King with Committees using EIG as ICP

In this section we will provide an example of the *Phase King Protocol with Committees*. As ICP we use the *Exponential Information Gathering* protocol as shown in Section 3.6. Further, we do not restrict the message size used by the ICP. Nevertheless, the extended *Phase King* algorithm is used in the single bit version as presented in Section 3.7. Recalling the previously discussed properties, the ICP needs $n \geq 3f + 1$ nodes and $f + 1$ rounds with an exponential increase in message size with respect to n. The *Phase King* algorithm on the other hand also uses $n \geq 3f + 1$ nodes but an increased number of $4(f + 1)$ rounds for single bit messages.

Assuming $f = 4$ for this example, the unmodified *Phase King* algorithm needs at least $n_{PK} \geq 13$ nodes and $r_{PK} = 20$ rounds to execute. As we have already discussed, the correctness of *Phase King* algorithm can be achieved, if at least one kings broadcast is fault free. Since all messages of ICP are received by every node, and the committees are known in advance, each node can evaluate the kings broadcast of each round on its own. Since the universal exchanges during the unmodified Phase King execution does not change when we use committees, the criteria of a fault free kings broadcast to guarantee correctness can safely be assumed. To establish this we execute the algorithm in 2 phases using an EIG instance able to withstand $f_{EIG_2} = 2$ faults each. We abbreviate this by $EIG_2(single)$ and $EIG_2(multi)$ for single bit messages and unrestricted messages, respectively, in the remainder.

Since the number of nodes needed for $n_{EIG_2} \geq 7$ we extend the sample model using Phase King with committees to $n_{PK_c} \geq 14$. To show that at least one kings broadcast is executed fault free under the assumption $f = f_{PK} = f_{PK_c} = 4$ we distinguish three cases: (i) Either 2 faults occur during each phase which can be handled by EIG_2 or (ii) 1 fault occurs during the first phase and 3 faults occur during the second phase which can be handled by EIG_2 in the first phase and thus the first phase has a correct kings broadcast or (iii) 1 fault occurs during the second phase and 3 faults occur during the first phase which can be handled by EIG_2 in the second phase and thus the second phase has a correct kings broadcast. Therefore, correctness of the algorithm using the constructed committees can be shown. [8, Ber92a]

$$r_{PK_c(*)} = 2(3 + r_{EIG_2(*)}) \tag{3.2}$$

As given in Equation 3.2, each phase consists of 3 rounds for the universal exchanges and $r_{EIG_2(*)}$ rounds for the ICP protocol. Since $r_{EIG_2(multi)} = 3$ the algorithm using two committees needs $r_{PK_c(multi)} = 12$ rounds of execution in contrast to $r_{PK} = 20$ for the unmodified version.

For our further evaluations we have restricted the communication model to support single bit messages only. Therefore, a mapping of the message size to a sufficient number of additional rounds, as presented in Section 3.8, has to be established. Therefore, $r_{EIG_2(single)} = 37$ and further $r_{PK_c(single)} = 80$. (A detailed conversion is given in Equation 3.1). Similar results can be derived for other scenarios, using restricted message sizes and EIG as ICP.

In summary, the committee technique can be used quite well to reduce the number of rounds in a system where message size is not restricted. In restricted systems the unmodified algorithms perform better.

3.11 Applying Committees to Phase Queen

In [7, Ber89b] a similar technique as described in Section 3.9 is applied to the *Phase Queen* algorithm. Here the Queens broadcast is substituted by an inter committee protocol ICP, fullfilling the same properties as the inter committee protocol used in Section 3.9. The major difference in the committees Q'_k used to reduce the round complexity in Phase Queen case is the way they have to be constructed.

Since we have to guarantee at least one broadcast of a queen (in our case a committee), we introduce f'_k representing the number of faulty nodes assumed per committee. [7, Ber89b] proposes to choose R' and the size of the committees $|Q'_k|$ as follows:

$$\sum_{k=1}^{R}(f'_k + 1) = f' + 1|Q'_k| = 4f'_k + 1 \tag{3.3}$$

By Equation 3.4 and the fact that ICP is a protocol which resiliency metric is at least $n \geq 4f + 1$ a phase of consense about the committees value must exist.

$$\sum_{k=1}^{R}(4f'_k+1) = 4\sum_{k=1}^{R}(f'_k+1)-3R = 4(f'+1)-3R = 4f'+1-3(R-1) \leq 4f'+1 \leq n \tag{3.4}$$

3.12 Recursively Applying Phase King

Another idea presented in [7, Ber89b] is to apply the *Phase King* algorithm with committees recursively to achieve nearly optimal round complexity $f + O(f)$ and asymptotically optimal total bit transfer. To achieve this a similar technique as presented in Section 3.9 is used. As before, the *Recursive Phase King* algorithm (RPK) executes in the same way as the Phase King algorithm, but splits the set of nodes $p \in Q$ into two equally sized subsets executing the kings broadcast. Therefore, new instances of RPK are executed using the two sets forming two committees Q_0 and Q_1. In each phase another committee is used as King, by applying RPK on the Phase King's committee set. This procedure is executed as long as the sets Q_0 and Q_1 are large enough to execute the round optimal consensus protocol ICP. During each recursion a Phase King algorithm is executed on the two currently used subcommittees.

$RPK(Q)$ executes the universal exchange for $n = |Q|$ nodes, in the succeeding rounds the algorithm $RPK(Q_0)$ and $RPK(Q_1)$ are executed, and so on. Since the sets are equally partitioned, every stage of the recursion needs $n + \frac{n}{2} + \frac{n}{4} + \dots$ rounds to execute the universal exchanges, and $f + 1$ additional rounds for the ICP, providing a nearly optimal reduction of the previously given number of rounds to $f + O(f)$. In summary, the *Recursively applied Phase King* algorithm requires $n \geq 3f + 1$ nodes, $f + O(f)$ communication rounds and a total bit transfer in the order of $O(nf)$. [7, Ber89b]

3.13 Phase King when Consensus is Repeadedly Needed

In the previous sections we discussed the requirement of consensus algorithms where a set of nodes has to agree on a set of values $\{0|1\}^n$. Generalizing the problem leads us to Multiconsensus, where agreement on a set of values $\{0|1\}^{n \times k}$ is required k times. [2, Bar95] presents an algorithm achieving it for $n \geq 3f + 1$ nodes and optimal amortized costs in all other measures, when k is sufficiently large. In this variant of the Multiconsensus problem each instance is started after the previous instance has terminated. The numbers of communication rounds r^*, the total number of bits b^* and the message size m^* for an instance of k-input sets is $r^* = O(1 + \frac{f}{k})$, $b^* = O(nt + \frac{nt^3}{k})$ and $m^* = O(1 + \frac{t^2}{k})$. As we can see, for $k \geq f^2$ this leads optimal lower bounds in all measures ($r^* = O(1)$, $b^* = O(nt)$ and $m^* = O(1)$). When restricting the message size to single bit messages, namely $m^* = 1$, algorithms have been developed providing $r^* = O(1 + \frac{f}{k})$ and $b^* = O(nt)$ but requiring an quadratic increase of nodes in f. [3, Bar91] [2, Bar95]

3.14 Results of Theoretical Analysis

Summing up, in this chapter we described the problem of reaching consensus and presented protocols solving the problem. Also a draft for the reduction of Triple Modular Redundancy and replica determinism to the consensus problem in Byzantine faulty environments was outlined.

Among the presented and theoretically analyzed protocols were:

1. Protocols requiring an exponentially increasing number of communication rounds like the Exponential Information Gathering protocol.

2. Protocols requiring polynomially increasing number of communication rounds like the Phase King protocol and the Phase Queen protocol.

3. Enhancements of these protocols enabling early stopping.

The theoretical results show that the early stopping variants of the algorithms only perform better than the algorithms themselves, if the number of faults specified (f) is high and the number of faults actually occurring (t) is much lower than f. Independent of the protocol the benefit of early stopping is only measurable for large f and relatively small t. Since in many systems it is more beneficial to guarantee a lower bound on the number of rounds a protocol executes than to optimize the average case further investigations have to be taken in the next chapter.

In addition, a divide and conquer strategy, namely, the committee technique has been presented. While the protocols using the committee technique perform better with respect to the number of rounds required if the message size is unrestricted, the unmodified protocols perform better when the message size is restricted to single bit messages. Since our implementation is restricted to single bit messages the committee technique is omited in the remaining simulations and evaluations.

CHAPTER 4

Simulations in Software

4.1 Purpose of a Software Simulator

After we have introduced some algorithms achieving consensus in a distributed system withstanding Byzantine faults Chapter 3, we are now going to simulate these in a software environment closely resembling the properties of our target environment. The purpose for these simulations is to get a better understanding of how the algorithms work in detail and to test their behaviour if used out of their specification. This enables us to easily explore their behavior in case the fault hypothesis is violated. Besides, we can determine the last round a decision is changed giving us a good indicator for optimizations.

In the next section (Section 4.2) we will give an overview of the simulation environment. We will start with describing the single parts of our environment and how they interact with each other. Afterwards we detail the adoptions of the algorithms to fit our environment. At the end of this chapter the simulation results for the previously shown algorithms are presented.

4.2 Big Picture

The simulator is developed as console application using C# and the .Net framework 2.0. Therefore, it is compatible for windows and any platform supporting Mono.

The simulator is based on the single shot principle. It therefore executes a single run of the algorithm at a time. Each run is described by a binary input vector, a list of faulty nodes and which of the algorithms should be used. (e.g. EIG, Phase King, Phase Queen). The simulation class handles the initialization of the system (including the creation of the nodes) as well as the simulation of a single run. The results of a simulation run are structured into the following items: (i) does the algorithm satisfy agreement, (ii) does the algorithm satisfy validity, (iii) what

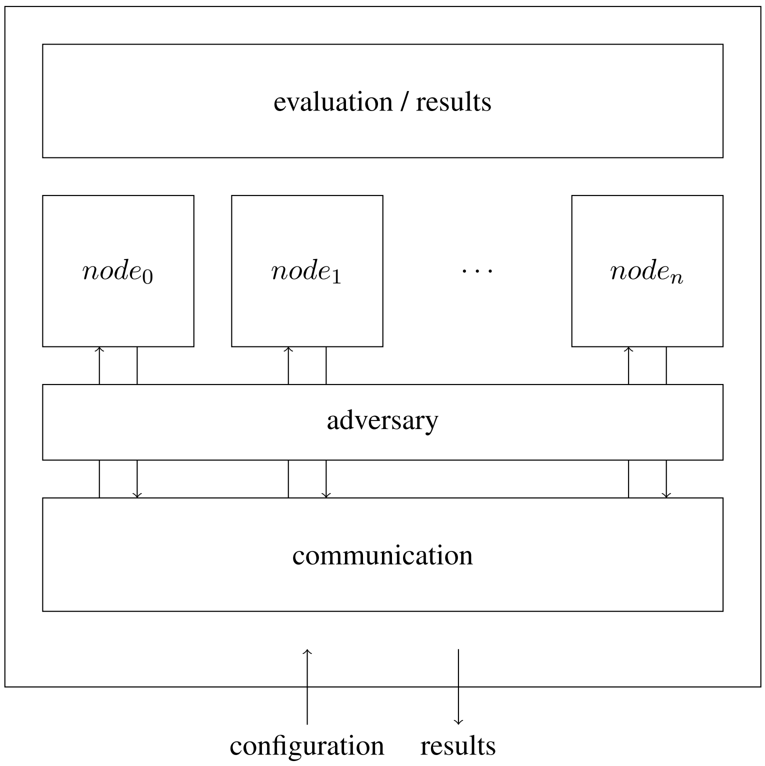


Figure 4.1: Structure of the SingleShotSimulation framework

decision value has been achieved, (iv) the time of termination of the nodes and (v) the latest change in the decision value for each node.

Communication during a single simulation run is implemented using a $n \times n$ matrix accessible to all nodes (as shown in Figure 4.2). For example: Writing a value into the first row and the third column of the communication matrix ($comm[0][2] = val$) corresponds to node p_0 sending val to node p_2. As our target environment only supports single bit, binary data transmissions, the entries of the matrix are implemented as boolean values. As each node is allowed to access its channels only, additional code in the simulator ensures, by checking the sender's and receivers' id, that no access violations occur. Therefore, only communication operations possible in the target environment can be executed in the simulator.

The round based model is implemented as a loop running until the last node terminates, as Algorithm 6 shows. The execution structure for the lock step synchronous model can be explained as follows:

1. First the *Receive* phase is executed, where the input vector of each node is updated by the communication which took place in the previous round. This is achieved by copying the corresponding column of the matrix into the input buffer of the nodes.

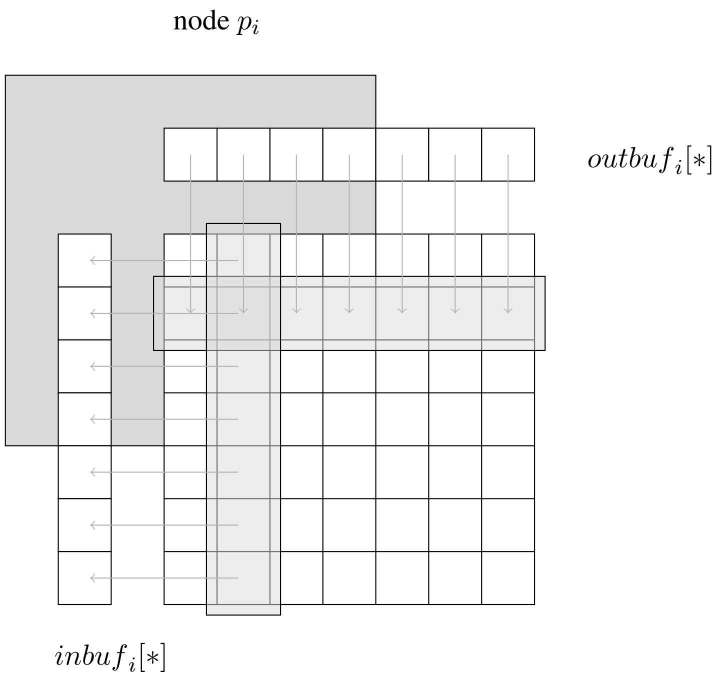


Figure 4.2: Communication primitive used during simulation

2. Next the communication matrix is reset. Therefore, already delivered values will have no effect on communication operations of later rounds.
3. Third, each correct node executes one round of its algorithm, while faulty nodes behave corresponding to their fault description.
4. Fourth, the *Send* stage of the round model is reached, forwarding the messages of the nodes outbuffers to the corresponding positions of the communication primitive.
5. In a last step the simulator checks whether all nodes have already decided. Based on this observation, the exit condition of the simulator can be calculated.

The node implementation class encapsulates the behaviour of a node participating in a certain implementation of an algorithm. The nodes initially have knowledge on their private value, their communication channels, the unique index assigned by the simulation environment and whether they are faulty. Additional properties inherent to the algorithm (like the number of faults to be tolerated) are also known by each node. For statistical purposes the node remembers the round number in which the last decison was taken, the round number in which the algorithm has terminated and its final decision value.

The functions Receive, Send, Run and ExecuteFault represent the node's public interface accessible by the simulation environment. The *Receive* function copies the corresponding col-

Algorithm 6 *RunSystem method of the SingleShotSimulation environment*

```
    Initialization:
 1: $round := 0$ {nr of the current round}
 2: $everyNodeTerminated := 0$ {have all nodes already terminated}
 3: $nodes$ {implementation of the nodes}

    LockStepSynchronous round model:
 4: while $\neg everyNodeTerminated$ do

      receive
 5:   for all $nodes : n$ do
 6:     $n.Receive()$

      execute
 7:   for all $nodes : n$ do
 8:     $n.Run(round)$
 9:     $n.ExecuteFault(round)$ {here certain faults are induced to the system}

      send
10:   for all $nodes : n$ do
11:     $n.Send()$

      exit condition
12:   $everyNodeTerminated := 1$
13:   for all $nodes : n$ do
14:     if $\neg n.HasTerminated$ then
15:       $everyNodeTerminated := 0$

16:   $round := round + 1$
```

umn of the global communication primitive into the internal receive buffer of the node, while the *Send* function copies the output buffer into the correct row of the communication matrix. The *Run* method executes a single round r of the implemented algorithm. To simulate faults, an adversary is implemented using the *ExecuteFault* method. If, at initialization, the node was marked faulty by the simulation environment, the adversary can manipulate the output buffer of the node. In the current implementation this is based on a random number generator. The generated random numbers are used to define the faulty rounds as well as to determine which bits of the output buffer are to be flipped.

To automate the simulation procedure even further, a simulation factory has been created. It exhaustively generates simulations for each combination of faulty nodes and input vectors (as shown in Figure 4.3). After calculating the list of the $\binom{n}{f}$ node fault masks the specified algorithm is executed for each combination of the fault masks with one of the 2^n possible input combinations. The runtime behaviour of the *SimulationFactory* is therefore exponential in n and f. After each execution the simulation results are gathered and evaluated.

In the following sections we present the transformations required to execute the algorithms presented in Chapter 3 within our target environment.

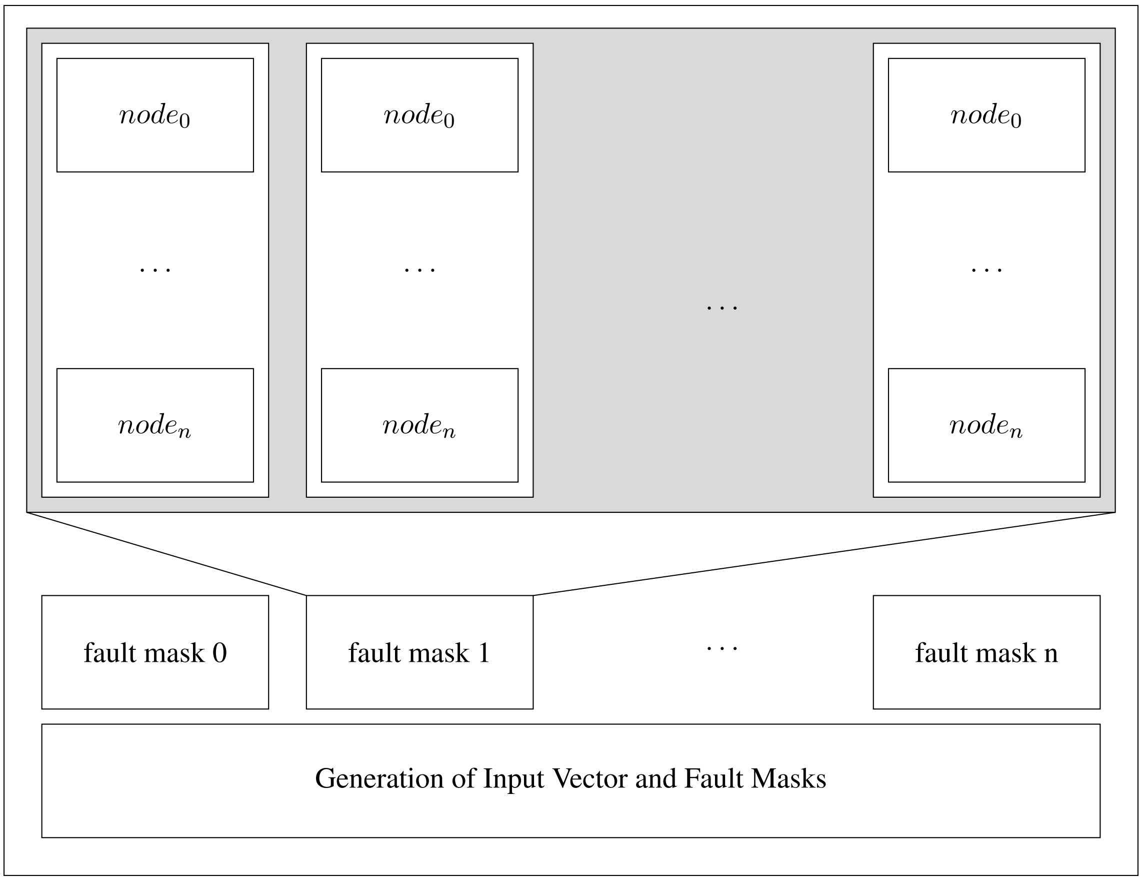


Figure 4.3: Big picture of the simulation environment

4.3 Transformation of EIG for Single Bit Messages

Exemplary for all transformations required, we describing the transformation of the Exponential Information Gathering algorithm (see Section 3.6) to single bit messages. This is required as our target environment, and therefore our simulator, can only handle one single bit message in any round. The *Exponential Information Gathering* protocol forces an exponential increase of communication bits with respect to n. Figure 4.4 outlines the message size needed for the algorithm with given f in a certain round r. Therefore, we have to create a transformation of the original algorithm's message size to our model.

Since in round 1 each node sends its own value to all the other nodes the communication effort is exactly 1 bit, thus the message size in round 1 is 1 bit. As in round 2 each node broadcasts all, except its own, values received in the previous round, the message size is already $n-1$ bits. In the next round every node broadcasts the values received in the previous round except the one received from itself. Leading us to $(n-1)(n-2)$ bits per message and so forth. Based on this observation, the message size for round r is given by Equation 4.1.

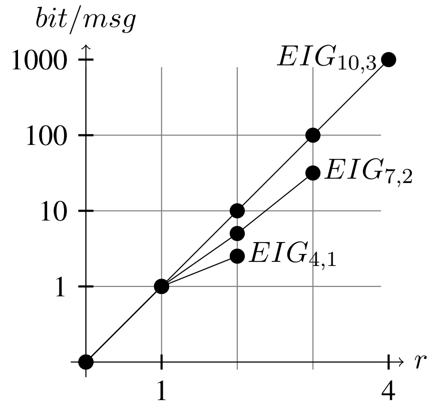


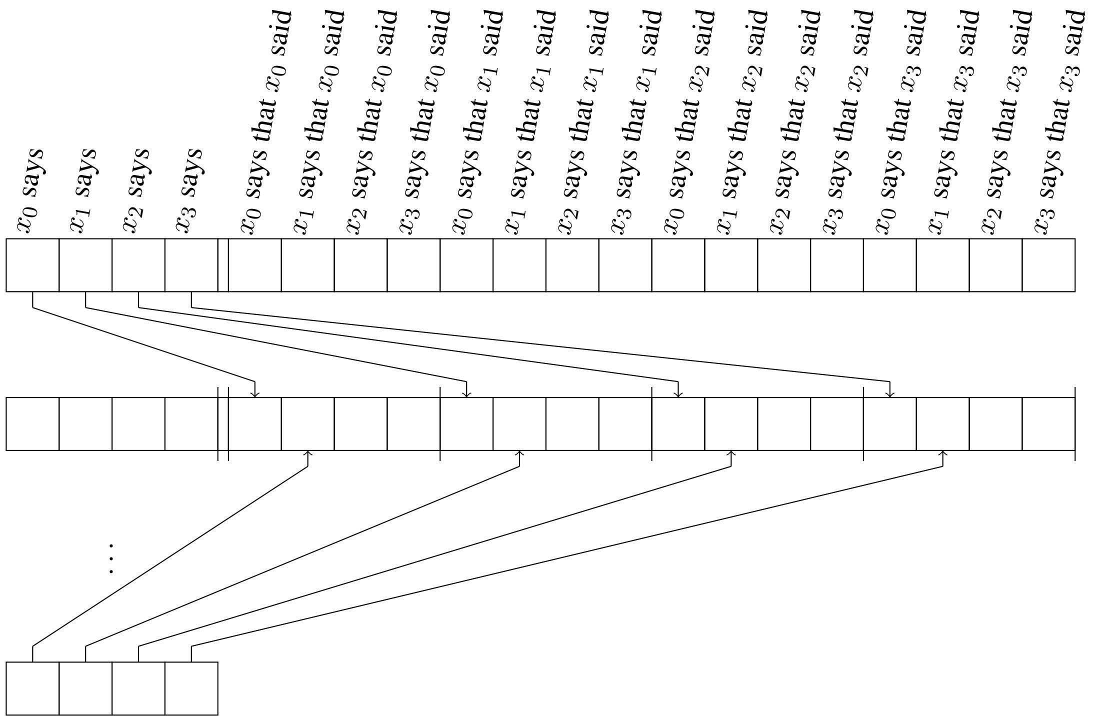
Figure 4.4: EIG: Increase of message size per round

$$M_{size}(r) = \prod_{k=1}^{r} (n-k) \tag{4.1}$$

The simplest transformation is to increase the number of rounds such that all message bits can safely be transmitted using single bit messages. Therefore, after transforming the *Exponential Information Gathering*, its measure for the message size converts to single bit messages, whereas the *number of rounds / time complexity* and the *message complexity* increase to $\mathcal{O}(exp(n))$. To enable a more uniform simulator design, however, we have decided to also transmit the messages received from the node itself in every round. This increases the time and message complexity slightly compared to the optimum implementation.

The information gathered within the rounds is stored in a linear memory array. Each entry represents a unique item of the tree. The first n positions in the memory store the messages received in the first round. From round 1 onward each node broadcasts a bit of information gathered in the previous rounds to the other nodes. A detailed outline of how this transmission of information is executed is shown in Figure 4.5. After all the information has been gathered the resolve function $CalculateDecision()$ is used to calculate the final decision of the node Y. The resolution process is solely based on the information stored in the linear memory array. Therefore, $CalculateDecision()$ has to identify which values in the array represent valid decision tree entries and which have to be ignored. Consider, e.g., a tree of height two. The first entry of the second level (p_0 said that p_0's value is) will not be used by the algorithm to deduce the decision value and has therefore to be ignored. A similar argument can be used for all other nodes in the system. Since f and n are known in advance, the positions to be skipped (and therefore also the valid positions) are known at design time. The resolve function for $f = 1$ and $n = 4$ is shown in Figure 4.6.

As the transformations of the other algorithms simulated are straight forward, we do not



x_0 says
x_1 says
x_2 says
x_3 says
x_0 says that x_0 said
x_1 says that x_0 said
x_2 says that x_0 said
x_3 says that x_0 said
x_0 says that x_1 said
x_1 says that x_1 said
x_2 says that x_1 said
x_3 says that x_1 said
x_0 says that x_2 said
x_1 says that x_2 said
x_2 says that x_2 said
x_3 says that x_2 said
x_0 says that x_3 said
x_1 says that x_3 said
x_2 says that x_3 said
x_3 says that x_3 said

Figure 4.5: EIG: Example usage of linear memory used during information gathering

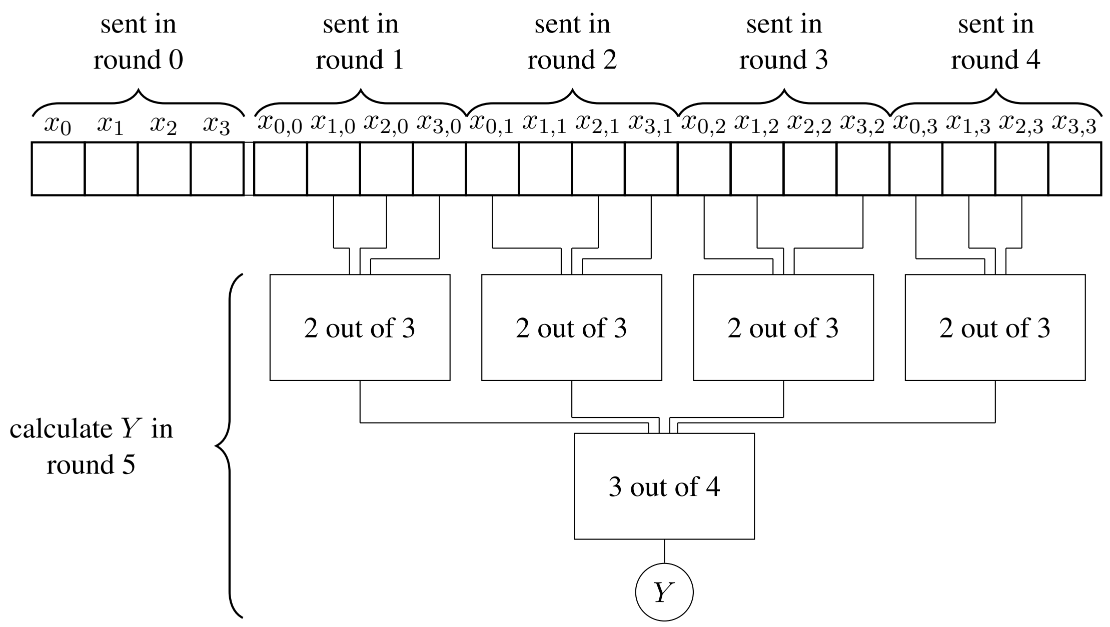


Figure 4.6: EIG: Example of the resolve function for $n = 4$ and $f = 1$

detail them further. In the following Section 4.4 we present the simulation results and, based on the findings, select algorithms for the evaluation in hardware.

4.4 Simulation based Comparison

In this section the results of the protocol simulations are presented. For our evaluation about 10^9 scenarios have been executed. The fault injection of our tests have been parameterized with the number of nodes (n), the number of faults the implementation is able to handle (f) and the number of faults actually occurring (t). The faulty nodes of a simulation run are calculated offline before its start. The experiments were designed to be exhaustive, therefore each possible combination of faulty nodes will be simulated. The faulty nodes are allowed to completely modify their output buffers. The degree of output buffer corruption is generated randomly for each round, based on a normal distribution of the faults. The simulations are designed to augment the theoretical reasoning on the correctness of the algorithms.

Additionally, the implementation of the algorithms in the simulation environment can be used as a reference for estimating the complexity of the hardware implementation and the required hardware resources.

In Table 4.1 the results for the Exponential Information Gathering (EIG) protocol are listed. Only the more interesting parameter combinations are listed. The results for cases where the number of actual faults is higher than the number of tolerable faults have to be considered carefully, as the properties validity and agreement are only defined for correct nodes. If, e.g., all nodes in the system are fault, the correctness of the run will be 100% as no correct node has violated agreement.

If the number of tolerable faults exceeds one ($f > 1$), the number of required rounds is quite high (in fact the EIG has the worst round complexity of all the investigated algorithms in this case). For $f = 1$ on the other hand, the round complexity is quite good. A decision is reached in 5 rounds. Even if the single tolerable fault is slightly exceeded ($t = 2$), still 81% of the tested scenarios were decided correctly. For $f = 1$ and $n = 4$ 20 memory bits are required for storing the decision tree. The implementation of the resolve function as well as the address generation for accessing the tree during the information gathering phase have been identified to be the most expensive parts of a potential hardware implementation.

The results for the algorithms Phase King, Early Stopping Phase King, Phase Queen and Early Stopping Phase Queen are presented in Tables 4.2 - 4.5. As we can see, the results regarding the last round of termination match the theoretical results given in Tables 3.3 - 3.5.

As presented in Table 4.2, the Phase King algorithm requires a moderate number of 8 rounds of execution, given $f = 1$, while it would take even 10 rounds of execution in case of early stopping. The simulation of the Phase King algorithm outside its specification leads to a correct

Protocol	n	f	t	runs		last round of	
				correct(%)	faulty(%)	decision	termination
EIG	4	1	0	100	0	5	5
EIG	4	1	1	100	0	5	5
EIG	4	1	2	81	19	5	5
EIG	7	2	0	100	0	57	57
EIG	7	2	1	100	0	57	57
EIG	7	2	2	100	0	57	57
EIG	7	2	3	81	19	57	57
EIG	7	2	4	69	31	57	57

Table 4.1: Simulation results for Exponential Information Gathering

Protocol	n	f	t	runs		last round of	
				correct(%)	faulty(%)	decision	termination
PK	4	1	0	100	0	3	8
PK	4	1	1	100	0	8	8
PK	4	1	2	87	13	8	8
PK	7	2	0	100	0	3	12
PK	7	2	1	100	0	7	12
PK	7	2	2	100	0	12	12
PK	7	2	3	81	19	12	12
PK	7	2	4	78	22	12	12

Table 4.2: Simulation results for Phase King

result in 87% of all cases, which is quite good, compared to the EIG implementation where 81% of all the cases were correct.

Like the EIG protocol it requires $n \geq 3f + 1$ which is optimal. The hardware implementation costs of the Phase King algorithm are lower than the ones for the EIG protocol. As the algorithm has an increased number of rounds and requires counter for detecting the strong preference it has a slightly higher implementation cost as the Phase Queen one. The early stopping variant requires two additional rounds of communication for $f = 1$ and achieves a correctness of 82% if $t = 2$, which is slightly worse than the unmodified algorithm. Therefore, the Phase King algorithm suits better for the implementation in hardware than its early stopping variant.

According to the simulator implementation of the Phase Queen algorithm, it requires less hardware than the other two protocols. Further, it decides in the least number of communication rounds. Additionally, the simulations' results show that the protocol performs best regarding the correctness of the protocols when used outside their specification, but it requires a higher number of nodes to tolerate f faults. In the case of $f = 1$ the Exponential Information Gathering

Protocol	n	f	t	runs		last round of	
				correct(%)	faulty(%)	decision	termination
ESPK	4	1	0	100	0	3	10
ESPK	4	1	1	100	0	10	10
ESPK	4	1	2	82	18	10	10
ESPK	7	1	0	100	0	3	10
ESPK	7	1	1	100	0	8	10
ESPK	7	1	2	73	27	10	10
ESPK	7	2	0	100	0	3	10
ESPK	7	2	1	100	0	8	15
ESPK	7	2	2	100	0	15	15
ESPK	7	2	3	79	21	15	15
ESPK	7	2	4	74	26	15	15

Table 4.3: Simulation results for Early Stopping Phase King

Protocol	n	f	t	runs		last round of	
				correct(%)	faulty(%)	decision	termination
PQ	5	1	0	100	0	1	4
PQ	5	1	1	100	0	4	4
PQ	5	1	2	93	7	4	4
PQ	5	1	3	88	12	4	4
PQ	9	2	0	100	0	1	6
PQ	9	2	1	100	0	4	6
PQ	9	2	2	100	0	6	6
PQ	9	2	3	92	8	6	6
PQ	9	2	4	85	15	6	6
PQ	9	2	4	70	30	6	6

Table 4.4: Simulation results for Phase Queen

algorithm, the Phase King algorithm and the Early Stopping Phase King algorithm require 4 nodes, while the Phase Queen algorithm and its early stopping version require 5 nodes. As an increased number of nodes normally also mean an increased number of replicated application hardware, the costs of the Phase Queen algorithm are too high for our purpose.

Like the Phase Queen algorithm, the Early Stopping Phase Queen algorithm requires 5 nodes in the case of $f = 1$, but needs two more rounds of communication. Additionally, the performance when used out of specification is worse than observed during simulations of the Phase Queen algorithm. Taking this into account the Phase Queen algorithm and its early stopping version do not suit very well to our requirements and therefore we restrict our further evaluation on algorithms using $n = 3f + 1$ nodes.

Protocol	n	f	t	runs correct(%)	runs faulty(%)	last round of decision	last round of termination
ESPQ	5	1	0	100	0	1	6
ESPQ	5	1	1	100	0	4	6
ESPQ	5	1	2	75	25	6	6
ESPQ	5	1	3	57	43	6	6
ESPQ	9	1	0	100	0	1	6
ESPQ	9	1	1	100	0	4	6
ESPQ	9	1	2	95	5	6	6
ESPQ	9	2	0	100	0	1	6
ESPQ	9	2	1	100	0	6	9
ESPQ	9	2	2	100	0	9	9
ESPQ	9	2	3	80	20	9	9
ESPQ	9	2	4	61	39	9	9
ESPQ	9	2	4	47	53	9	9

Table 4.5: Simulation results for Early Stopping Phase Queen

Summarizing, given $f = 1$ and a minimum number of required nodes $n = 4$, the Phase King algorithm performs best. Therefore, it is the best choice for our hardware implementation. To have a baseline for comparison to [25, Pol09], additionally the EIG protocol is implemented in hardware.

CHAPTER 5

Hardware Framework

5.1 Introduction

In this chapter we will introduce the hardware framework building the basis for our consensus implementations. In Section 5.2 and 5.4 we introduce the major problems inherent to high performance circuit design due to (i) the limited timing margins available, (ii) the impossibility of accurate delay prediction, (iii) the problems occurring if components are used outside their specification and (iv) the problem of crossing clock domain boundaries. In Section 5.3 the most widely used clocking paradigms are introduced and their advantages as well as their disadvantages are outlined. This is followed by the presentation of a communication buffer for multisynchronous systems developed by [25, Pol09] in Section 5.6. This communication primitive forms the basis for our further implementations. On top of it the round generation model is implemented as described in Section 5.7.

5.2 Delay, Skew, Runts and Glitches

One of the major challenges when designing hardware is the unpredictability of signal delays. Due to the influence of process variations, changes in supply voltage or in the temperature, as well as data dependent delays, such as gate delays or interconnect delays, the exact delay of a signal path cannot be predicted deterministically. Thus, we can assert that the delay on individual paths cannot be assumed to be equal. [34, Wak00]

The maximum delay deviation between two signal paths is called the skew. The larger the skew of two correlated data lines is, the less timely correlated are the informations. Therefore, skew may cause inconsistent interpretation of the signal at the inputs of different logic gates and therefore lead to an invalid dynamic state. Even worse, if we capture such an invalid dynamic state, the steady state of the circuit may also be compromised. Dependent on the change of the

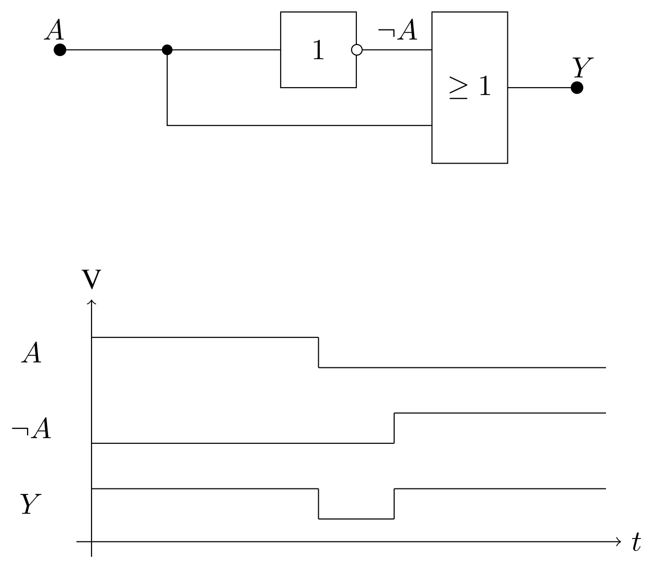


Figure 5.1: Example of a static-1 hazard

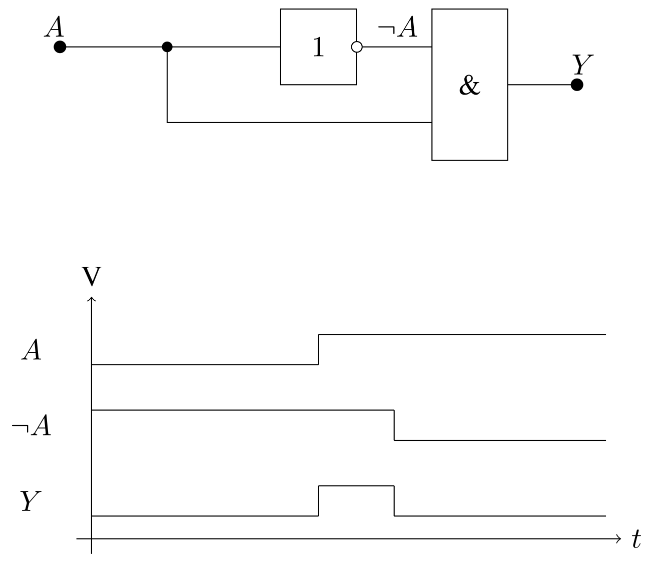


Figure 5.2: Example of a static-0 hazard

input signal and the response of the output signal, we distinguish between static-0, static-1 and dynamic hazards. The manifestation of a hazard in the physical implementation of a circuit is called a glitch. [32, Spa01] [34, Wak00]

In Figure 5.1 a static-1 hazard is depicted where a change of the input values forces the output to shortly go to zero while the start and the end value of the operation are one. This

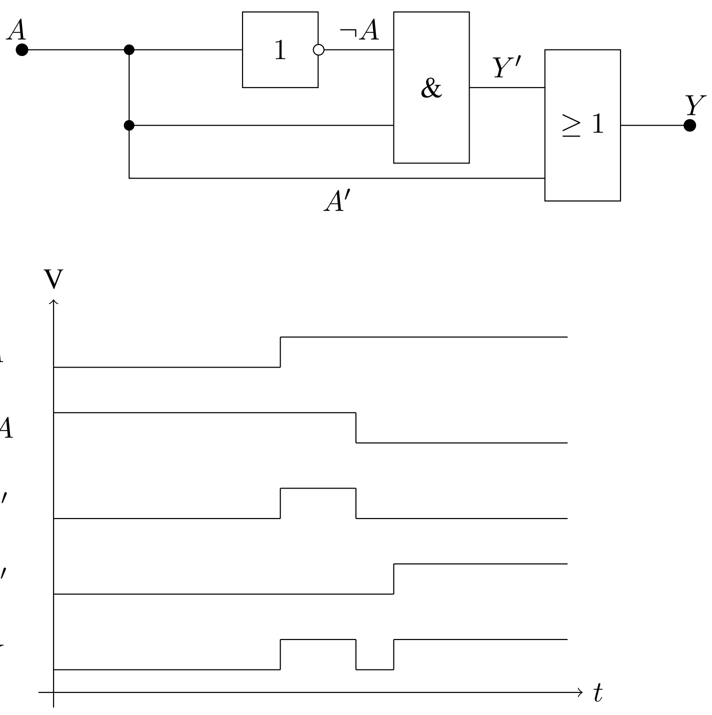


Figure 5.3: Example of a dynamic hazard

problems occurs because of the skew in the data lines. Therefore, a short interval in time exists where both input signals are erroneously different. A similar situation can be observed with the static-0 hazard in Figure 5.2, where the output should remain zero but a positive glitch is created. A dynamic hazard is outlined in Figure 5.3. Depending on the structure of the circuit, a positive or a negative glitch may occur before the real output transition. The first part of the circuit (the inverter and the *AND* gate) is called the glitch producer, creating static-0 or static-1 hazards. The second part, the *OR* gate, is called the edge producer. The situation may even be worse. The glitches may be too short to reach their full amplitude (called runts) and therefore violate the digital abstraction, leading to hard to detect problems in the circuit.

Summing up, skew cannot be predicted precisely, which may lead to inconsistent transient states. If these intermediate states are latched by the circuit, wrong data may pollute even the steady state of the circuit. As not all glitches can be avoided by design, a suitable mechanism for latching stable signals only is required. In Section 5.3 a selection of possible design styles solving the problem will be given.

5.3 Clocking Schemes

The main elements of sequential circuits are storage blocks (like flip flops or latches). They retain the state of the system. The functionality of the circuit is, however, implemented by combinational gates (like AND-gates, OR-gates and Inverters). The combinational logic uses the current state stored in the sequential element and the input values of the circuit to calculate the next state. To implement a consistent behavior throughout the system, the sequential elements have to be coordinated to guarantee a clean state change. The coordination of these components is called the timing of the circuit. A common way to implement the timing is to introduce an additional signal, the so called clock signal, which is used to trigger the storage operation of the sequential elements. In the following, we will outline some common design styles used in state of the art high speed circuit designs. Design styles can be evaluated by different properties such as the area overhead introduced, the support for composability, the level of testability, the robustness of the circuits, the power consumption of the circuit as well as the maximum speed (frequency) that can be achieved within a circuit. The main tasks of a clocking scheme are to indicate a safe time for an information sink to save a value without capturing an intermediate state and the time the information source may provide a new value after the sink has consumed the previous.

Synchronous Design

The synchronous design paradigm is by far the most widely used one today [32, Spa01]. It uses a central clock source to coordinate all sequential elements throughout the circuit. This central clock source enables the design tools to analyze the complete circuit and to determine the maximum speed it can be safely operated with. On the other hand, exactly this clock distribution forms the major drawback for todays high-frequency circuits. The distribution of the clock signal with a minimum skew throughout the whole chip is quite challenging or even impossible for larger chips [36, Zhu02]. To be able to drive the vast amount of inputs connected to the clock net, the clock buffers must be very strong and therefore the power consumption of the clock net is very high. [17, Mat03]

Fortunately modern design tools are very advanced and powerful enabling us to still cope with current design challenges. As long as the design's constraints and timing margins can be met, another tremendous advantage is that in most cases the static timing analysis makes a special hazard analysis unnecessary, since a propagation of erroneous signals to a steady state is prevented by design. [34, Wak00]

Nevertheless, the effort in routing the clock tree and the introduction of a single point of failure due to the single clock source, makes the synchronous design approach unsuitable for our needs.

Asynchronous Design

In the asynchronous design paradigm no shared clock signal is used. Thus, many limitations of the synchronous design can be overcome when using the asynchronous style. Nevertheless many new challenges have to be solved. As already discussed the main challenges for a timing paradigm are to determine the time an input signal can be safely captured and the time an output signal may be changed without interfering with the sinks, respectively. Asynchronous circuits used the most natural mechanism to achieve timing closure, namely handshaking. The source tells the sink, using a dedicated request signal, that new data is available, while the sink uses an acknowledge signal to flag the reception of the data. Two main strategies exist for implementing handshaking [32, Spa01]. The first one is called bundled data [32, Spa01]. It assumes that the data and the request line have similar delays and therefore the signaling of data validity can be safely implemented using a dedicated request signal. Note that the timing constraint for the request signal are much easier to guarantee as for a clock signal in the synchronous case since the request signal is a local signal only. The second approach uses specialized coding [32, Spa01]. Therefore, the request and the data are combined into a multi-rail signal. This enables the sink to detect the skew on the signal lines and it can wait until the complete multi-rail signal has arrived safely. The big advantage of handshaking is that a closed loop control scheme is used. Therefore, the sink is able to execute back-pressure, if the data source is too fast.

Summing up, in asynchronous design we exchange the global clock signal with explicit handshaking converting from an open loop to a closed loop circuit. The advantage gained, in return, is a self regulating data flow based on a direct relation between validity and consistency of data and therefore no delay and timing assumptions are required. The major drawback, besides the intellectual challenges and the lack of tool support is that, due to the lack of a global time base, many problems (including consensus) do not have a solution in a fully asynchronous system [12, Fis85]. Therefore, the asynchronous paradigm is not suitable for our implementation.

Globally Asynchronous Locally Synchronous Design

System based on the *Globally Asynchronous Locally Synchronous (GALS)* design principle consist of islands designed with the synchronous design approach, but uses asynchronous communication in between. Thus, we can use the models and tools known from the synchronous world to design the islands, and, second, have the flexibility of the asynchronous paradigm when composing the different modules to a complete system. Since each of these synchronous islands has its own clock domain, the clock tree complexity is decreased substantially. This also leads to a reduction of power consumption, as well as to higher robustness against electro magnetic interferences. [36, Zhu02]

The major drawback of this design style is an increase of communication effort between the components. Typical concepts for communication include asynchronous communication channels with full handshaking [36, Zhu02] and pausible clocking [35, Yun96]. The sender and the receiver use their restrictive clock domains. To safely transport data between them, the handshake signals must be synchronized [36, Zhu02] to avoid disambigues data to be received. The second concepts overcomes the need of synchronizers in the handshake signals by stopping the receiver clock domain while data is transmitted. On the reception of the data, the receivers clock is reactivated and it can process the newly arrived data.

Mesochronous Design

Mesochronous design is a derivation of the synchronous design principle. In mesochronous designs we can assume multiple synchronous islands as in the GALS approach. The difference between GALS and mesochronous systems is that, while the clock relations in GALS systems are unknown, in mesochronous systems the clocks of all islands have a bounded phase relation. A commonly seen example for bounded phase relations are *Phase Locked Loops (PLLs)* driven by the same source. While, commonly a PLL generates a nearly exact phase shifted signal of the source, in mesochronous systems the phase alignment of the clocks is much more relaxed (up to several clock cycles).

By the assumption of identical frequencies of the clocks and varying but bounded phase relationships, we can use the models and tools from the synchronous design paradigm here, while lowering the efforts in synchronizing the inter module communication. By the bounded phase relation we can design communication buffers to compensate for the phase differences and enable a safe communication.

5.4 Metastability

When building digital circuits it is assumed that only two voltage levels may occur in the system, namely high and low. In physical implementations, however, these two states have to be represented by analogue voltage ranges, leaving a forbidden range between them. It is assumed that signals only cross this forbidden range while switching from one state into the other and therefore will only be in this area for a short timespan. As we already have discussed this abstraction may not be valid, if runts occur.

In this section we will introduce an even more problematic situation, namely metastability. If a circuit latches a signal shortly before or after a transition, the time to decide if the old or the new value is correct will increase significantly. During this timespan the signals of the memory element may stay in the forbidden voltage range. [34, Wak00]

The consequence of this out of specification usage is called metastability. It constitutes one of the major threats when designing high performance circuits. As combinational gates may map metastable inputs to metastable outputs, metastability may propagate through the whole system. Other manifestations of metastability can be late transitions, glitches or even oscillations [25, Pol09] [18, Kle87] [9, Cha73]. Late transitions may occur, if the intermediate voltage level of the memory element is not interpreted as signal change by the next stage and therefore a signal transition is only generated after metastability has resolved. This behavior sounds quite benign, however, it increases the signal delay to the next memory element and the timing analysis of a synchronous system may no longer be valid and the next memory element will become metastable as well. Glitches may occur if, in contrast to the late transition case, metastability is interpreted as signal change by the following stage. In such a case a signal transition will occur at the beginning of the resolution process and, if the memory element decides to the old value, a second transition will be created at the end of the metastable state leading to a glitch [9, Cha73]. As can be seen, metastability may be interpreted differently by two successor stages (as glitch or as late transitions) and therefore may lead to an inconsistent system state not considered at design time. Lastly, oscillations may occur, if the rise and fall time of the signals is much shorter than the internal delay of the memory element [18, Kle87]. As this is normally not the case in current CMOS technology, we will not go into further details here.

Why are we concerned with metastability? In a sufficient well designed circuit such a phenomenon should normally not occur. Unfortunately, on the boundaries of the circuits the data signals are not in the sphere of control of our circuit and therefore metastability will surely occur. Current countermeasures, like synchronizers, [9, Cha73] are mainly based on statistically increasing the time between two observable metastable states but do not remove the problem completely [9, Cha73].

In summary, basically all bistable components can suffer from metastability when they are used out of their specification. Metastability can therefore not be eliminated but it can be made less probable by careful design.

5.5 Architectural Design

This section gives an overview on our design framework and the way it is composed out of its different components. Further, we describe the used test environment. A detailed discussion of the design units is given in Sections 5.6 and 5.7. As depicted in Figure 5.4, the test environment used consists of three seperate *Field Programmable Gate Arrays (FPGAs)*. *FPGA A* is used as a dedicated controller FPGA managing the clock generation of the system, while the other FPGAs, B and C, are hosting the four nodes of our system. The framework presented in the following sections is similar to the one used in [25, Pol09].

As we want to evaluate protocols prone to Byzantine faults, we also have to provide a framework which can handle these kind of faults. Therefore, we decided against using a synchronous design and employ a mesochronous one instead. The mesochronous clocks may be created using the *Distributed Algorithms for Robust Tick Synchronization* (*DARTS*) protocol [14, Fue06] [29, SSS07]. The DARTS protocol generates ticks with bounded phase shift. While other approaches use a single tick and propagate it with great effort throughout the whole system, DARTS uses a distributed, Byzantine fault tolerant tick generation algorithm with dedicated tick generators at each node. Therefore, no global clock tree exists, eliminating this single point of failure. In

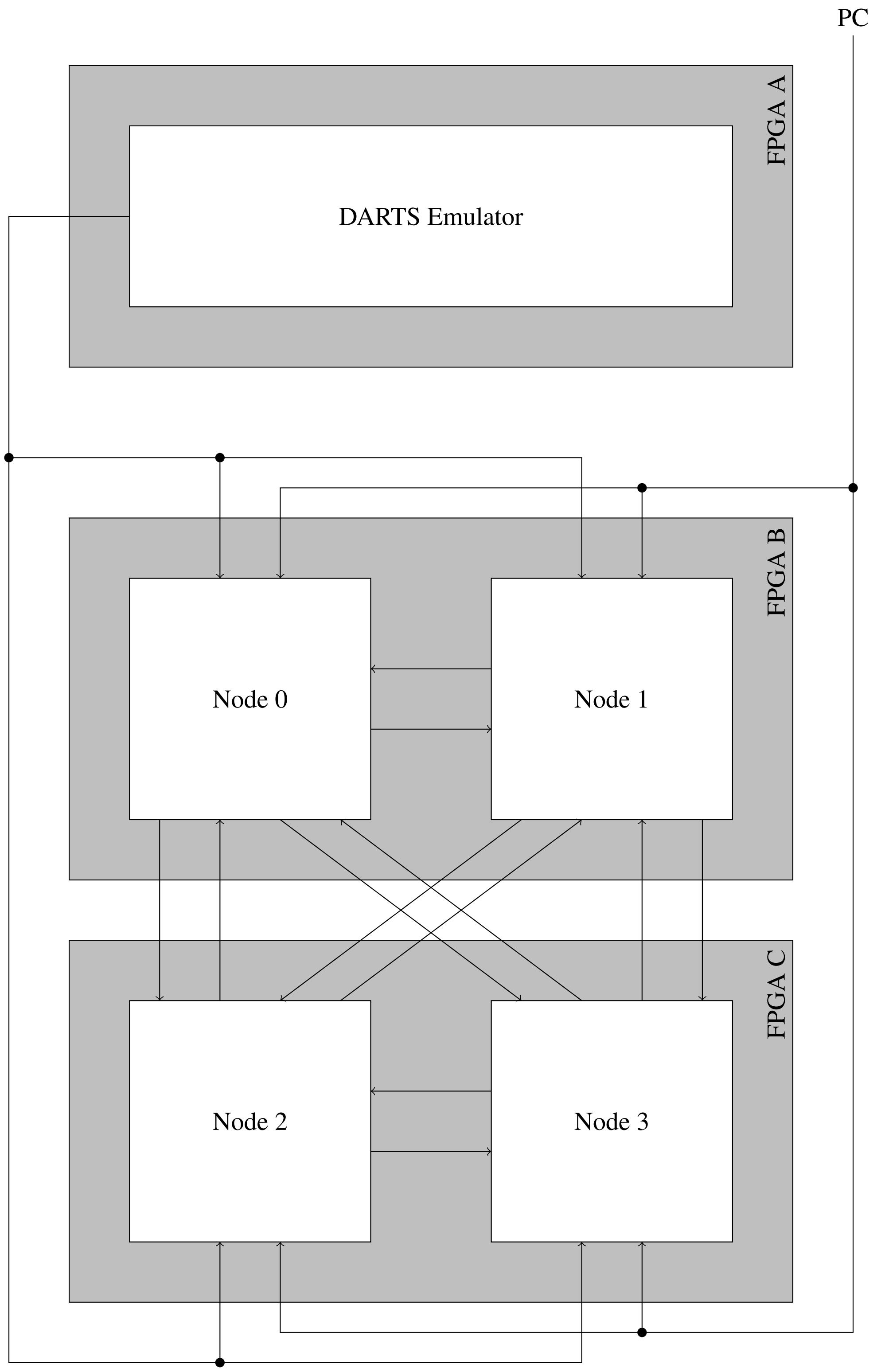


Figure 5.4: Overview of hardware framework

contrast to a GALS system, the tick generation on each node is not simply done by an oscillator, but with a fault tolerant algorithm providing a fixed bounded phase shift. This algorithm can tolerate up to f faults for a number of at least $n \geq 3f + 1$ nodes. This is a suitable requirement for all the protocols under concern in this thesis. [14, Fue06] [29, SSS07]

As our development hardware is off the shelf, it was quite hard to integrate the DARTS chips into the development environment. Therefore, we decided to use a simulation of the DARTS capabilities implemented on FPGA A taken from [25, Pol09]. The simulator centrally creates ticks for each FPGA, while adhering to the phase shift constraints of the original DARTS algorithm. The generator is configured using a dedicated memory. The memory can be filled by a monitoring PC before the experiments start. An important advantage of the simulation over the real DARTS environment is that the same pattern can be replayed multiple times making debugging problems much easier.

Each of the FPGAs B and C contains two nodes. Namely FPGA B contains node 0 and node 1, while FPGA C contains node 2 and 3. Communication between the nodes takes place via a dedicated, fully connected network. Thus, each node is equipped with four transmitter and four receiver components. These are interconnected with each other forming $n \times n$ unidirectional communication links. The compensation of the phase shift between the different clock domains is performed by a communication buffer within the receiver components. The way the communication buffer works is elaborated in Section 5.6.

The communication with the monitoring PC is implemented via USB. Each configuration register and memory block is mapped to a dedicated address which can be accessed by the PC. Thus, we can read and write memory blocks on each FPGA from the PC. This grants us the possibility to configure the above mentioned clock generators, as well as the input values of each node. For better evaluation and testing of the protocols, we further support the configuration of saboteurs from the software during runtime. Therefore, it is possible to set one or more nodes faulty. We support multiple fault scenarios for the nodes:

1. Set the communication links between two FPGAs to High Z. This simulates a floating output, where neither the logical level 0 nor the logical level 1 is driven.

2. Set the communication links to 0 or 1. This simulates stuck at 0 and stuck at 1 faults, respectively.

3. Invert all bits on a line. This method is available either for the data path and the clock path, respectively.

4. Send the same signal on the data path as well as on the clock path. This simulates a bridging fault. The signal to be replicated can be either the clock or the data signal.

5. Sending inconsistent values to neighboring nodes. This mode represents the typical Byzantine behavior.

To check the results of the consensus protocols we also provide a memory region storing the expected output values. After each run of a protocol we crosscheck the results with the values given in this buffer and increment an error counter if they do not match.

5.6 Communication Buffer

We have already outlined the major problems that have to be solved when communication is required (see Sections 5.2 and 5.4). Synchronous design solves this problem by enforcing that a send operation taking place at clock tick C_i^k on node i is read at the next clock tick C_j^k at node j. It is guaranteed by design that the setup-/hold-window of the receiver will not be violated.

Generally, this assumption can not be maintained for GALS systems. Fortunately our mesochronous clocking scheme provides more guarantees as a plain GALS systems. By exploiting the bounded phase relation, we can use a relaxed version of the synchronous communication principle. The solution requires that we wait a sufficient number of clock cycles before evaluating the sent data at the receiver. From the viewpoint of throughput, this solution is quite similar to the introduction of macroticks and microticks. Here a microtick equates a tick of the native clock, while a macrotick is the already synchronized clock with a precission $\pi \leq 1$. Both variants lack in throughput of data at the cost of high clock frequencies required.

By applying a sufficiently large ring buffer, the clock skew can be tolerated without sacrificing throughput. In [26, PHS09] a solution with full throughput is presented. For this purpose, separate read and write addresses are used within the ring buffer. Thus, if the buffer is sufficiently large and the offset between the read and write addresses is sufficiently spaced, metastability or inconsistencies can no longer occur. In contrast to the macrotick approach, the throughput is not decreased while the system is still provable correct.

As outlined in Figure 5.5 the communication subsystem consists of three major components. The transmitter operates as a peripherial slave within the clock domain of the senders application logic. The application can transfer data by passing it via an input register. It is the responsibility of the transmitter to fill in idle patterns in case no new data is provided. After new data has been applied it is 8b/10b encoded and serialized to the receivers communication buffer. The communication buffer is part of the receiver. While the decoder and the output register are completely within the sphere of the receivers clock domain, the communication buffer is split into two parts. On the one side, the logic used for reading the data out of the ring buffer is controlled by the receivers clock, while on the other side the logic used for writing data into the buffer is within the sphere of the sender. To compensate the clock skew of these two clock domains a sufficient buffer size has to be applied. Therefore, the buffer size and the address margins calculated are of major importance to avoid metastability. The size and safety margins for the buffer have to be calculated at design time.

As stated in the previous section, we use an emulation of the DARTS clocking algorithm. Thus, the communication buffer used in this environment has to face the same constraints as a

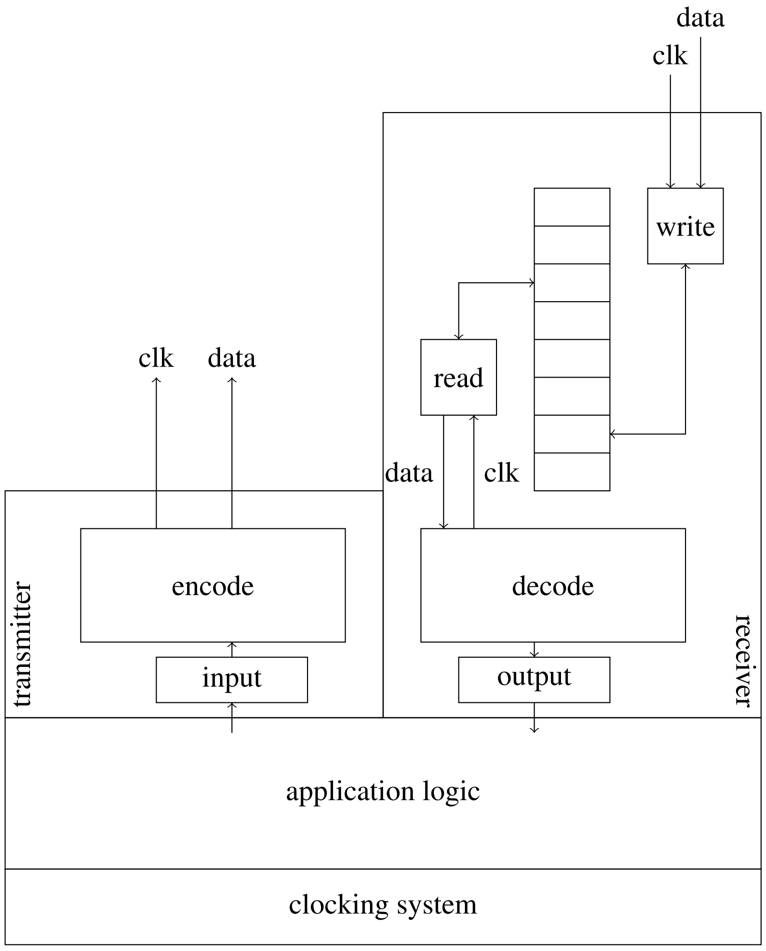


Figure 5.5: Scheme of the communication subsystem

communication buffer used with DARTS. By the precedence relation introduced in Section 2.6 we can restate the problem of misaligned clocks to Equation 5.1.

$$\forall i, j \in P, \forall k > 0 : C_i^k \rightarrow C_j^{k+\pi} \tag{5.1}$$

Equation 5.1 gives the precision constraining the clocking system. Stating that no two clocks within the system are more than π ticks apart. As we have described above, a major concern in communication is the guarantee that (i) no new write action to a memory of the buffer takes place before this location has been read and that (ii) no memory location is read before data has been written to it. Note that the ring buffer is initially prefilled. Thus, the initial values in the buffer are considered as valid write operations.

To support the two properties given above we have to pair the write and read transitions. Based on the proofs of [26, PHS09], we can state a relation between a read and the corresponding write operation (Equation 5.2).

$$W_i^k = C_i^k \rightarrow C_j^{k+\alpha} = R_j^k \tag{5.2}$$

Further, [26, PHS09] has shown that a memory element has safely been read before it is overwritten, if Equation 5.3 holds. By this, α, as given in Equation 5.4, determines the start address of the first write operation, while β, as given in Equation 5.5, gives a bound on the minimum required buffer size.

$$R_j^k = C_i^k \rightarrow C_j^{k+\pi+\beta} = W_j^k \tag{5.3}$$

$$\alpha = \pi + \lceil \frac{\Delta_{send}^{+} + \Delta_{msg}^{+} + \Delta_{mem}^{+} - \Delta_{recv}^{-}}{T^{-}} \rceil \tag{5.4}$$

As α is the start address of the first write operation, it is assumed that the buffers first α elements are initially prefilled. Δ_{send}^{+} and Δ_{recv}^{-} give the upper and lower bound on the clock delays on the sender and receiver side, respectively. Δ_{msg}^{+} gives the upper bound of ticks it takes to transfer the message from the senders input buffer to the receivers communication buffer. Δ_{mem}^{+} is the upper bound of ticks the communication buffer requires to write an element. π is the clocking system's precision and T^{-} the minimum time between two succeeding ticks of a clock.

$$\beta = 2\pi + \lceil \frac{\Delta_{send}^{+} + \Delta_{msg}^{+} + \Delta_{mem}^{+} - \Delta_{recv}^{-}}{T^{-}} \rceil + \lceil \frac{\Delta_{recv}^{+} + \Delta_{rd} + -\Delta_{send}^{-} - \Delta_{msg}^{-}}{T^{-}} \rceil \tag{5.5}$$

In addition to the start address offset given by α, we have to guarantee that no overwrite takes place before the location has already been read. This additional margin is given by the maximum and minimum clock delay on the receiver and the sender side Δ_{recv}^{+} and Δ_{send}^{-}. Δ_{rd}^{+} is the upper bound needed to read a certain memory location in the communication buffer. Δ_{msg}^{-} gives the lower bound, which states how long it takes to transfer the message from the communication buffer to the receiver's output buffer.

5.7 Round Generation

All the algorithms evaluated in this thesis rely on the lockstep synchronous execution model. As described in Section 3.1, the execution takes place in fixed rounds. In each round an algorithm can send an arbitrary number of messages. By definition, each successfully transmitted message sent during round r from node p_i to node p_j is available at node p_j at the beginning of round $r+1$. Since our hardware framework only supports one message per clock cycle, we decided to restrict the number of messages sent by each node p_i to each node p_j ($j \in P$) to one. Therefore, we have a proportional mapping of the message complexity to the time complexity of our algorithms. Since larger messages have to be split up in several shorter ones, the algorithm's bit complexity directly influences its message complexity and further its time complexity. Therefore, we only use *resilience* and *time complexity* when comparing the algorithms.

As our algorithms require an underlying round structure, we implemented a component notifying the application logic when a new round starts and when a new message receipt occurs. The generation of these signals is based on the microtick signal of our clock generation. We can not rely on message reception here, as faulty messages would compromise the round generation scheme. Since the round generator has some restrictions on the underlying communication subsystem, the communication subsystem has to provide (i) a constant, a priori known message transmission time of t_{slot} micorticks (ii) a constant, a priori known transmission latency of $t_{latency}$ micorticks and (iii) a constant, a priori known startup time of t_{start} micorticks. Since our communication subsystem provides all of these criteria, a round generation and a message notification scheme as given in Algorithm 7 can be used for our implementation. [25, Pol09]

Our communication layer supports the sending of an 8 bit message at the begin of every message slot. By the common system wide reset provided by the clocking system, we can safely align the start of the first message slot directly after the communication systems startup (t_{start} microticks). All successive message slots are aligned back to back. Therefore, after startup the node has to wait for t_{start} microticks before it starts the first round. After this the algorithm infinitely repeats to signal the beginning of a new round every $t_{slot} + t_{sync}$ microticks.

$$t_{sync} = t_{slot} \lceil \frac{t_{latency} - t_{slot} + t_{calc} + 1}{t_{slot}} \rceil \tag{5.6}$$

Since the lockstep model requires that each successfully transmitted message sent during round r is delivered in round $r+1$ we have to wait for at least t_{slot} microticks for sending out all bits of the message. To provide a sufficient compensation for the message latency and the zero time computation assumed by the lockstep synchronous model, we wait additional t_{sync} microticks until the next round starts. As the message slots are aligned back to back t_{sync} (as given in Equation 5.6) has to be a multiple of t_{slot} microticks.

Algorithm 7 *Round Generation Algorithm*

```
   Initialization:
1: $r := \{0\}$ {round counter}
2: $nr$ {number of rounds algorithm needs}
3: $t_{start}$ {startup time of communication layer}
4: $t_{latency}$ {transmission latency of a message}
5: $t_{slot}$ {length of a message slot}
6: $t_{sync}$ {synchronization time}
7: $t_{calc}$ {calculation time of the algorithm}

   round generation:
8: wait for $t_{start}$ microticks
9: for ever do

10:    signal start of round $r$
11:    {compensate last rounds messages latency}
12:    wait for $t_{latency}$ microticks
13:    signal message reception

14:    {wait for the message arrival}
15:    wait for $t_{slot}$ microticks

16:    {realign execution to the next message slot}
17:    wait for $t_{sync} - (t_{latency} - t_{slot} + 1)$ microticks

18:    if $r < nr$ then
19:        $r = r + 1$
20:    else
21:        $r := 0$
```

5.8 Information Mapping

The message passing subsystem used (see Section 5.6) is designed to transmit messages of 8 bit size. To use the maximum bandwidth provided we decided to execute 8 instances of the protocols in parallel and map their information to one message. Since each of the protocols discussed in Chapter 3 is designed to use messages of 1 bit size, one possibility is to map the information to the message one to one. Another possibility is to use a single instance of the protocol which works with 8 bits of data. In this section we discuss the advantages and drawbacks of both versions and show usecases justifying the decision for the chosen mapping.

Due to physical variations, different methods of calibration or the adjustment of sensors the value can differ even in a fault free execution. Assume a set of values measured by a temperature sensor. In the fault free case, all temperature sensors will measure nearly the same values and therefore the results will only vary in the least significant bits.

In the case that no explicit decision can be reached, the decision finding process in the consensus protocols under evaluation always falls back to a previously chosen default value. When using a single, eight bit consensus protocol, even a deviation in the least significant bit may prohibit the calculation of a decision value and the default value may be used. When using eight single bit instances of the algorithm, however, the most significant bits will be the same and a correct decision value is found. Only for the least significant bits, no solution may be found and the default value will be used. Validity will not be violated in this case, as for the most significant bits the majority value is chosen and for the least significant bits both choices will be correct,as both values '0' and '1' were present in the input space.

Let's reconcile the agreement and validity properties outlined in Section 3.2 for both versions: For this example the values 0x0F, 0x0E, 0x0D and 0x0C have been chosen. In the case of a single consensus instance handling $n = 4$ and $f = 1$ operating on an 8 bit value domain the results of the EIG resolve function using an 8 bit default value $v = 0x00$ will decide to $0x00$, as the input values do not match. Since the EIG algorithm is only designed for exact input values, the algorithm will consider three nodes faulty which violates $f = 1$.

In the case of 8 consensus instances in parallel the resolve functions lead results in a range of values around the sensors values in most of the cases. For each input value the result is calculated bitwise. Therefore, the input values x_{p_0} = 0x0F (=0b00001111), x_{p_1} = 0x0E (=0b00001110), x_{p_2} = 0x0D (=0b00001101) and x_{p_3} = 0x0C (=0b00001100) lead to a resolve value of 0x0C (0b00001100) at label "node 0 said that . . . ", 0x0D (0b00001101) at label "node 1 said that . . . ", 0x0E (0b00001110) at label "node 2 said that . . . ", 0x0F (0b00001111) at label "node 3 said that . . . " and a decision value of y_{p_*} = 0x0C (=0b00001100) for the whole tree. Nevertheless, cases exist where the decision value does not meet the range around the original values. As an example, assume x_{p_0} = 0x0F (=0b00001111), x_{p_1} = 0x0E (=0b00001110), x_{p_2} = 0x11 (=0b00010001) and x_{p_3} = 0x10 (=0b00010000). Although the deviations of these values from each other are as far apart as in the example before, the result of the resolve function for y_{p_*} = 0x00 (=0b00000000). However, validity is satisfied, since at least one correct node has the input value 0b0 considering each bit separately. Since all nodes decide to the same value agreement is satisfied too.

An exhaustive elaboration of 8 bit values was made for both variants. All decisions differing from the nearest input value by 5 were categorized as invalid. When the maximum deviation of any two input values is restricted by 4 the analysis showed that 91.6% of all runs were considered correctly if each bit is treated separately, while only 15.7% were considered correctly when using eight bit consensus. A more in-depth analysis of this topic is out of the scope of this thesis and will therefore not be performed here.

CHAPTER 6

Implementation in Hardware

In the previous chapter we outlined an architectural overview of the hardware framework we use. This chapter presents the implementation of the *EIG* and *Phase King* protocol on top of this framework. The components outlined in this chapter are designed for protocols using $n = 4$ and $f = 1$. In contrast to the software simulations, the hardware components are designed to execute 8 consensus instances in parallel, therefore processing a byte of information. Each instance is limited to single bit messages and no information of the other ones are incorporated into its execution. As the hardware framework transmits a byte of information in parallel, the data of all 8 instances can be transmitted using a single invocation of the message layer. Therefore, bit x ($0 \leq x < 8$) of the message corresponds to the consensus instance x. To hide the details of the implementation from the application logic and to enable it to use it with both protocols, the implementations are encapsulated behind a common interface depicted in Figure 6.1.

Note: Components and signals used for fault injection and testing have been omitted.

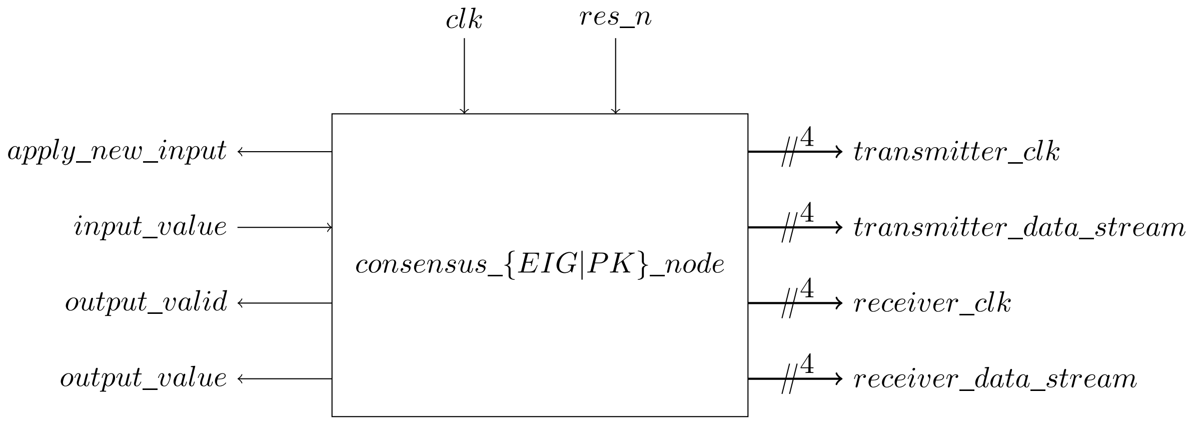


Figure 6.1: Interface between application logic and consensus implementation

clk and *res_n* are the clock and the reset signal, respectively, provided by the controller node. The *input_value* port is used to supply the consensus protocol with the value to reach consensus for. The result of the consensus calculation can be read from port *output_value*. *output_valid* indicates, if the output value can be read in the current clock cycle ('1') or if the calculation is still in progress ('0'). The signals *transmitter_clk*$(0 \ldots 3)$ and *transmitter_data_stream*$(0 \ldots 3)$ as well as *receiver_clk*$(0 \ldots 3)$ and *receiver_data_stream*$(0 \ldots 3)$ are the interfaces to all other nodes. Please note that each node has a interface to itself, to keep the protocol implementation as regular as possible. Details on the communication protocol used on these links has already be given in Section 5.6.

6.1 Implementation of the EIG Protocol

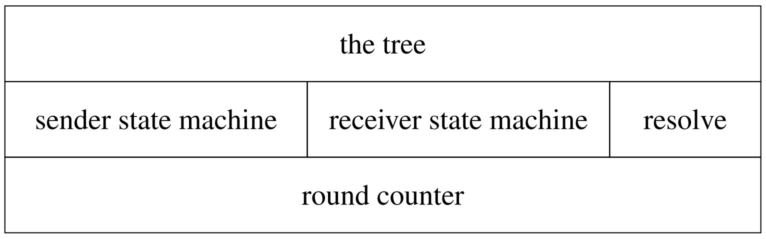


Figure 6.2: Architectural design of the EIG component

An architectural overview of the EIG component's design can be found in Figure 6.2. The tree forms the central element of this implementation. Each of the 20 records of this linear memory can store one message. All information gathered during the execution is saved in the tree. Since 8 instances of the EIG protocol are executed in parallel, each message contains the information of all the 8 instances. Therefore, the least significant bit of the message ($message[0]$) contains

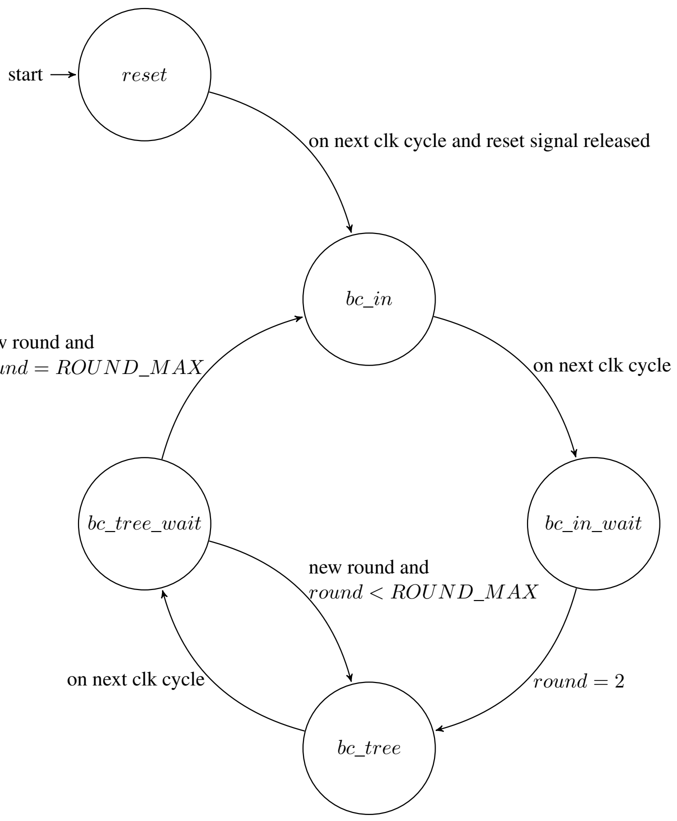


Figure 6.3: Sender state machine of EIG implementation

the data corresponding to instance 0, $message[1]$ the data corresponding to instance 1 and so forth. The round counter is configured to trigger 5 rounds, each consisting of 1 message. The round counter also coordinates the receiver state machine, the sender state machine and the resolve function.

Sender State Machine

The state machine responsible for broadcasting the messages to the other nodes is quite straight forward and can be found in Figure 6.3. It starts in the state $reset$. After the component's reset signal is released, the state transition to state bc_in is triggered at the next clock tick. In state bc_in the input value of the node is broadcasted to each of the 4 nodes of the system (including the node itself). Afterwards a transition to bc_in_wait is triggered. It is used to wait for the beginning of the next round. At the beginning of round 2 (as signaled by the round counter) the first level of the tree is broadcasted in state bc_tree. Similar to bc_in_wait, bc_tree_wait is a helper state for bc_tree to implement the waiting logic until the next round switch occurs. Dependent on the number of rounds already executed, the successor state of bc_tree_wait is different. If all tree levels have already been broadcasted (namely $round = ROUND\_MAX = 5$), the state

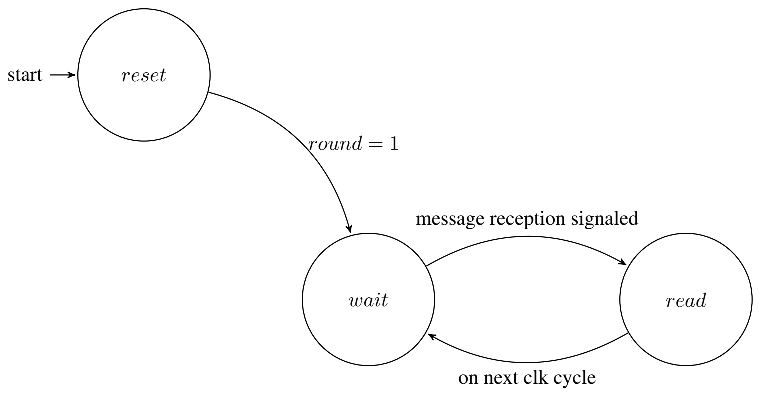


Figure 6.4: Receiver state machine of the EIG implementation

machine starts the next consensus iteration by returning to round 1 and reading the next input value. If, on the other hand, the broadcast of the tree has not yet been finished, the state machine returns to the state bc_tree.

Receiver State Machine

The purpose of the receiver state machine is simply to save the received messages at the right positions in the $tree$. As we have seen in Section 5.7, the latest possible reception time of each message is signaled by the round counter. Using this information it is simple to decide when to read the messages safely out of the receive buffers. This behavior is implemented by synchronizing to the round counter at startup (state $reset$) which is achieve by simply waiting for the start of round 1. After this synchronization, the state machine stays in the $wait$ state until the round counter indicates the reception of a new message triggering a state change to the $read$ state where the message is copied to the tree. One clock cycle later the state machine switches back to the $wait$ state.

$$tree[(round - 1) * 4 + i] = \text{received data from node } i; \tag{6.1}$$

The addressing of the data in the tree memory is done according to Equation 6.1. For more detailed information on the data storage within the tree please refer to Section 4.3).

6.2 Resolve Function of the EIG Implementation

The resolve function is triggered in round 1. Thus, the values used, namely $tree[4 \dots 19]$, are the values of the previous iteration of the consensus protocol. The hardware implementation of the resolve function is designed similarly to the one used for the simulator in Section 4.3. The major difference is, that the messages used in the hardware implementation consist of 8 bits, while the simulation environment only uses single bit messages.

The resolve function is shown in Figure 6.5. For each group of messages stored in the leaves of the tree, a bitwise resolve function is executed. The results of these intermediate functions (namely $resolve0$, $resolve1$, $resolve2$ and $resolve3$) are used as the input values for the next resolve stage. Please note that all information processed by a single node only ($tree[4]$, $tree[9]$, $tree[14]$ and $tree[19]$) do not contribute to the resolve function.

6.3 Design of Component Implementing the Phase King Protocol

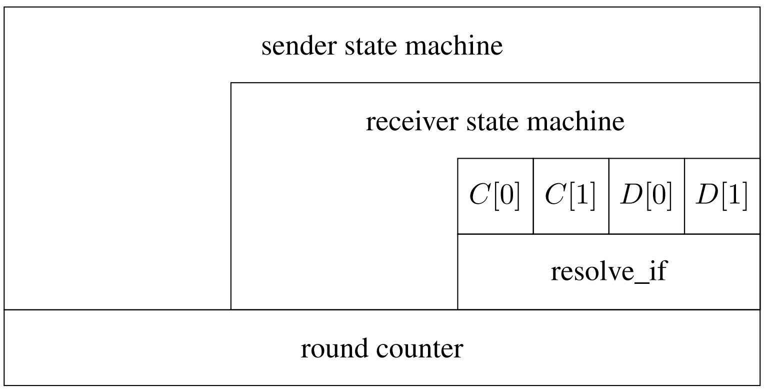


Figure 6.6: Architectural design of Phase King component

An overview of the Phase King component implementation is given in Figure 6.6. The protocol is executed in 8 rounds. In contrast to the EIG implementation, where the gathered information is stored in a linear memory which has to be post-processed by a dedicated resolve function after all information is gathered, the Phase King implementation executes its resolve functions multiple times. The Phase King resolve function calculates a weight value by counting the numbers of ones and zeros for each bit position in all messages received in the current round. Depending on the current round, the calculated resolve values give an indicator for the strength of a preferred value. In our implementation, the execution of the resolve function is triggered by the receiver state machine. The calculated results are stored in the registers $C[0]$, $C[1]$, $D[0]$ and $D[1]$. These registers are accessed later on by the sender state machine to retrieve these values. (A detailed description can be found in Section 3.7.)

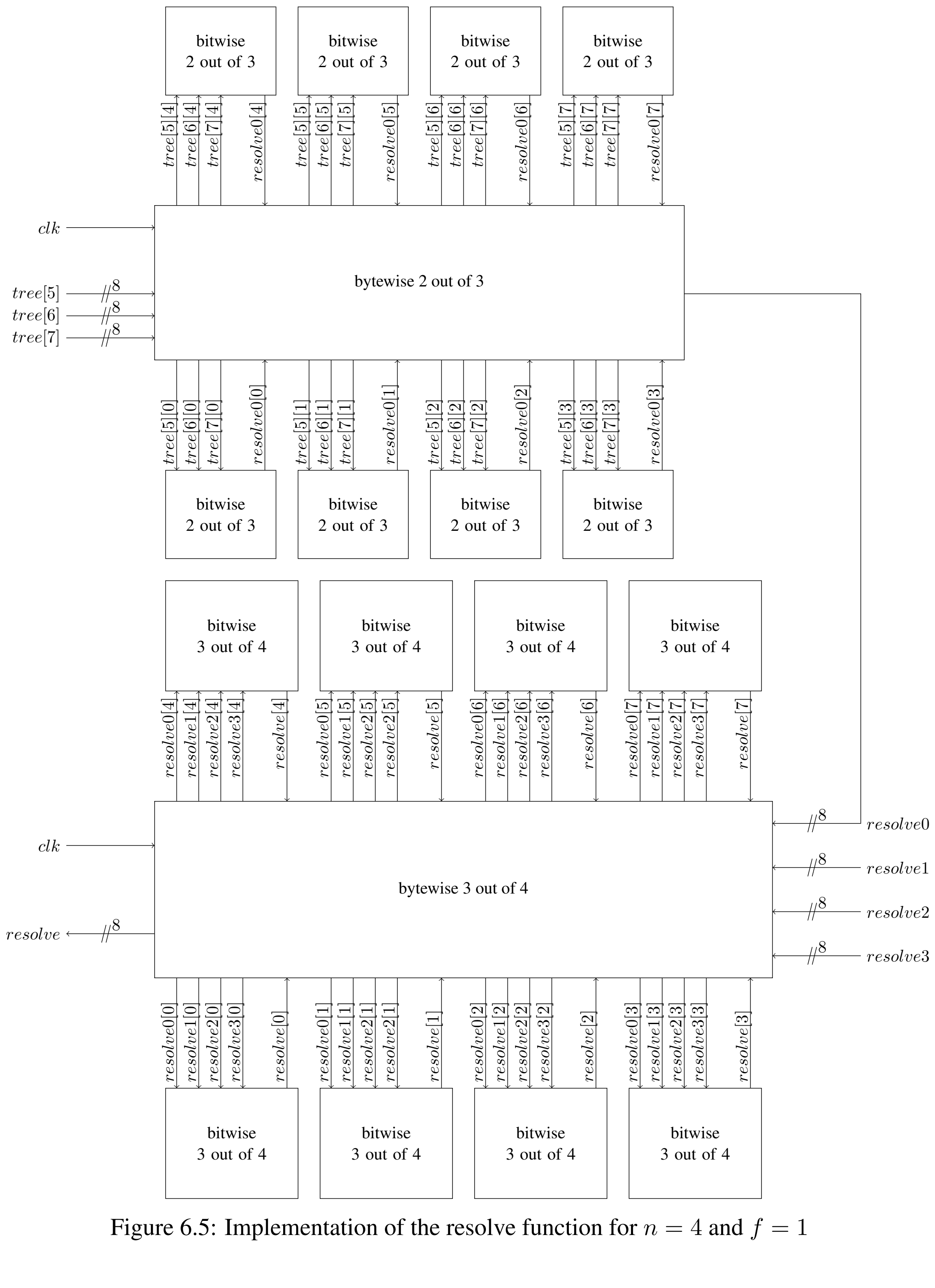


Figure 6.5: Implementation of the resolve function for $n = 4$ and $f = 1$

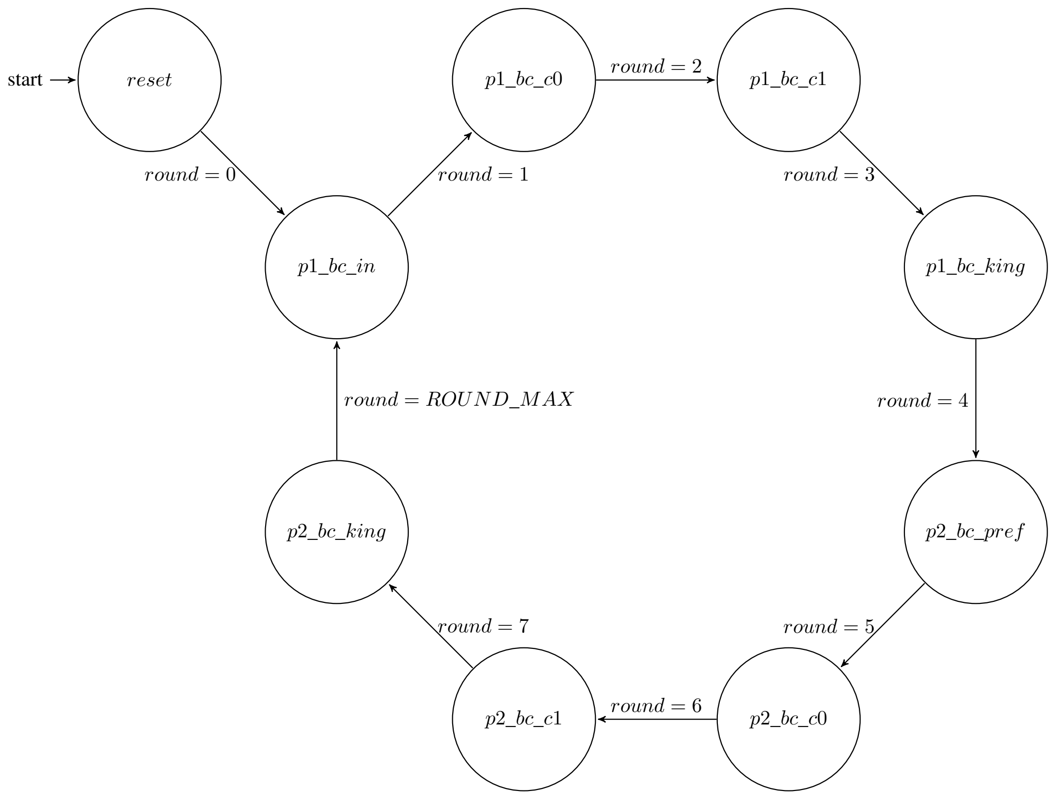


Figure 6.7: Simplified sender state machine of the Phase King implementation (wait states removed)

Sender State Machine

The sender state machine is shown in Figure 6.7. Please note that all wait states have been removed from the figure to increase its readability. The wait states are used to synchronize the message receptions with the round counter.

At system startup (after the clock has started and the reset signal has been released) the state machine synchronizes to the node's round counter by waiting for $round = 0$. Afterwards the input value is broadcasted to all nodes (including the node itself) and the local preference value is set to the local input value in state $p1\_bc\_in$. In round 1 (state $p1\_bc\_c0$), the state machine broadcasts $thresholdHigh(C[0])$ (according to Algorithm 8) and afterwards in round 2 $thresholdHigh(C[1])$ (state $p1\_bc\_c1$). The values of $C[0]$ and $C[1]$ have been calculated by the receiver state machine as depicted in Figure 6.6.

Afterwards, when reaching $round = 3$ the King's Phase is initiated. Therefore, the node with the index 0 broadcasts its current preference. This is called the King's broadcast of phase 1. The broadcast value is calculated based on $D[1]$ and according to $thresholdLow(D[1])$ as given in Algortihm 9. At the beginning of phase 2 ($round = 4$) each node updates its local preference value according to the received King's broadcast and the values reflecting D_0 and D_1 (calculated in phase 1) in state $p2\_bc\_pref$ and broadcasts its preference to all other nodes. The states $p2\_bc\_c0$ and $p2\_bc\_c1$ are similar to the states $p1\_bc\_c0$ and $p1\_bc\_c1$, respectively, but use updated values for $C[0]$ and $C[1]$. The King's broadcast of phase 2 is sent in state $p2\_bc\_king$ by the node with index 1. The calculation is similar to the calculation in state $p1\_bc\_king$, but uses the updated register $D[1]$ of phase 2. Finally in $round = ROUND\_MAX = 8$ the final decision is reached in state $p1\_bc\_in$ and the next input value is already broadcasted. (A more detailled discussion of the algorithm and the registers $C[0]$, $C[1]$, $D[0]$ and $D[1]$ can be found in Section 3.7.)

Algorithm 8 $thresholdHigh(value)$ *used in sender state machine of Phase King*

```
1: for value(i), $0 \le i \le 8-1$: do

2:     if $value(i) \ge N - F$ then
3:         $output(i) = 1$
4:     else
5:         $output(i) = 0$
```

Algorithm 9 $thresholdLow(value)$ *used in sender state machine of Phase King*

```
1: for value(i), $0 \le i \le 8-1$: do

2:     if $value(i) > N - F$ then
3:         $output(i) = 1$
4:     else
5:         $output(i) = 0$
```

Receiver State Machine

Like the sender state machine, the receiver state machine depicted in Figure 6.8 is reduced to the most important states. Thus, states with the purpose of simply waiting for the next message reception are neglected here. Therefore, a transition is triggered whenever the round counter component signals the safe access to messages previously received.

The receiver state machine starts in the state $reset$. A synchronization to the round counting component is done by waiting for the first message receipt in round 1. After this, the state machine triggers a transition to the state $p1\_read\_in$ where the input value of the other nodes can be safely read from the receiver components. In this stage the values for $C[0]$ and $C[1]$ are calculated by the resolve functions using the input values of all nodes (including the node itself). Since in each message 8 values for 8 instances of the Phase King protocol are packed, the weighted indicators have to be calculated for each bit of the messages separately. $C[0]$ holds

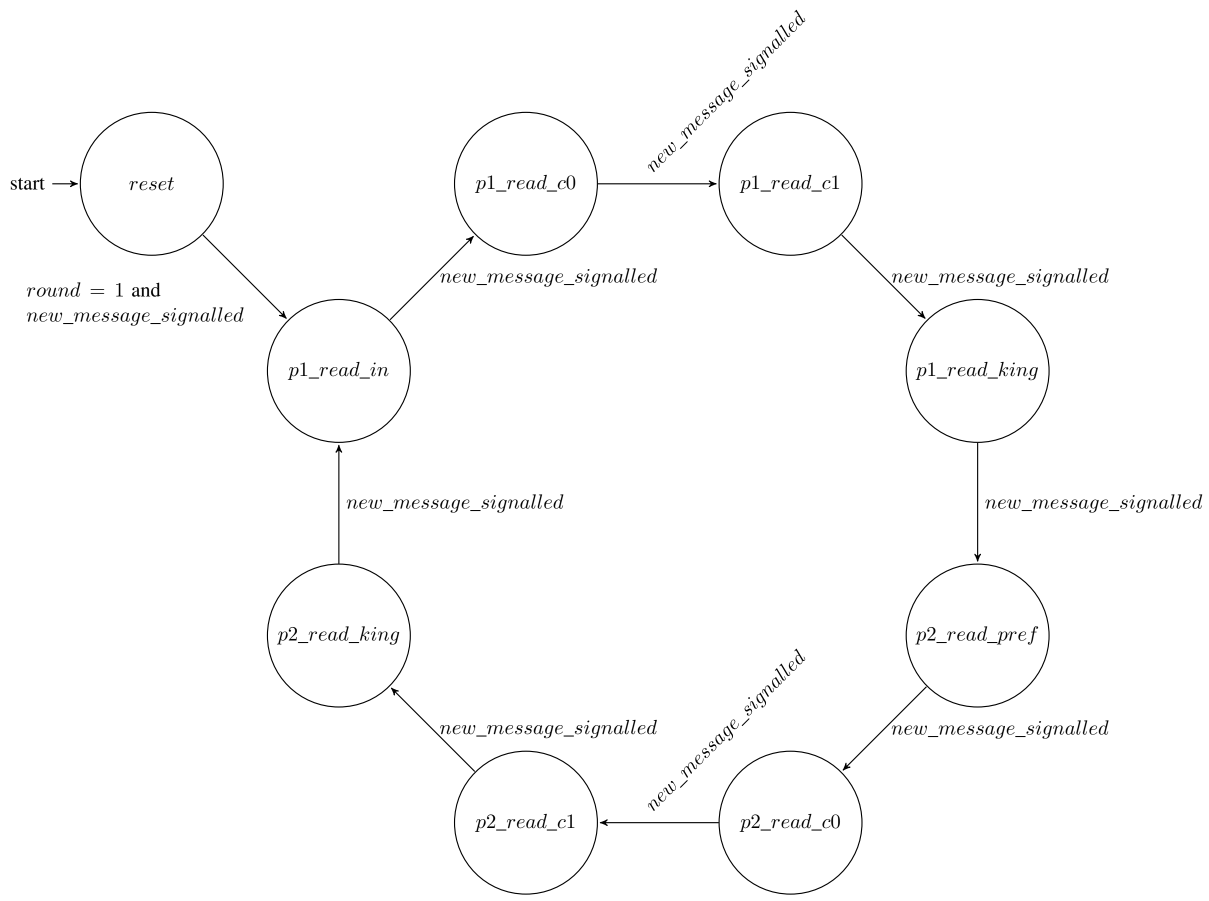
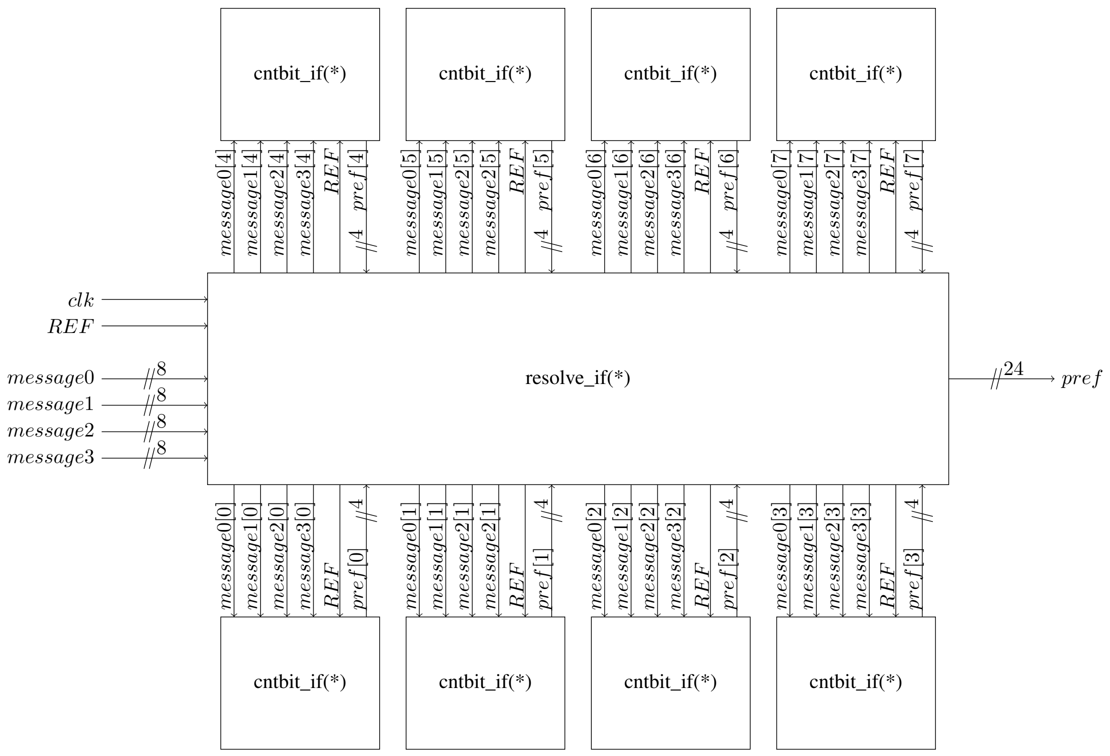

start
$reset$
$round = 1$ and $new\_message\_signalled$
$p1\_read\_in$
$new\_message\_signalled$
$p1\_read\_c0$
$new\_message\_signalled$
$p1\_read\_c1$
$new\_message\_signalled$
$p1\_read\_king$
$new\_message\_signalled$
$p2\_read\_pref$
$new\_message\_signalled$
$p2\_read\_c0$
$new\_message\_signalled$
$p2\_read\_c1$
$new\_message\_signalled$
$p2\_read\_king$
$new\_message\_signalled$

Figure 6.8: Simplified receiver state machine of the Phase King implementation (wait states removed)

the number of counted '0' per bit over all messages and $C[1]$ holds the number of counted '1' per bit over all messages. The value reflecting $C[0]$ is received in the state $p1\_read\_c0$. Upon these values, $D[0]$ is calculated by the resolve function, weighting them by counting the bits set to '1' in the received messages. Upon the received values $C[1]$, the same procedure is applied to calculate the value $D[1]$ (in state $p1\_read\_c1$). Finally, the current phase of the protocol is completed in state $p1\_read\_king$. In this state the King's broadcast is received. In contrast to the other messages, the kings message is received from a single node only (namely from the king of the phase). Thus, in phase 1 the King's broadcast is only received from node 0. Values in other receiver buffers are ignored. On the reception of the next message the state machine triggers a transition to the state $p2\_read\_pref$ starting the second phase of the protocol. The behavior in this second phase is similar to the first one, except that the King's message is received from node 1.



cntbit_if(*)
cntbit_if(*)
cntbit_if(*)
cntbit_if(*)
$message0[4]$ $message1[4]$ $message2[4]$ $message3[4]$ REF $pref[4]$ 4
$message0[5]$ $message1[5]$ $message2[5]$ $message2[5]$ REF $pref[5]$ 4
$message0[6]$ $message1[6]$ $message2[6]$ $message3[6]$ REF $pref[6]$ 4
$message0[7]$ $message1[7]$ $message2[7]$ $message3[7]$ REF $pref[7]$ 4
clk
REF
$message0$ 8
$message1$ 8
$message2$ 8
$message3$ 8
resolve_if(*)
24 $pref$
$message0[0]$ $message1[0]$ $message2[0]$ $message3[0]$ REF $pref[0]$ 4
$message0[1]$ $message1[1]$ $message2[1]$ $message2[1]$ REF $pref[1]$ 4
$message0[2]$ $message1[2]$ $message2[2]$ $message3[2]$ REF $pref[2]$ 4
$message0[3]$ $message1[3]$ $message2[3]$ $message3[3]$ REF $pref[3]$ 4
cntbit_if(*)
cntbit_if(*)
cntbit_if(*)
cntbit_if(*)

Figure 6.9: Implementation of the $resolve\_if(*)$ component for $n = 4$ and $f = 1$

6.4 Resolve Functions for Phase King

The resolve functions for Phase King used in this implementation are based on two weighting types. Both calculate an indicator $pref$ in the ranging from 0 to 4 for each bit of the preference value. The first weighting type counts the number of received ones, while the second weighting type counts the number of zeros, abbreviated with $resolve\_if(1)$ and $resolve\_if(0)$. The result of this weighting components is stored in registers holding 8 of these $pref$ values (one for each instance of the Phase King protocol). This preferences are calculated bitwise over all four messages received. Assume we received the values 0x01, 0x03, 0x04, 0xFF from the nodes in a round. The preference values would calculate to $1, 1, 1, 1, 1, 2, 2, 3$ for applying $resolve\_if(1)$ and $3, 3, 3, 3, 3, 2, 2, 1$ for $resolve\_if(0)$.

As presented in Figure 6.9, the whole eight bit values are passed to the component and split into groups by combinational units which are responsible for calculating the weights for each group. The units can be parameterized with a reference value, determining the weighting criteria, namely if ones or zeros have to be counted. As we have discussed previously, these indicators are stored in the registers $C[0]$, $C[1]$, $D[0]$ and $D[1]$. Based on these registers, the broadcast messages are calculated according to Algorithms 8 and 9.

CHAPTER 7

Evaluation of Hardware Implementation

In the first part (Section 7.1-7.7) we illustrate how the implemented algorithms work by discussing different execution scenarios. They highlight how different input values and faults influence the execution of the algorithm. These tests were executed using ModelSim on a PC. In the second part (Section 7.9) measurements executed on a real FPGA prototype are presented. The measurements were performed using a specialized test framework implemented on top of the consensus nodes. Each node was equipped with a dedicated tester, generating input values and comparing the results to pre-calculated reference values. To make the results comparable to the ones gathered from the simulations in software (presented in Secton 4.4), we use $f = 1$ and $t = 0$, $t = 1$ and $t = 2$.

7.1 Evaluation of the Implementations

As outlined in Section 5.5, the behaviour of the nodes can be configured via a software tool. This configuration includes (i) if the node is faulty and (ii) if it is faulty which kind of faults are applied. The structure of the fault configuration registered for one round of a single, faulty node is shown in Figure 7.1. This registers are split into two groups. The first one configures if a node is faulty and which of its links exhibits what kind of faults. To emulate byzantine behavior, the second part of the fault configuration, enables us to specify which messages to send to each of the neighbors.

The configuration of the link faults is split into groups of eight bits. Each group is used for the configuration of one neighbor. As shown in Figure 7.2, various link faults can be emulated. Possible configurations are:

1. Bit 1 set: Simulates a floating output ('Z') on the $data$ link.

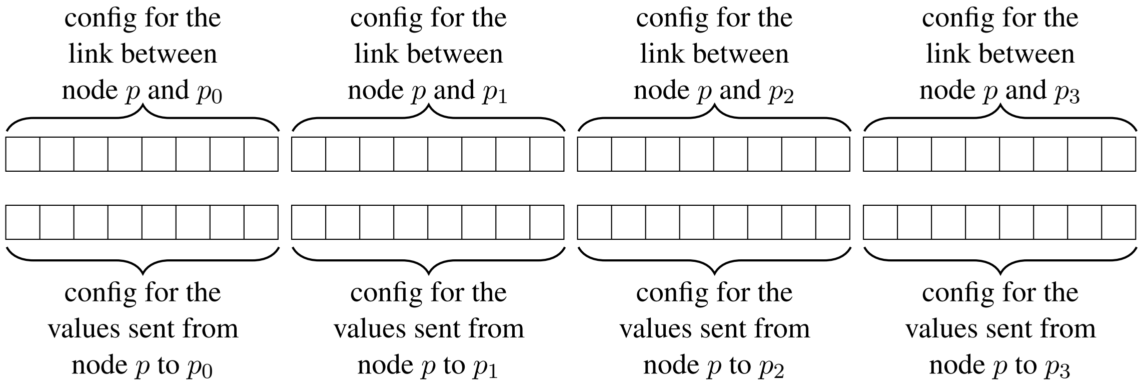

config for the link between node p and p_0
config for the link between node p and p_1
config for the link between node p and p_2
config for the link between node p and p_3
config for the values sent from node p to p_0
config for the values sent from node p to p_1
config for the values sent from node p to p_2
config for the values sent from node p to p_3

Figure 7.1: Fault configuration record for one round executed by a faulty node p

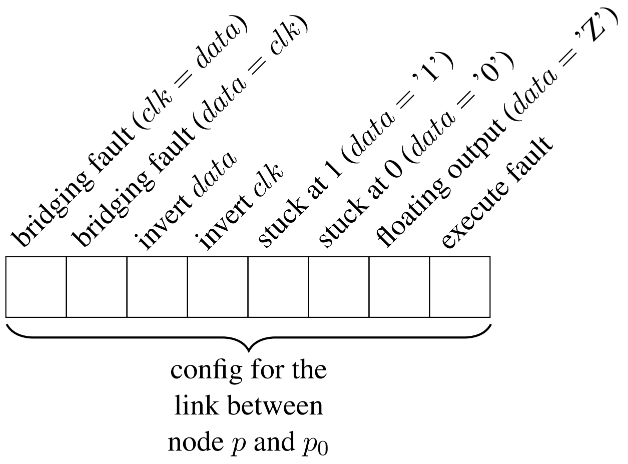

bridging fault ($clk = data$)
bridging fault ($data = clk$)
invert $data$
invert clk
stuck at 1 ($data =$ '1')
stuck at 0 ($data =$ '0')
floating output ($data =$ 'Z')
execute fault
config for the link between node p and p_0

Figure 7.2: Fault configuration of a link fault

2. Bit 2 set: Simulates a stuck at 0 fault on the $data$ link.
3. Bit 3 set: Simulates a stuck at 1 fault on the $data$ link.
4. Bit 4 set: Inverts the information on the $data$ link.
5. Bit 5 set: Inverts the clock signal on the clk link.
6. Bit 6 set: Simulates a bridging fault by duplicating the clk signal on the $data$ link.
7. Bit 7 set: Simulates a bridging fault by duplicating the $data$ signal on the clk link.

Bit 0 has a special meaning. It is used as a master flag to enable or disable the fault simulation on the link.

The evaluations presented in the following sections include three different scenarios for each implemented algorithm:

1. Case 1: All nodes behave correctly and process the values according to their implementation.
2. Case 2: Node p_3 behaves faulty and sets the communication links (data rail and clock rail) between node p_3 and node p_2 to several, invalid states.
3. Case 3: Node p_3 behaves faulty and broadcasts spurious messages. The major difference between this case and the case described before is that the messages broadcasted here have a valid structure and thus the fault can not be detected by the communication subsystem.

7.2 EIG Case 1 - Fault Free Execution

In this scenario the values $x_0 = 1$, $x_1 = 2$, $x_2 = 3$ and $x_3 = 4$ are used as input values and all nodes behave according to their specification. To give a better insight on the algorithm, we display the round number, the input value, the received data, the information tree and the calculated output value of node p_2 in the simulation. To create a correct reference for the faults injected throughout case 2 and 3, we also depicted the outgoing $data$ and clk rails between node p_3 and p_2 as well as the configuraton of the faults injected into node p_3.

Node p_2's execution of this scenario is shown in Figure 7.3. In round 1 new input values are processed by the algorithm (1). Afterwards, each node broadcasts its input value to all other nodes (including itself). These values are received during round 1 (2) and are available for the receiver at the begin of round 2. In round 2 they are added to the information tree (3). Thus, the first four entries of the tree are filled with the input values of the nodes. Next, each node broadcasts the informations stored in $tree[0..3]$. Therefore, in each round $r > 1$, the nodes distribute their local view of node p_{r-1}'s proposal. Since no fault occurs in this scenario, the tree correctly fills with the values of the nodes. For example: (4) shows the values received during round 2, which are the values send by node 1 as witnessed by the other nodes in round 1. The correct final decision of 0 is made at the begin of round 6 (5).

7.3 EIG Case 2 - Communication Link Failures

In the second scenario node p_3 is configured to be faulty. In this scenario the link between node p_3 and node p_2 is corrupted.

The input values of all nodes are set to 1 in this execution. As we can see in Figure 7.4, the data link is configured to exhibit an open fault (high Z) in the first round (1), a stuck at 0 fault in the second round and a stuck at 1 fault in the third round. In the fourth round the information sent via the data rail is inverted and in the fifth round the clock signal on the clk rail is inverted. In round six and seven bridging faults are applied by replicating the clk signal on the data rail and the $data$ signal on the clk rail. (2) shows the tree filled with the input values of the nodes. Since the communication link between node p_3 and node p_2 is corrupted, the value witnessed from node p_3 is zero because the communication subsystem has detected the erroneous transmission and discarded the value. The reception of $0x00$ from p_3 at p_2 can be witnessed throughout the whole information gathering phase. Since only the link between these two nodes is corrupted, the other nodes (p_0 and p_1) received correct values from p_3, as we can see in (3). Here, p_0 and p_1 report that node 3 has stated its initial value to be $0x01$, while all information received from p_3 is $0x00$. Nevertheless, the information gathered by the algorithm results in $0x01$, which is correct.

7.4 EIG Case 3 - Spurious Messages

In the third scenario (case 3), again, the inputs are set to 1 for all nodes. In this case, however, the faulty behaviour of node p_3 manifests itself by broadcasting arbitrary but valid messages to the nodes. In case 2 the link faults could be detected by the communication subsystem and $0x00$ was delivered for the faulty links. In this scenario the faulty node's spurious messages reach the consensus algorithm.

As depicted in Figure 7.5, permutations of $0x01$, $0x02$, $0x03$ and $0x04$ are broadcasted by node p_3 to the other nodes (1). In round 1 p_3 sends $0x01$ to node p_0, $0x02$ to node p_1, $0x03$ to node p_2 (2) and $0x04$ to itself. In the follwing round (round 2) we can see the manifestation of the faulty messages in the information tree (3). Similar corruptions can be observed in round 3 (4) and round 4 (5). Since the majority of all groups of information in the tree is $0x01$, the decision value resolved in round 6 is still $0x01$, which is the correct result for this execution.

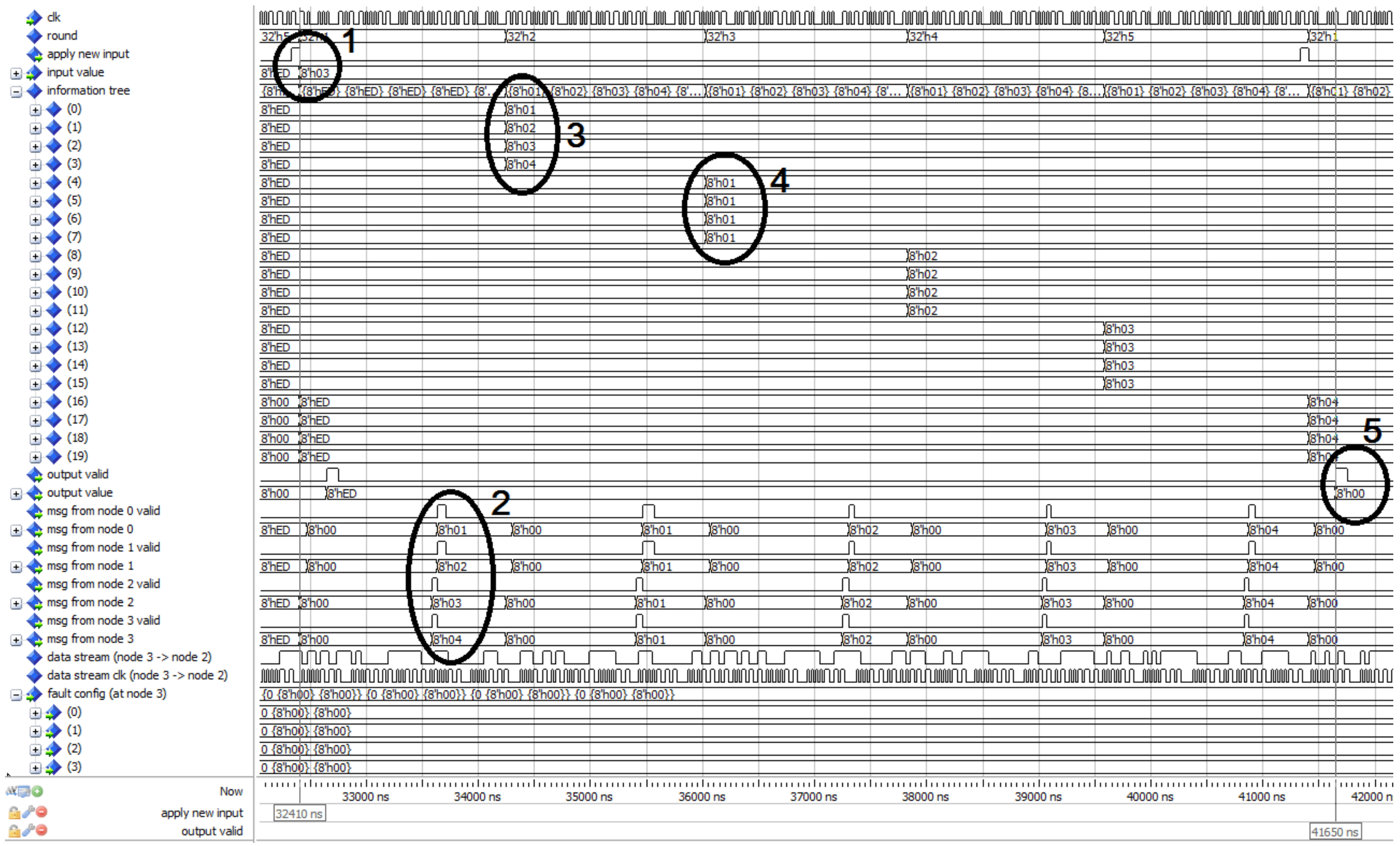


Figure 7.3: EIG Case 1: Fault free execution of p_2 with the input values $x_0 = 1$, $x_1 = 2$, $x_2 = 3$ and $x_3 = 4$

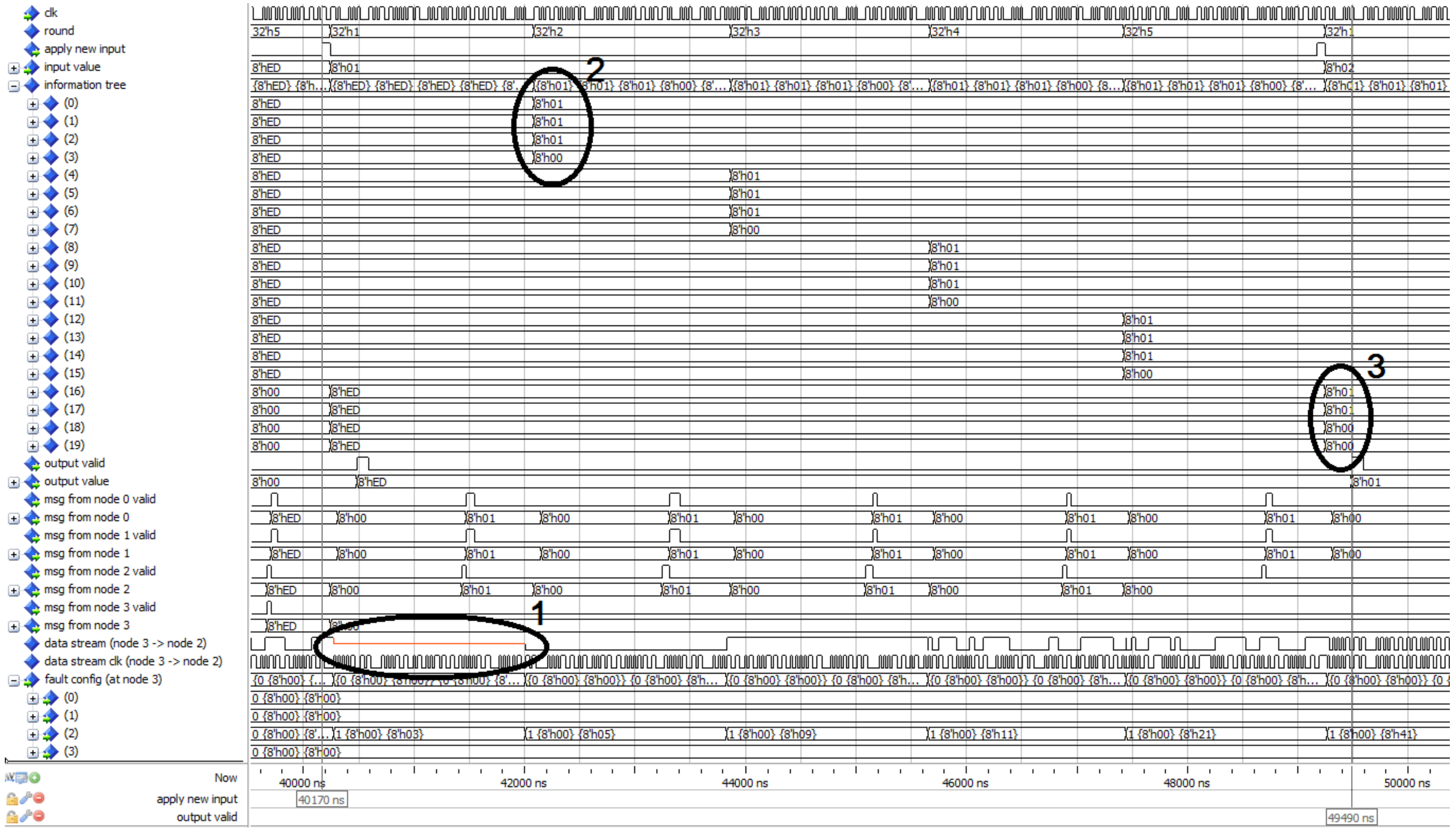


Figure 7.4: EIG Case 2: p_3 generates faulty signals on the communication links to node p_2 (input values $x_0 = x_1 = x_2 = x_3 = 0x01$)

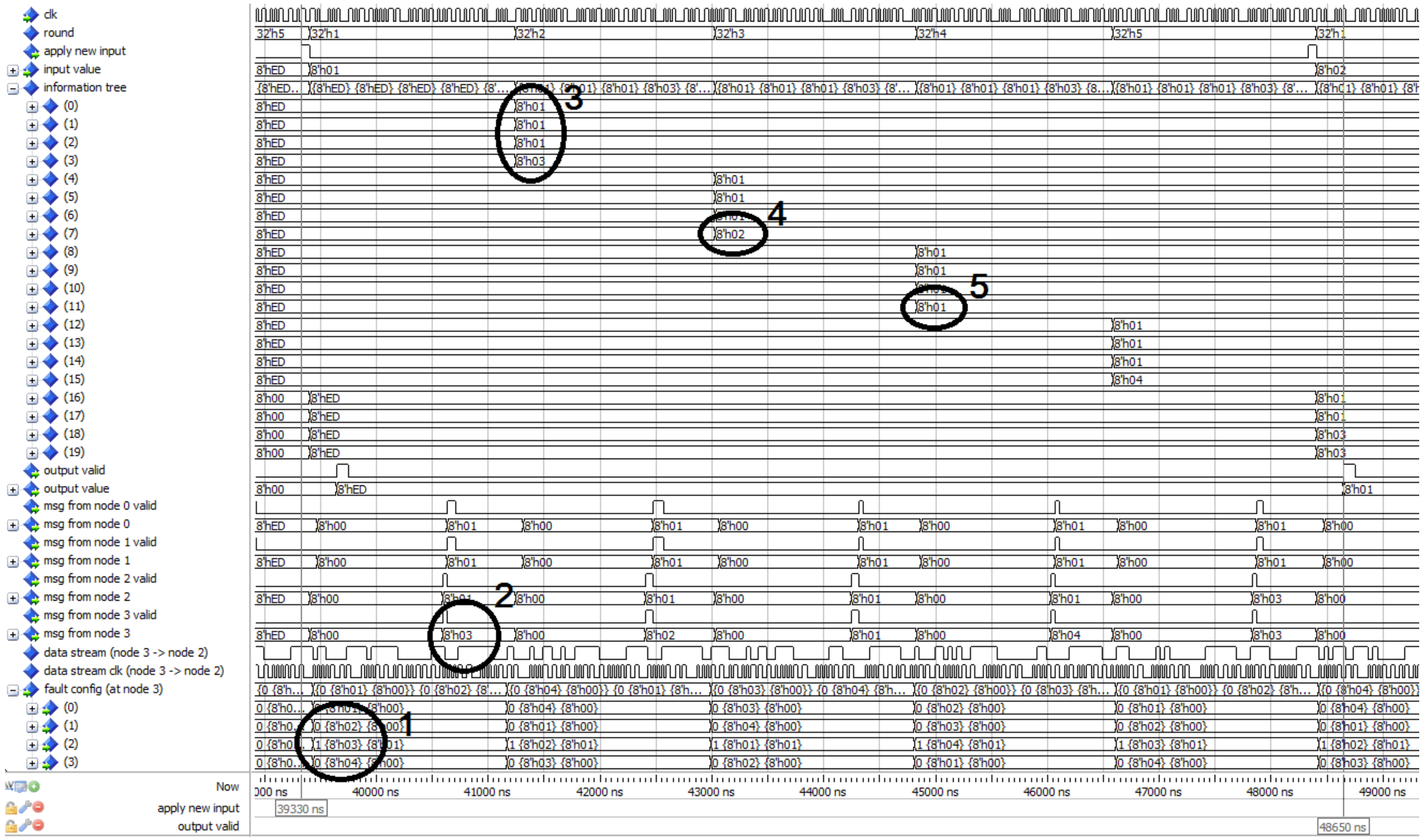


Figure 7.5: EIG Case 3: p_3 broadcasts arbitrary but valid messages to all the other nodes (input values $x_0 = x_1 = x_2 = x_3 = 0x01$)

7.5 Phase King Case 1 - Fault Free Execution

As shown in Figure 7.6, the input values are set to $0x01$ (1). Since in this case all nodes behave correctly, the execution shown for node p_2 is similar to the executions observable on all other nodes. Thus, every node sets its local preference value to the input value and broadcasts it to all other nodes in round 1.

Since all nodes received $0x01$, at the begin of round 2 (2), $C[0]$ and $C[1]$ (calculating the number of received '0's and '1's for every single bit evaluate to $0, 4, 4, 4, 4, 4, 4, 4$ and $4, 0, 0, 0, 0, 0, 0, 0$, respectively (3). These values are transformed to an 8 bit value according to the given threshold (see Section 3.7) and broadcasted to all other nodes (4) and (6). The transformed value of $C[0]$ is received at the start of round 3 where $D[0]$ is calculated (5) according to the received number of '1's (for each bit of the messages). The same calculation is executed for $D[1]$ (7) with the received values of $C[1]$ in round 4. In our sample $D[0]$ and $D[1]$ evaluate to $0, 4, 4, 4, 4, 4, 4, 4$ and $4, 0, 0, 0, 0, 0, 0, 0$, respectively. According to the values of $D[1]$, the kings broadcast is sent by node p_0 in round 4 (8). In round 5 the nodes update their local preference values depending on the trust in their current preference (manifested by $D[0]$ and $D[1]$) and the message received from the king of the current phase (p_0). The updated preference value is then sent to all nodes similarly to the input value in round 1. This described procedure is repeated for all later rounds. At the beginning of round 9, the output value of the protocols implementation is set to the calculated preference value. In this case the output value is evaluated to $0x01$, which is correct (9).

7.6 Phase King Case 2 - Communication Link Faults

In the second scenario, the communication link between node p_3 and p_2 is faulty. Like in Section 7.3, the *data stream* and the *data stream clk* are modified. In the first round a floating output of the data stream is simulated at node p_3. In the second and third round a stuck at 0 and a stuck at 1 fault are applied, respectively. In round 4 and 5 the data stream and the data stream clk are inverted, respectively. In round 6 and 7 a bridging fault is simulated by duplicating the signal of the data stream on the data stream clk and vice versa. This configuration is repeated from round 8 onwards.

As we can see in Figure 7.7, the input value is set to $0x01$ (1). As the link faults could be detected by the communication subsystem and therefore are replaced by $0x00$, the faults manifest themselves as $0x00$ in the algorithm (2). This directly influences the result of $C[0]$ and $C[1]$ (3). Due to the faults, $C[0]$ and $C[1]$ evaluate to $1, 4, 4, 4, 4, 4, 4, 4$ and $3, 0, 0, 0, 0, 0, 0, 0$, respectively. Their values are received in round 3 (4) and round 4 (6). A similar observation can be made for $D[0]$ (5) and $D[1]$ (7). The King's broadcast can be seen in (8). Since all other nodes behave correctly, the decision value is still evaluated to $0x01$ (9), which is correct.

7.7 Phase King Case 3 - Spurious Messages

All inputs are set to $0x01$. Like in Section 7.4, node p_3 sends spurious messages. In contrast to case 2, the messages apper to be valid for the communication subsystem and therefore are delivered to the nodes. Again, permutations of $0x01$, $0x02$, $0x03$ and $0x04$ are broadcasted by node p_3.

As shown in Figure 7.5, in round 1 node p_3 sends $0x01$, $0x02$, $0x03$ and $0x04$ to node p_0, p_1, p_2 and p_3, respectively (1). Therefore, node p_2 receives $0x03$ during round 1 (2). This time the manifestation of the fault can be identified in round 2 when $C[0]$ and $C[1]$ are calculated (3). For the first bit no deviation with respect to a correct execution can be identified. Since the spurious value received was $0x03$, the first bit is set to '1' as in $0x01$. Therefore, the influences of node 3 cannot be observed in the results for the first bit of the messages. Nevertheless, influences of the spurious message can be identified in the results for the second bit. Similar behaviour can be observed in the results of $D[0]$ and $D[1]$. Since node 3 never broadcasts the kings message, the spurious message in round 4 (4) and round 8 (5) don't have any influences to the protocols execution. By the thresholds used for the decision finding, node p_2 decides correctly to $0x01$ in this case (6).

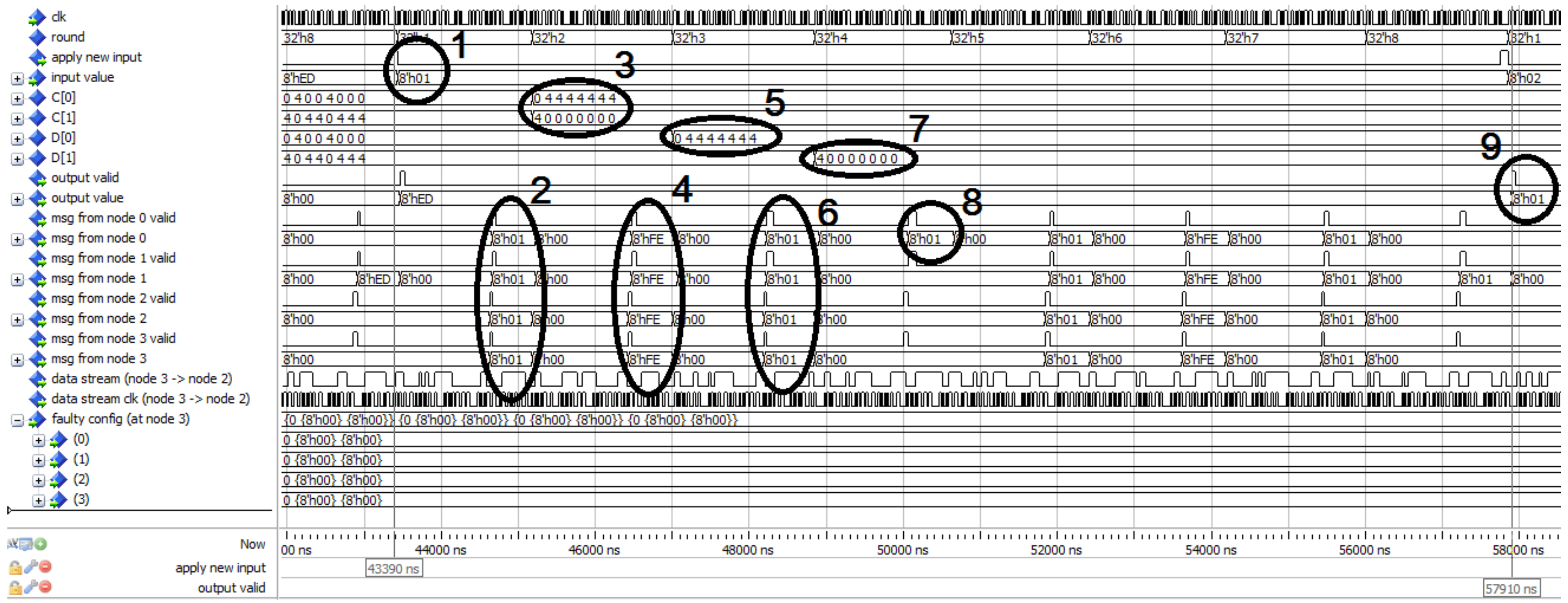


Figure 7.6: PK Case 1: Fault free execution of p_2 with the input values $x_0 = x_1 = x_2 = x_3 = 0x01$

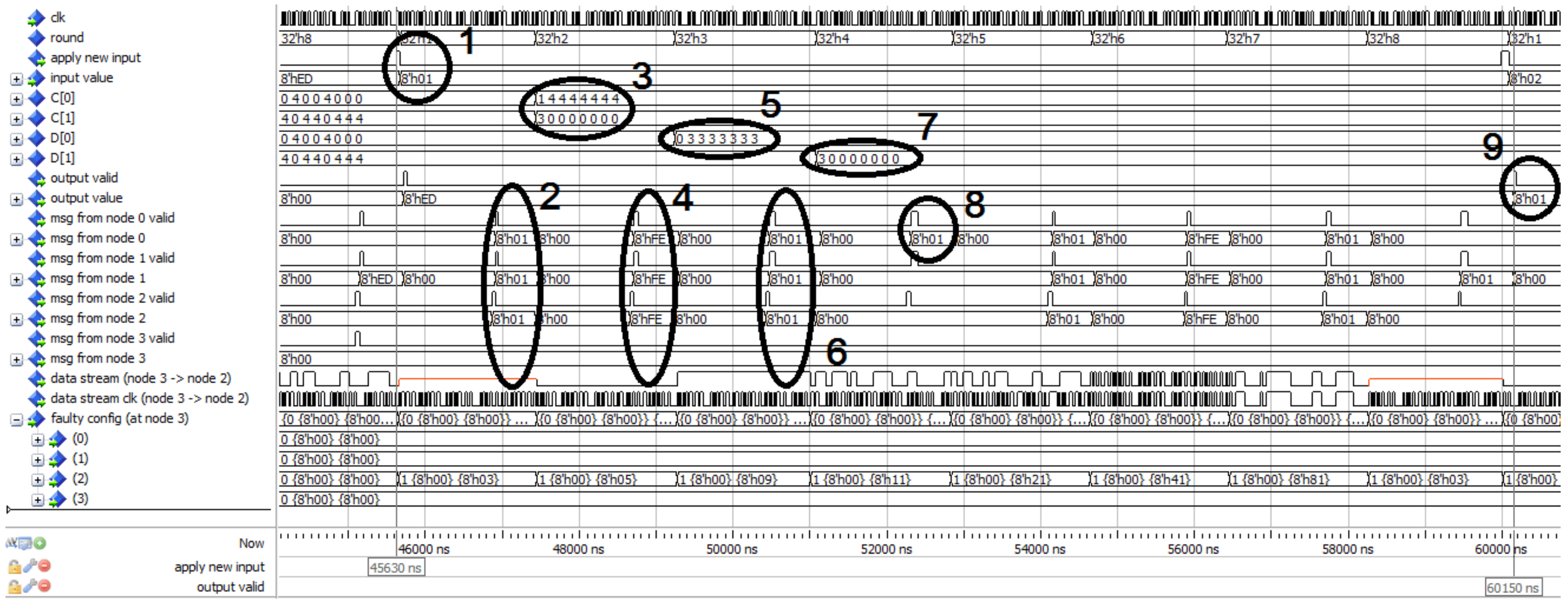


Figure 7.7: PK Case 2: p_3 generates faulty signals on the communication links to p_2 with the input values $x_0 = x_1 = x_2 = x_3 = 0x01$

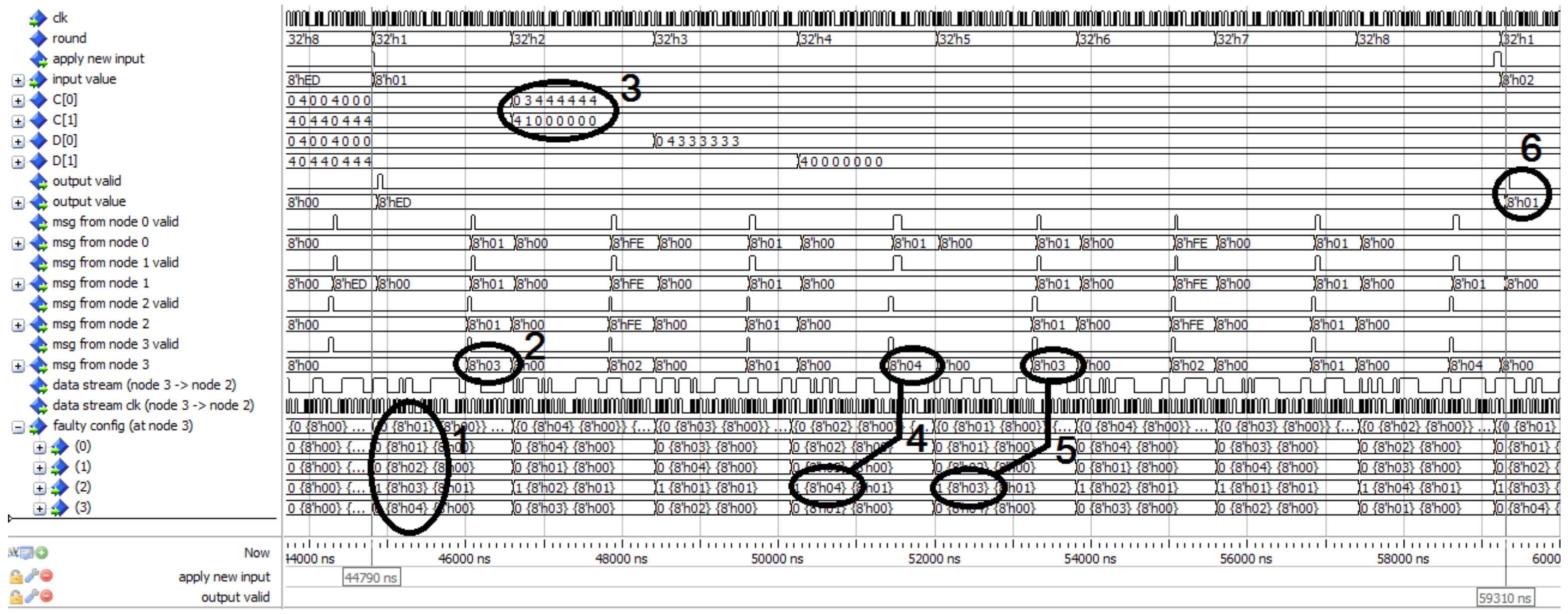


Figure 7.8: PK Case 3: p_3 broadcasts arbitrary but valid messages to all the other nodes using $x_0 = x_1 = x_2 = x_3 = 0x01$ as input

7.8 Hardware Measurements

The evaluation presented in this section has been performed on a Xilinx Virtex 4 FPGA. A setup as presented in Section 5.5 has been applied. The testcases applied have been computed offline on the PC. Via a software tool each testcase has been downloaded to the FPGA. After 200 ms the status of the protocols has been uploaded to the PC and evaluated. To achieve adequate results, we performed tests in more than 2 10^6 configurations for both implementations using $t = 0$, $t = 1$ and $t = 2$.

As shown in Table 7.1, the implementation of the Exponential Information Gathering protocol requires 5 rounds of communication and has a throughput of 8 bit of information per 150 microticks. The latency of a single execution (the time until the results for a given input value are available) is 154 microticks. The protocol's implementation provides a correctness of 100% for $t = 0$ and $t = 1$ for a given $f = 1$, as expected. If two faults are applied ($t = 2 > f$) the protocol's results are correct in 75% of the evaluated cases. To support comparability, the tests have been generated similar to the ones given in Section 4.4.

Protocol	n	f	t	runs correct($\%$)	runs faulty($\%$)	throughput	latency
EIG	4	1	0	100	0	150 ticks/8 bit	154 ticks
EIG	4	1	1	100	0	150 ticks/8 bit	154 ticks
EIG	4	1	2	75	25	150 ticks/8 bit	154 ticks
PK	4	1	0	100	0	240 ticks/8 bit	241 ticks
PK	4	1	1	100	0	240 ticks/8 bit	241 ticks
PK	4	1	2	80	20	240 ticks/8 bit	241 ticks

Table 7.1: Results of hardware evaluation

The implementation of Phase King requires 8 rounds of communication and, thus, provides a throughput of 8 bit of information per 240 microticks and a latency of 241 microticks. Like EIG, it provides a correctness of 100%, if the number of faults applied are at most the number of faults specified ($t \leq f$). If two faults are applied ($t = 2 > f$) the implementation solves the consensus problem correctly in 80% of the evaluated cases. Therefore, the tests evaluated reflect the benchmarks known from the previous simulations in software.

In summary, both implementations have been tested exhaustively. The results of the implemented protocols match the results gathered from the software simulation. Both provide a correctness of 100%, if $n \geq 3f + 1$ and $t \leq f$ holds. While the EIG's throughput is much better than the one Phase King provides, this is not the case for higher numbers of tolerable faults, which can be seen as a direct consequence of the exponential increase of message (or in our case round) complexity of the EIG algorithm.

7.9 Device Utilization

Besides the robustness of the implementations, we also evaluated the device utilization the protocols require. Therefore, only the communication subsystem and a single node of the protocols implementation are considered here. Further, the additionally installed hardware for the tests presented in the previous section has been omitted here.

	EIG	PK
Number of Slice Flip Flops	600 $(< 1\%)$	595 $(< 1\%)$
Total Number of 4 input LUTs	2670 $(< 5\%)$	2867 $(< 5\%)$
Number of occupied Slices	1541 $(< 4\%)$	1624 $(< 6\%)$
Max Frequency	110.858 MHz	114.378 MHz

Table 7.2: Device utilization of a Xilinx Virtex 4 (xc4vlx60)

The implementation of the EIG protocol on an Xilinx Virtex 4 FPGA utilizes less than 1% of the FPGA's flip flops and less than 5% of the FPGA's lookup tables (see Table 7.2). The timing analysis of the EIG implementation results in a maximum frequency of 110 MHz for the design. The Phase King protocol utilizes less than 1% of the FPGA's flip flops and less than 5% of the lookup tables. The timing analysis lead to a maximum frequency of 114 MHz, which is slightly faster than the one of the EIG implementation. Compared to the maximum frequency (128 MHz) of the used design for the receiver component of the communication buffer the speed of the consensus implementations is still good.

Concluding the evaluations taken, in the case of $f = 1$ both implementations perform well. Both grant acceptable utilization of the FPGA. The implementation of the Phase King protocol supports a higher speed of the circuit than the Exponential Information Gathering protocol, but, compared to the number of rounds (and as a direct consequence the number of microticks) required for a single execution, EIG performs better in this setup ($f = 1$). Nevertheless, given a higher number of f, the Phase King implementation is the protocol to be prefered.

CHAPTER 8

Conclusion and Future Work

In this thesis we have shown that the implementation of consensus algorithms are a powerful alternative to using TMR systems with replica determinism. A thorough analysis of three well known consensus algorithms, namely of the Exponential Information Gathering, the Phase Queen and the Phase King algorithm, was performed and the algorithms have been adopted to work in the confines of VLSI circuits. The restrictions on message size imposed by the VLSI circuits make it necessary to take the message complexity into account when analyzing the timing behavior of the algorithms. The concentration on the round complexity is not sufficient in this case. Due to this restriction, the runtime complexity of the EIG protocol becomes exponential in the number of tolerable faults in case of single bit messages. Nevertheless, the EIG protocol is still the best choice (out of the three examined ones) when only a single fault has to be considered. All other analyzed protocols have a linear increase in runtime, if they are converted to single bit messages. The different algorithms were compared using a software simulator where their basic properties in case of the sole usage of single bit messages were checked. Finally we implemented two of the algorithms in hardware and were able to demonstrate that both, the EIG and the Phase King protocol, perform as expected when implemented on an FPGA.

There are still some open questions which were out of the scope for this work. The most important ones are:

1. All implementations considered assume a fully connected network. The impact of a sparse communication structure would be an interesting questions for future research.

2. It is clear that hardware may behave Byzantine but is a Byzantine fault tolerant system necessary in all cases or can more benign fault models be used in some? Where would be the limits of the benign models?

3. Would error correction on the communication links improve the system reliability? Could the fault hypothesis be relaxed in such a case?

4. How does the hardware complexity scale with different timing models? Would a purely synchronous model be smaller than our mesochronous one?

5. We have seen that using eight single bit consensus instances in parallel still lead, in contrast to one eight bit instance, to acceptable input values when handling inexact input data. A more thorough statistical analysis of different input combination and the resulting decision values would be advantageous.

Bibliography

[1] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition).* John Wiley Interscience, March 2004.

[2] Amotz Bar-Noy, Xiaotie Deng, Juan A. Garay, and Tiko Kameda. Optimal amortized distributed consensus. *Information and Computation*, 120(1):93–100, July 1995.

[3] Amotz Bar-Noy and Danny Dolev. Consensus algorithms with one-bit messages. *Distributed Computing*, 4(3):105–110, 1991.

[4] Amotz Bar-noy, Danny Dolev, Cynthia Dwork, and H Raymond Strong. Shifting gears: Changing algorithms on the fly to expedite byzantine agreement. In *Information and Computation*, pages 42–51, 1987.

[5] P. Berman, J. A. Garay, and K. J. Perry. Towards optimal distributed consensus. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, SFCS '89, pages 410–415, Washington, DC, USA, 1989. IEEE Computer Society.

[6] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Optimal early stopping in distributed consensus (extended abstract). In *Proceedings of the 6th International Workshop on Distributed Algorithms*, WDAG '92, pages 221–237, London, UK, UK, 1992. Springer-Verlag.

[7] Piotr Berman and JuanA. Garay. Asymptotically optimal distributed consensus. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and SimonettaRonchi Della Rocca, editors, *Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 80–94. Springer Berlin Heidelberg, 1989.

[8] Piotr Berman, JuanA. Garay, and KennethJ. Perry. Bit optimal distributed consensus. In Ricardo Baeza-Yates and Udi Manber, editors, *Computer Science*, pages 313–321. Springer US, 1992.

[9] T.J. Chaney and C.E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22(4):421–422, April 1973.

[10] Danny Dolev, Michael J Fischer, Rob Fowler T, Nancy A Lynch, and H. Raymond Strong. An efficient algorithm for byzantine agreement without authentication. *Information and Control*, 52:257–274, 1982.

[11] Michael J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *Proceedings of the 1983 International FCT-Conference on Fundamentals of Computation Theory*, pages 127–140, London, UK, UK, 1983. Springer-Verlag.

[12] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[13] Matthias Függer and Ulrich Schmid. Reconciling fault-tolerant distributed computing and systems-on-chip. *Distributed Computing*, 24(6):323–355, 2012.

[14] Matthias Fugger, Ulrich Schmid, Gottfried Fuchs, and Gerald Kempf. Fault-tolerant distributed clock generation in vlsi systems-on-chip. In *Sixth European Conference on Dependable Computing, 2006. EDCC'06.*, pages 87–96. IEEE, 2006.

[15] Juan A. Garay and Yoram Moses. Fully polynomial byzantine agreement for processors in rounds. *SIAM Journal on Computing*, 27(1):247–290, February 1998.

[16] Vassos Hadzilacos and Sam Toueg. Distributed systems (2nd ed.). chapter Fault-tolerant broadcasts and related problems, pages 97–145. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[17] Matthew Heath and Ian Harris. A deterministic globally asynchronous locally synchronous microprocessor architecture. *Fifth International Workshop on Microprocessor Test and Verification*, 0:119, 2003.

[18] L. Kleeman and Antonio Cantoni. Metastable behavior in digital systems. *IEEE Design and Test of Computers*, 4(6):4–19, Dec 1987.

[19] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.

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

[21] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[22] Jennifer Lundelius and Nancy Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62:190–204, 1984.

[23] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[24] Stefan Poledna. Replica determinism in distributed real-time systems: A brief survey. Research Report 6/1993, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 1993.

[25] Thomas Polzer. Fault-tolerant hardware implementation of a consensus algorithm. Master's thesis, Institut für Technische Informatik, 2009.

[26] Thomas Polzer, Thomas Handl, and Andreas Steininger. A metastability-free multi-synchronous communication scheme for socs. In *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS '09, pages 578–592, Berlin, Heidelberg, 2009. Springer-Verlag.

[27] William B. Rouse and Andrew P. Sage. *Handbook of systems engineering and management*. New York, NY, USA, 1999.

[28] Andrew Royal and PeterY.K. Cheung. Globally asynchronous locally synchronous fpga architectures. In Peter Y. K. Cheung and GeorgeA. Constantinides, editors, *Field Programmable Logic and Application*, volume 2778 of *Lecture Notes in Computer Science*, pages 355–364. Springer Berlin Heidelberg, 2003.

[29] Ulrich Schmid, Andreas Steininger, and Manfred Sust. FIT-IT-Projekt DARTS: Dezentrale fehlertolerante Taktgenerierung. *Elektrotechnik & Informationstechnik (e&i) 1-2/07, 2007*, Jan. 2007. (Invited paper).

[30] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[31] Nozer D. Singpurwalla. *Reliability and risk : a Bayesian perspective*. Wiley series in probability and statistics. J. Wiley & Sons, Hoboken (N.J.), San Francisco, Weinheim, 2006.

[32] J. Sparsø. Asynchronous circuit design - a tutorial. In *Chapters 1-8 in "Principles of asynchronous circuit design - A systems Perspective"*, pages 1–152. Kluwer Academic Publishers, Boston / Dordrecht / London, dec 2001.

[33] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[34] John F. Wakerly. *Digital Design: Principles and Practices*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 3rd edition, 2000.

[35] Kenneth Y. Yun and Ryan P. Donohue. Pausible clocking: A first step toward heterogeneous systems. In *In Proceedings of the International Conference on Computer Design (ICCD)*, pages 118–123. IEEE Computer Society Press, 1996.

[36] Shengxian Zhuang, Weidong Li, Jonas Carlsson, Kent Palmkvist, and Lars Wanhammar. An asynchronous wrapper with novel handshake circuits for gals systems. In *In Proceedings of the IEEE 2002 International Conference on Communications, Circuits and Systems*, pages 1521–1525, 2002.