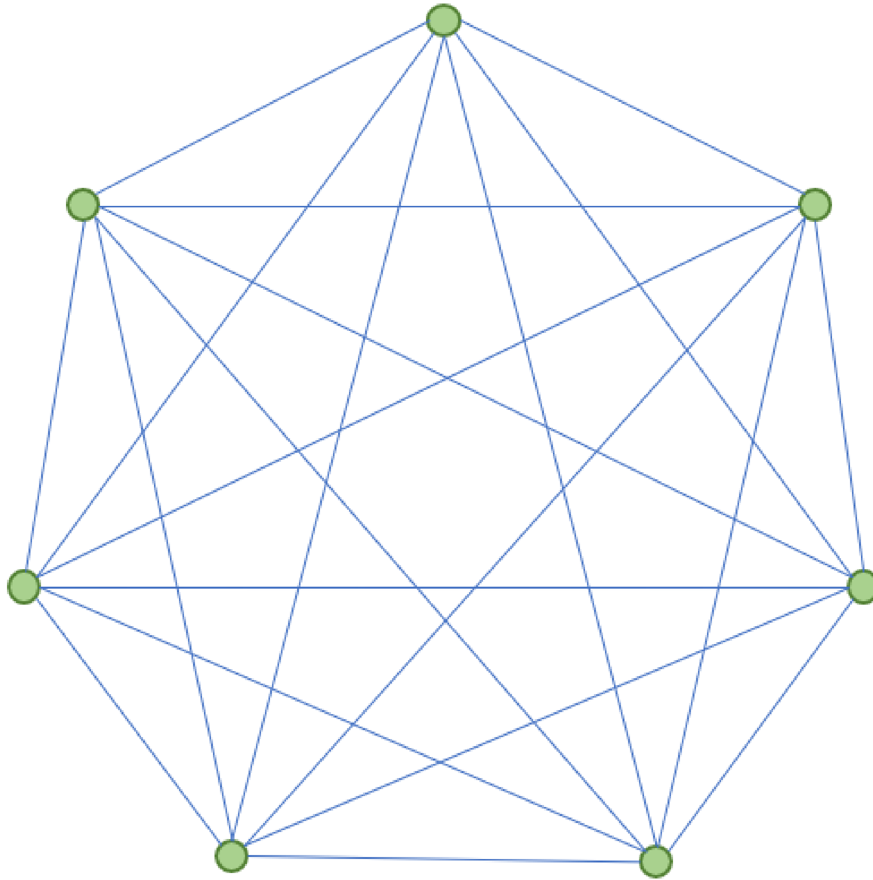

Self-Stabilizing Binary Consensus

Implementation and Evaluation of Self-Stabilizing Binary Consensus

Master's thesis in Computer science and engineering

Amanda Sjöö

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Self-Stabilizing Binary Consensus

Implementation and Evaluation of Self-Stabilizing Binary Consensus

Amanda Sjöö

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science & Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Self-Stabilizing Binary Consensus
Implementation and Evaluation of Self-Stabilizing Binary Consensus
Amanda Sjöö

© AMANDA SJÖÖ, 2021.

Supervisor: Elad Michael Schiller, Department of Computer Science & Engineering
Examiner: Ahmed Ali-Eldin Hassan, Department of Computer Science & Engineering

Master's Thesis 2021
Department of Computer Science & Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Visualization of a fully connected distributed system with seven processors.

Typeset in LaTeX
Gothenburg, Sweden 2021

Self-Stabilizing Binary Consensus
Implementation and Evaluation of Self-Stabilizing Binary Consensus
Amanda Sjöö
Department of Computer Science & Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Binary consensus is a fundamental problem in distributed systems that is especially hard to solve for some system models. In this thesis, we explore binary consensus for a specifically complicated system model; A fully connected asynchronous message passing system with unreliable channels where at most a minority of processors can fail and arbitrary transient faults can happen. This system model extends the system model of previous binary consensus algorithms and is thus even more fault tolerant.

One of the main challenges in distributed systems is to create algorithms that are both efficient and have high reliability and high fault-tolerance. For this thesis, the objective is to find if a binary consensus algorithm in this strict system model can be efficient.

In the thesis we solved binary consensus with two different approaches, using randomization and using the class Ω of failure detectors. We then evaluated the algorithms with focus on efficiency *i.e.,* latency. We also implemented other improvement techniques to our algorithms in order to get an even better performance. The two techniques we used was hybrids and the Look-Ahead method. By implementing all different versions of binary consensus we were able to compare their different behaviours and speculate in their usefulness for different distributed systems. From our tests we especially found that the randomized binary consensus algorithm showed very promising result regarding latency and system scalability.

Keywords: Binary Consensus, Self-Stabilization, Arbitrary Transient Faults, Distributed Systems, Fault-Tolerance, Binary Consensus using Randomization, Failure Detectors.

Acknowledgements

I would like to express my sincerest gratitude to my supervisor Professor Elad Michael Schiller for giving me the opportunity to work with such a fun subject and a great project. I'm especially grateful for all the tremendous support I have received during the project.

I would also like to give a special thanks to Daniel Karlberg and Daniel Kem [53], who have helped me greatly when getting started with the project and whom have always been there to help me.

Finally, I would like to say thanks to my examiner Professor Ahmed Ali-Eldin Hassan for all the encouragement and help.

Amanda Sjöö, Gothenburg, June 2021

Contents

List of Figures xi

List of Tables xiii

1 Introduction 1
- 1.1 The studied problem: Binary Consensus . . . 1
- 1.2 System and fault model . . . 2
- 1.3 Related work . . . 3
- 1.4 Our contribution . . . 3

2 The literature study 5
- 2.1 Summary of how to solve binary consensus . . . 5
  - 2.1.1 Failure detectors . . . 6
    - 2.1.1.1 Failure detectors . . . 6
    - 2.1.1.2 Leader oracle Ω . . . 7
  - 2.1.2 Binary Consensus using Randomization . . . 8
- 2.2 Improvements . . . 8
  - 2.2.1 Hybrids . . . 9
  - 2.2.2 Look-Ahead technique . . . 9

3 Implementations 11
- 3.1 Overview of algorithms . . . 11
- 3.2 Self-stabilizing binary consensus using Ω . . . 11
  - 3.2.1 Self-stabilizing Ω failure detector using ◇*t*-SOURCE assumption 14
  - 3.2.2 Self-stabilizing Ω failure detector using ◇MS_PAT assumption 17
- 3.3 Self-stabilizing binary consensus using common coin . . . 18
- 3.4 Implementation of improvements . . . 21
  - 3.4.1 Ω failure detector-hybrid . . . 21
  - 3.4.2 The Look-Ahead technique . . . 21

4 Evaluation 23
- 4.1 Environment . . . 23
- 4.2 Evaluation criteria . . . 24
- 4.3 Experiments . . . 24
- 4.4 Result of evaluation . . . 25
  - 4.4.1 Binary consensus using CC . . . 25
  - 4.4.2 Binary consensus using Ω failure detector . . . 26

4.4.3 Comparison between Ω and CC 28

5 Discussion and conclusion **33**

Bibliography **35**

List of Figures

4.1 This figure show the results of the evaluation of the algorithm related to the binary consensus algorithm that use randomization and common coin (no. 7). The different sub-figures show the different criteria measured during the evaluation. Figure (a) shows the average latency in milliseconds for the algorithm. (b) shows the time spent waiting for messages in milliseconds. Figure (c) shows the average total number of messages received before coming to a consensus. Lastly (d) shows the average number of rounds for the algorithm. 27
4.2 This figure shows the results of the evaluation of most of the algorithms related to the binary consensus algorithm that use failure detectors, specifically the Ω failure detector (no. 1, 3, 5, and 6). The different sub-figures shows the different criteria measured during the evaluation. Figure (a) shows the average latency in milliseconds for the algorithms. (b) shows the time spent waiting for messages in milliseconds. Figure (c) shows the average total number of messages received before coming to a consensus. Lastly (d) shows the average number of rounds for each algorithm. 29
4.3 This figure shows the comparison of latency between the base algorithm and the version of the algorithm improved with the Look-Ahead technique of the algorithms that was based on the Ω failure detector (no. 1-6). The different sub-figures shows the different ways the Ω failure detector was implemented. Figure (a) shows the latency in millisecond of the two algorithms that used the ◇ MS_Pat assumption. Figure (b) shows the latency in millisecond of the two algorithms that used the ◇*t*-SOURCE assumption. Lastly (c) shows the latency in milliseconds of the two algorithms that used the hybrid version of Ω. 30
4.4 This figure show the evaluation of the binary consensus algorithm that used randomization (no. 7) and the the best binary consensus algorithm that used a failure detector (no. 6). Figure (a) shows the average latency in milliseconds for the algorithms. (b) show the time spent waiting for messages in milliseconds. Figure (c) show the average total number of messages received before coming to a consensus. Lastly (d) shows the average number of rounds for each algorithm. 32

List of Tables

2.1 Table of all failure detector classes with corresponding properties. . . 7

4.1 All versions of the algorithms that was evaluated, expressed in the type of binary consensus algorithm, the oracles used and additional upgrades. 23

1
Introduction

The act of coming to consensus is to have all members of a group agree on the same thing. This is notoriously hard for humans to do and instinctively, one might think computers would solve it easily. However, it turns out that even computers struggle with this task. In the area of distributed systems, this problem is fundamental both because of how complex the problem is to solve and because of the significance it has in solving other distributed problems. The problem, simply called the consensus problem, is not very useful to solve in itself. However, other agreement problems, *e.g.,* state machine replication and atomic broadcast require it. In today's society, there also exist plenty of distributed applications where it is vital to have an underlying consensus algorithm. Examples of such applications are cloud computing [9], clock synchronization [81], blockchains [11], and load balancing [3].

It is safe to say that consensus play a big part in ensuring high system reliability for distributed systems. This also put a lot of pressure in ensuring that the consensus algorithm itself has high fault-tolerance and high reliability, which in turn, gives an incentive to always upgrade the consensus protocols.

The consensus problem is well defined and well-studied. Nevertheless, is it far from completely solved and researchers are still working on finding solutions for more strict system models with high reliability and high fault tolerance. For this master thesis, we study consensus in a very strict system model that is supposed to resemble a real-world distributed system in many ways. Our system model will extend the fault model of previously studied system models and thus make it even more fault tolerant than existing consensus protocols [65].

However, the extended fault model comes at a price of a more complex solution. When the solution is more complex, the computational cost will probably increase. The question of interest is whether it is possible to achieve a high enough performance for the solution to be usable in real world application.

1.1 The studied problem: Binary Consensus

The problem we study in this thesis is called binary consensus and is formally defined as follows.

Every process p_i has to propose a value $v_i \in V$ via an invocation of the $\textbf{propose}_i(v_i)$ *operation, where V is a finite set of values. Let Alg be an algorithm that solves consensus. Alg has to satisfy safety (i.e., validity, integrity, and agreement) and liveness (i.e., termination).*

- **Validity**. *Suppose that v is decided. Then,* $\textbf{propose}(v)$ *was invoked by some process.*
- ***Integrity.*** *Suppose a process decides. It does so at most once.*
- ***Agreement.*** *No two processes decide different values.*
- ***Termination.*** *All non-faulty processes decide.*

When the set, V, of proposal values is unlimited, the problem is called multi-valued consensus. In addition, the problem is called binary consensus when the set V only contains 0 or 1. It is known that multi-valued consensus can be solved by using binary consensus since the problems are computationally related *i.e.,* multi-valued consensus can be reduced to binary consensus. For this reason, this thesis will only focus on solving the problem of binary consensus even though multi-valued consensus is more useful.

1.2 System and fault model

When talking about a problem in distributed systems it is necessary to mention the context it is designed to execute in. These system assumptions are known as the system model and the fault model of a problem and defines the faults which the solution is suppose to handle.

There exist a lot of different system models, ranging from those based solely on real-world systems to more theoretical models. However, there is often some baseline of how the system model is designed. For example, a common setup is to have a network of distributed processors which are vertices in a fully connected graph where the edges are communication links at which messages can be sent. The variation between different models comes in the reliability of links, if processors can fail and other additional assumptions.

We want the system model to resemble a real-life distributed system. For this reason we choose to base our algorithm on an asynchronous model. The main characteristics of asynchronous models are that there is no upper bound in the amount of time process take to receive, process and respond to an incoming message. In reality, assuming asynchrony is arguably stricter than necessary when depicting real life. However, since it is very general without being to unrealistic, it is commonly used anyways.

For this thesis we choose to study a fully connected asynchronous message-passing system of n nodes $\mathcal{P} = \{p_0, \ldots, p_{n-1}\}$. This means that each node in the system can send/receive messages to/from everyone, but there are no guarantees on the com-

munication delay. It is also assumed that each node in the system have a unique id known by everyone.

For the fault model we include communication failures, such as packet loss, duplication and reordering, which is realistic when depicting communication with the UDP-protocol. We also assume processors can fail according to the fail-stop model. We say that node $p_i \in \mathcal{P}$ experiences a fail-stop failure if it stops taking steps forever.

In addition to the failures described above, we also aim to recover from *arbitrary transient faults*, which is described as any temporary violation of assumptions according to which the system and network were designed to operate. An arbitrary transient fault can for example corrupt control variables, such as the program counter, local variables, and indices. These faults are a rare occurrence, but they are not to be overlooked. They could possibly end up leaving the system in a random state from which the binary consensus algorithm can not continue. When an arbitrary transient fault happen, we assume that the violations bring the system to an arbitrary state. An algorithm is called *self-stabilizing* if it is able to recover the system to a executable state even when an arbitrary transient fault happens [65].

1.3 Related work

We note the existence of self-stabilizing algorithms for replication in message passing systems, e.g., [66, 46, 67, 68, 69, 49, 70, 31, 64, 50, 45, 47, 44, 8, 43, 30, 6, 80, 18, 48, 5, 35, 7, 7, 79, 33, 34, 17, 36, 29, 37, 13, 38, 24, 12, 20, 21, 23, 25, 19]. We note that the literature on self-stabilization mainly focuses on communications networks, e.g., [54, 74, 57, 16, 62, 15, 75, 76, 61, 32, 14, 60, 52, 58, 63, 56, 55, 59, 51, 82, 26, 27, 22, 28, 39, 4]. However, this project focuses on the most recent state of the art, as detailed in Section 2.

1.4 Our contribution

We provide, to the best of our knowledge, the first implementation and evaluation of a self-stabilizing binary consensus algorithm that uses the underlying Common Coin principle and the first implementation of a self-stabilizing binary consensus that uses an Ω-failure detector based on the ◇T-SOURCE assumption. We also provide a comparison between two distinctively different self-stabilizing binary consensus algorithms and an evaluation of how the look-ahead technique by [83] can improve an implementation of a self-stabilizing binary consensus algorithm that uses an Ω-failure detector.

2

The literature study

In this chapter we take a look at the answers to two questions. The first question is: "how do we solve consensus within our system model" and is answered in section 2.1 and the second question, "how can we further improve the performance of a consensus algorithm", is answered in section 2.2.

2.1 Summary of how to solve binary consensus

The first thing to notice when studying this subject is that the binary consensus problem is impossible to solve for many system models. A classic impossibility is by M. Fischer, N. Lynch, and M. Paterson called the FLP-impossibility for short [40]. This impossibility states that in an asynchronous system where at least one node can crash-fail, binary consensus is impossible. The impossibility derive from the fact that, since there is no bound on the message delay, it is impossible to distinguish between a crashed process and one that is really slow [40]. In order to proceed, we therefore either need to add some additional assumption to enrich the system enough to circumvent the impossibility or weaken the asynchronous assumption.

As already mentioned, the use of an asynchronous system model is actually stricter than necessary for the intended environment. For example, when using an asynchronous system, it is assumed that there is no bound on the message delays. This in practice means a messages will never arrive at all even though the processor is non-faulty. In real-life, if such a scenario happens, the system would be considered faulty.

Raynal [78] describes three different small and arguably reasonable assumptions that enrich the system enough for it to be possible to solve the binary consensus problem. These three approaches can briefly be described as follows:

- Adding an assumption about the message delivery. The assumption considers that there is a round in which all processes receive the first majority of messages from the same set of correct processes. It is important to note that in this assumption we do not state anything about when this event will happen, only that it will happen at some point for some round r.
- Provide the processes with information on processor failures within the system. This is called the failure-detector based approach. There exist several different classes of failure detectors that each have their working conditions and provide

the processors with some information about failures.
- Change the solution from being deterministic to non-deterministic. The last approach is to introduce randomness into the system. The two models used for this are called local coin (LC) and common coin (CC).

In this thesis we will focus on the two latter approaches. However, before we provide a more detailed description of the two, we will discuss another condition that make consensus impossible. This is related to the number of failures that can be tolerated in our system. It has been proven that the number of fail-stop failures f in the system should be less than $n/2$ [78], where n is the total number of nodes in the system taking part in the consensus protocol. This we will also take into account in our system model.

To summarize, the system model considered in our thesis for binary consensus to be solvable follows.

- Fully connected asynchronous message passing system.
- Every node have a unique id known by everyone.
- Nodes in the system can fail-stop, but at most a minority ($f < n/2$)
- Channels are unreliable, i.e, packet loss, duplication and reordering can happen.
- Arbitrary transient faults can happen.
- Either an available failure detector or random oracle

2.1.1 Failure detectors

A failure detector oracle [41] is an entity used as a building block in distributed systems and provide the processors with information about the stability of the processors. The oracle can either be centralized where all processors can access the same failure detector or decentralized where there exist several local entities, one for each processor in the system. The family of failure detectors is big and there is a large variety between the different types. Examples of failure detector classes include; the class simply called failure detectors, the class Ω of eventual failure detectors and the class θ [78]. In this thesis we use two classes, the Ω and failure detectors.

2.1.1.1 Failure detectors

The failure detector class [41] of detectors adds a local variable called *suspected* to each process p_i. The local variable is used to track node ids of processors which the detector suspects have failed. In this group there exists several types of failure detectors that differ in their accuracy in finding suspicious processors and in the way suspicion is shared amongst the nodes in the system. These differences can be described in the properties *completeness* and *accuracy*, which are used to categorize the types. In the following text the properties are described further.

Degrees of completeness

- Strong completeness: If a process p_j crashes, it eventually appears permanently in the set $suspected_i$ of all correct processes.
- Weak completeness: If a process p_j crashes, it eventually appears permanently in the set $suspected_i$ of some correct process.

Degrees of accuracy

- Strong accuracy. No process p_j appears in a set $suspected_i$ before crashing.
- Weak accuracy. Some correct process never appears in a set $suspected_i$.
- Eventual strong accuracy. There is a time after which no correct process appears in a set $suspected_i$.
- Eventual weak accuracy. There is a time after which some correct process never appears in a set $suspected_i$.

These six properties can be combined in eight different ways, which give us eight types of failure detectors. A presentation of the failure detectors can be found in table 2.1.

Completeness/Accuracy	**Strong**	**Weak**	**Eventually strong**	**Eventually weak**
Strong	Perfect (P)	Strong (S)	Eventually perfect ($\diamond P$)	Eventually strong ($\diamond S$)
Weak	Weak (Q)	Quasi-perfect (W)	Eventually weak ($\diamond Q$)	Eventually quasi-perfect ($\diamond W$)

Table 2.1: Table of all failure detector classes with corresponding properties.

2.1.1.2 Leader oracle Ω

The eventual leader failure detector class Ω [78], is sometimes referred to as a failure detector and sometimes as a leader oracle. This class provides each process p_i with a read-only local variable, $leader_i$, that contains the node id of the processor which the leader oracle thinks is the most suitable to be leader, i.e. the most stable node of the system. The class of Ω failure detectors are defined by the following properties:

- Validity. $\forall i : \forall \tau : leader_i^\tau$ contains a process identity.
- Eventual leadership. $\exists \ell \in Correct(F), \exists \tau : \forall \tau' \geq \tau : \forall i \in Correct(F) : leader_i^{\tau'} = \ell$

These properties state that a common, non faulty leader is eventually elected forever. A leader is called *common* when at least a majority of nodes in the system have the same node as leader. Before this happens, there is an anarchy period where different nodes can have a different leader. It is known that the class Ω of failure detectors is the weakest to solve binary consensus [10]. Unfortunately, the FLP-impossibility also makes it impossible to implement the Ω failure detector in our system model. Raynal [78], mentions two different assumptions that can be used to circumvent the

impossibility.

$\diamond t$-MS-PAT: The eventual message pattern assumption, $\diamond t$-MS_PAT [71], is defined as such. There is a finite time τ, a non-faulty process q, and a set Q of (t + 1) processes such that, after τ, each process $p_j \in Q$ always receives a winning response from q to each of its queries (until p_j possibly crashes). *Winning response* is the name of the first majority of responses to a query. In other words, eventually, each non-faulty node will always receive a response from the same node p_j within the first majority of responses. An advantage of this assumption is that no use of timers or physical time is required. It only states a constraint on the delivery order of some messages.

$\diamond t$-SOURCE : This model uses a definition called eventual timely channel. If the channels are unidirectional, then a channel $ch(i, j)$ is eventually timely if there is a bound δ such that after some finite time τ, the transit time of the message from p_i to p_j is bounded by δ.

The assumption eventual t-source ($\diamond t$- SOURCE) [1] states that there is a non-faulty process p that has t output channels that are eventually timely.

2.1.2 Binary Consensus using Randomization

The impossibility of solving binary consensus in a asynchronous system, where at least one process may crash, is related to determinism. Therefore, one way of circumvent the impossibility is to introduce randomness into the system [78]. The termination property of the consensus problem state that after some finite time every non-faulty node must decide. However, when using randomness this property can only be ensured with some probability. The two models that can be used to introduce randomness into the system is called *local coin* (LC) and *common coin* (CC).

Local Coin (LC) [77]: In this model each process have access to a random number generator or *random-oracle*. Such an oracle is defined by an operation denoted *random()* that returns the value 0 or 1, each with probability 0.5.

Common Coin (CC) [77]: This oracle provide each processes with a primitive denoted *random()* that returns a random bit with the probability of 0.5 each time it is called by a process. The common coin also have a property that differentiate it from LC. This property, which we call the global property, states that the r-th invocation of *random()* by any processor p_i return the same bit b_r. This means that all processors receive the the same random bits in the same order.

2.2 Improvements

In this section we present two different ideas of how binary consensus can be improved. The first one is by create hybrid-versions of already existing algorithms and the second idea is a technique that can called *look-ahead*.

2.2.1 Hybrids

In order to solve binary consensus it is sufficient to use one of the above mentioned approaches but it is also possible to use a combination of several approaches. In theory it can actually be beneficial to combine approaches and create a hybrid solution. During the literature study we found two such hybrids [78].

Hybrid Ω failure detector: The first is a Ω failure detector that uses two different underlying assumptions. In this case we can choose to use either the $\diamond t$-MS-PAT assumption or the $\diamond t$-SOURCE assumption to implement a Ω in our system model, but another option is to combine the algorithms which give an algorithm that builds an eventual leader in a system model whose runs satisfy at least one or both assumptions. Since the two assumptions are computationally incomparable to each other, such a hybrid algorithm benefits from the best of both worlds [72]. It will provide an increased overall assumption coverage, which in theory should result in the system finding a common leader faster.

Hybrid binary consensus algorithm: The second hybrid is a binary consensus algorithm that uses both an eventual leader oracle and a random oracle. Similarly to the other hybrid, by combining the different solutions, you end up with an solution that can benefit from the best of both worlds. A binary consensus algorithm that can come to a consensus as soon as the leader oracle behaves correctly (common leader exists for the system) and possibly earlier if the random oracle behaves correctly (every node have the same value) [2].

To summarize, by using a hybrid solution it is possible to get an algorithm that takes advantage of the advantageous properties of several failure detectors/oracles at the same time as the negative parts become less impactful. The downside is that more than one additional assumption is needed and the hybrid versions will be a bit more computationally heavy than the non-hybrid equivalent.

2.2.2 Look-Ahead technique

Many binary consensus algorithms proceed in consecutive rounds. In each round the processors exchange messages with some information about the current state (can be current leader, proposed value etc.). A processor will use the information to try make a decision and continue to the next round. Since each processor work at their own pace, due to the asynchronous nature of the system, different processors can be in different rounds at the same time. The basic idea of the Look-Ahead technique is to make use of *future messages i.e.,* messages sent from other nodes in rounds with a higher value than you.[83].

In some cases, these future messages, contains information that a processor p_i can use in order to faster come to a decision in an earlier round. For example, in many algorithms processors share their decision for a round with everyone else before continuing to the next round. In this case, if a slow node receive a message containing such decision from a future round it could use this information to also make a deci-

sion. However, not every future message is useful and it is therefore very important to consider which future messages can be used for different algorithms.

3

Implementations

In this chapter we will begin with giving an overview of the algorithms and implementations we have implemented. The next section will describe each algorithm more in detail. The last section will give a better description of how to implement the improvements.

3.1 Overview of algorithms

The algorithms we implemented are a binary consensus algorithm that uses a failure detector and one that uses randomness. When it comes to the choice of failure detector we choose to use the Ω failure detector which was implemented in three different ways, using the $\diamond t$-MS-PAT assumption, with the $\diamond t$-SOURCE assumption and using the hybrid version that combine both assumptions. For the binary consensus algorithm that uses randomness we use the common coin (CC) random oracle since the two approaches, LC and CC, are quite similar but CC is expected to be faster due to its global property. Lastly, we will use the Look-Ahead technique on all algorithms that use the Ω failure detector to improve the performance even more. Just to summarize, these are the algorithms we have implemented:

Base binary consensus algorithms:

- Binary consensus using common coin (CC)
- Binary consensus using eventual leader (Ω) oracle
  - Ω failure detector with $\diamond t$-SOURCE assumption
  - Ω failure detector with $\diamond$MS_PAT assumption

Hybrid BC algorithm:

- Hybrid binary consensus using two eventual leader Ω failure detectors ($\diamond t$-SOURCE and $\diamond$MS_PAT)

3.2 Self-stabilizing binary consensus using Ω

The self-stabilizing binary consensus algorithm that uses an Ω presented in this paper will from now on be refered to as algorihm 1. Algorithm 1 builds a binary consensus algorithm for our system model and fault model. It does this with the assumption that the algorithm have access to a Ω failure detector *i.e.,* an eventual leader oracle that also work in our system and fault model. The algorithm is based on an algorithm by A.Mostéfaoui and M.Raynal [73]. However, it has been modified

to also handle arbitrary transient faults as shown by O.Lundström, M.Raynal and E.Schiller [65]. The pseudo code of the algorithm is seen in 1.

Underlying principle: The idea of this binary consensus algorithm is to use the property of the Ω failure detector *i.e.,* eventually a unique non-faulty node will be elected as leader for all non-faulty nodes. When a common leader exists for the system *i.e.,* at least a majority of nodes have the same leader for a round r, the system can come to a consensus by deciding the value which the common leader proposes. The problem is that the nodes don't know when a common leader exists. Therefore, the objective of the algorithm is to figure out when this property of the Ω failure detector is satisfied.

The algorithm itself proceeds in consecutive rounds containing two phases, 0 and 1. In phase 0 the goal is for each node to see if a common leader exists. This is called local confirmation, *i.e.,* when a individual node have noticed a common leader. The nodes does this by exchanging messages with information about their current leader. In phase 1 the objective is to confirm that at least a majority of nodes also know about the common leader and thus transform the local confirmation to a global confirmation. Only when at least a majority of nodes have knowledge about a common leader is it safe to decide a value.

Local variables at each process: Each process p_i manages the following local variables:

- $est[r][i][p]$: array that contains both estimation values for the phases $p \in \{0, 1\}$ for each node $p_i \in \mathcal{P}$ at every round r.
- $phs[r][i]$: array that contain the current (or at least last known) phase of each process $p_i \in \mathcal{P}$ for each round r.
- myLead$[r][i]$: array that stores the leader of each process $p_i \in \mathcal{P}$ for each round r.
- decVal$[i]$: store the decided value of each process $p_i \in \mathcal{P}$.

Behavior of the process: Each node start each round by selecting a new leader with the help of the Ω failure detector (line 11). To find out whether a common leader exists for the round, the nodes exchange their information with one another. The id of the leader, together with current round and phase values are broadcast to everyone until the node received responses with the corresponding round and phase numbers from at least a majority of other nodes (lines 13-21). A response contains the same information as the initial message. From the responses, a node check if there exists a common leader or not (lines 16-17) and choose an estimation for phase 1 thereafter. p_i will use $\perp$ as estimation in case there is no common leader, and the value v from the leader otherwise. An estimated value v is also shared in these initial messages.

In phase 1, p_i share its estimation with all other nodes until a majority of corresponding responses arrive (lines 20-21). Depending on the responses of phase 1, three different things happen. If a majority of nodes confirm a common leader, p_i

decide this value (line 24). If at least one node have seen a common leader, p_i adopt that value into the next round for phase 0 (line 25). In the last scenario, no node that responded have seen a common leader. In this case p_i continue to the next round and phase with the same estimation (line 26).

Upon the arrival of a PHASE message from p_j, p_i will save the estimated value, leader value and decided value if there are any (lines 31, 32, 33). If necessary p_i will also respond to the message with a PHASE message (line 34).

Fault handling: To handle transient faults, p_i will check for any strange behavior of the variables est_i and phs_i. If p_i can see that some round is skipped, the object will be deactivated (lines 9 and 10). p_i can be activated again by the arrival of a PHASE message from another node (line 28).

Algorithm 1 Self-stabilizing algorithm for binary consensus using Ω; code for p_i

1: $\mathit{initState} = ([[-1, ..., -1], ...], [[[\bot, \bot], ..., [\bot, \bot]], ...], [[\bot, ..., \bot], ...], [[\bot, ..., \bot], ...]);$
2: $\mathit{r(k)} := \max(\{-1\} \cup \{r \in \mathbb{Z}^+ : \mathit{phs}[r][i] \geq 0\});$
3: $\mathit{phsEnd}(x)$ **do return** $\mathit{phs}[r(i)][i] = x \wedge (|\{p_k \in \mathcal{P} : r(k) = r(i) \wedge \mathit{phs}[r(k)][k] = x\}| \geq n - t);$
4: $\mathit{phsZeroBreak}()$ **do return** $\mathit{phs}[r(i)][i] = 0 \wedge (\exists p_k \in \mathcal{P} : r(k) > r(i) \vee (r(k) = r(i) \wedge \mathit{phs}[r(i)][k] = 1));$
5: $\mathit{decided}()$ do return $(O \neq \bot \wedge |\{p_k \in \mathcal{P} : O.\mathrm{decVal}[k] \neq \bot\}| \geq t + 1);$

6: **operation** propose(v) **do** $\{((\mathit{phs}, \mathit{est}, \mathrm{myLead}, \mathrm{decVal}), \mathit{est}[0][i][0]) \leftarrow (\mathit{initState}, v)\}$
7: **operation** result() if $\mathit{decided}()$ **then return** decVal **else return** $\bot$

8: **do forever**
9:     if $(\mathit{est}[r(i)+1][i][0] \neq \bot) \vee (\mathit{phs}[x][i] \notin \{0,1\} \vee \bot \in \{\mathit{est}[x][i][0], \mathrm{myLead}[x][i]\} : x \in \{0, ..., r(i)-1\})$
10:         **then** $O \leftarrow \bot;$
11:     **if** $\nexists p_k \in \mathcal{P} : \mathrm{decVal}[k] \neq \bot$**then** $(\mathrm{myLead}[r(i)+1][i], \mathit{phs}[r(i)+1][i]) \leftarrow$(leader, 0);
12:     **else if** $\mathrm{decVal}[i] = \bot$ **then** $\mathrm{decVal}[i] \leftarrow \mathrm{decVal}\ [k];$
13:     **repeat**
14:         **if** $\mathit{phs}[r(i)][i] = 0$ **then**
15:             **if** $\mathit{phsEnd}(0)$ **then**
16:                 **if** $(\exists p_\ell \in \mathcal{P} : \mathit{est}[r(i)][\ell][0] \neq \bot \wedge \exists S \subseteq \mathcal{P} : |S| \geq n - t \wedge \forall p_k \in S : \mathrm{myLead}[r(i)][k] = \ell)$
17:                     **then** $\mathit{est}[r(i)][i][1] \leftarrow \mathit{est}[r(i)][\ell][0]$ **else** $\mathit{est}[r(i)][i][1] \leftarrow \bot;$
18:                 $\mathit{phs}[r(i)][i] \leftarrow 1;$
19:             **else if** $\mathrm{myLead}[r(i)][i] \neq$ leader then $(\mathit{est}[r(i)][i][1], \mathit{phs}[r(i)][i]) \leftarrow (\bot, 1);$
20:         **broadcast** PHASE(True, $r(i), \mathit{phs}[r(i)][i], \mathit{est}[r(i)][i]$, $\mathrm{myLead}[r(i)][i]$, $\mathrm{decVal}[i]$);
21:     **until** $\exists p_k \in \mathcal{P} : \mathrm{decVal}[k] \neq \bot \vee \mathit{phsEnd}(1);$
22:     **let** $\mathit{rec} = \{\mathit{est}[r(i)][k][1] : p_k \in \mathcal{P} \wedge \mathit{phs}[r(i)][i] = 1\};$
23:     **switch** True **do**
24:         **case** $\{v\} = \mathit{rec}$ **do** $(\mathit{est}[r(i)+1][0], \mathrm{decVal}[i]) \leftarrow (v, v);$
25:         **case** $\{\bot, v\} = \mathit{rec}$ **do** $\mathit{est}[r(i)+1][0] \leftarrow v;$
26:         **case** $\{\bot\} = \mathit{rec}$ **do** $\mathit{est}[r(i)+1][0] \leftarrow \mathit{est}[r(i)][0];$

27: **upon** PHASE($aJ, rJ, pJ, eJ, \ell J, dJ$) **arrival from** p_j**do begin**
28:     **if** $O = \bot \wedge eJ[0] \neq \bot$ **then** $O.((\mathit{phs}, \mathit{est}, \mathrm{myLead}, \mathrm{decVal}), \mathit{est}[0][i][0]) \leftarrow (\mathit{initState}, eJ[0]);$
29:     **if** $pJ \notin \{0,1\} \vee eJ[0] = \bot$ **then return**;
30:     $\mathit{phs}[rJ][j] \leftarrow \max\{\mathit{phs}[rJ][j], pJ\};$
31:     **foreach** $x \in \{0,1\} : \mathit{est}[rJ][j][x] = \bot$ **do** $\mathit{est}[rJ][j][x] \leftarrow eJ[x];$
32:     **if** $\mathrm{decVal}[j] = \bot$ **then** $\mathrm{decVal}[j] \leftarrow dJ;$
33:     **if** $\ell J \neq \bot$ **then** $\mathrm{myLead}[rJ][j] \leftarrow \ell J;$
34:     **if** aJ **then send** PHASE(False, $rJ, \mathit{phs}[rJ][i], \mathit{est}[rJ][i]$, myLead[rJ][i], decVal) **to** p_j;

3.2.1 Self-stabilizing Ω failure detector using $\diamond t$-SOURCE assumption

Algorithm 2 builds an Ω failure detector for our system and fault model. This algorithm uses the $\diamond t$-SOURCE assumption to circumvent the FLP-impossibility. The algorithm is based on a algorithm due to M. Aguilera, C. Delporte, H. Fauconnier, and S. Toueg [1] but is altered in order to also handle arbitrary transient faults and message duplication. The pseudo code for the algorithm can be found at 2. The boxed in lines of the code show the additional lines added in order to make the algorithm self-stabilizing.

Underlying principle: The purpose of an Ω failure detector is to get an overview of the stability of the nodes in the system and be able to pinpoint the most stable node (which will be proposed as leader). For this specific algorithm, we will use timers in order to determine the stability of nodes. Another algorithm that want to use the Ω failure detector is be able to read the value $leader_i$ (49-51) where Ω will return the most node it thinks is the most stable.

With the $\diamond t$-assumption we know that eventually, t channels of a non-faulty node, will be bounded by some unknown delay. In algorithm 2, the nodes monitor each other with the use of timers while periodically broadcasting alive-messages. The timer corresponding to each node start at some value β and will eventually increase if p_i have not received an alive-message within the deadline. Each time a node misses a deadline, p_i will also locally raise suspicion against the node and if enough nodes in the system have locally suspected the same node, then the node will be suspected globally.

Local variables at each process: Each process p_i manages the following local variables. The two arrays $timer_i$ and $timeout_i$, where $timer_i$ stores all the timers associated with each node and $timeout_i$ stores the current deadline for each node. The variable $count_i$ stores the amount of times each node have been suspected globally. $suspect_i$ is an array that contains the identities of the process that p_i currently suspect to have crashed. p_i also have a variables $recFrom_i$ and $msgId_i$ which store, the last message id (both ALIVE and SUSPECT id) for each node and p_i's own message id respectively. The algorithm also need three different constant value for the algorithm, *beta* which is the start deadline for each node in the system, B which is a big upper bound on the communication delay and δ which is a max gap between the lowest and the highest value in $count_i$.

Behavior of a process: A process p_i periodically send ALIVE messages containing its values in $count_i$ and the message id p_i expects to receive next from node p_j (lines 14-19). The ALIVE message have two aims: to inform the other processes that p_i is still alive and share its suspicion view. When an ALIVE message is received from a node, p_i will compare the *count* list received from p_j with its own (lines 23-25) and reset its local timer of p_j to the current value of $timeout_i[j]$ (line 26).

If the timer of processor p_k expire for the processor p_i, it suspects p_k to have crashed, but only as a local suspicion. A processor p_i only increase a processor p_k's count value in case at least a majority of nodes all suspect p_k. p_i will therefore broadcast a SUSPECT message containing the id of the node whose timer expired (line 31). p_i then increases the deadline of p_k and restart the timer (line 32-33). When a SUSPECT message arrives at p_i, it adds the node p_j to the list of whom suspects the node k (line 39). If a majority of nodes are included in the list that suspects k, its counter increases (lines 41-43) and the suspicion is reset for node k at line 44.

Lastly, if the value $leader_i$ is read by the upper layer, the algorithm will return the node with the lowest value in $count_i[k]$ (line 49-51).

Fault handling: In order to handle various faults a few measures need to be taken. The first problem has to do with message duplication. If we had no way of identifying messages and a single message was received multiple times, it would reset the timers multiple times. In this algorithm, we don't have rounds or phases to identify messages with. We therefore instead use message id (one for ALIVE and one for SUSPECT). By storing the newest message id received by each node (lines 21 and 37), p_i can compare message ids received and see if a message is new or not. We only handle new messages by handling messages with a larger id than earlier received (lines 22 and 38). However, there is one more problem to solve. Since arbitrary transient faults can happen, there could be a situation where one of the node's array $recFrom$ is corrupted and each value in the variable is changed into really large values. In order to accept new messages, all other nodes must first catch up to these really large values. The way to fix this problem is by having each node share what message id they expect to receive next from each node whenever they send a message (lines 16, 21, 31, 37).

Another problem that can happen is related to the variable *count*. This variable is used to determine which node is elected leader. If we assume that a transient fault happens that corrupts a single value in $count_i$ to be a really large number, then p_i would not be able to elect that affected node to leader until every other value in $count_i$ caught up. This can take a really long time. This problem is solved with the help of constant δ. δ help bound the max gap between the lowest and highest value in $count_i$. We also take two different actions to make sure that *count* is stable. First, we don't let any value in *count* be larger than δ + min($count_i$) (line 41). Second, we use the function *check()* every time $count_i$ is updated (lines 28 and 47). *check()* will inspect *count* to see if a transient fault happened that made the gap between lowest and highest value in $count_i$ be larger than δ. If this is the case, the values in $count_i$ will be changed to be within the range again (line 2).

The last problem we need to fix is related to the variable $timeout_i$. In case an arbitrary transient fault happens which changed one of the values in $timeout_i$ to a really large value, the next time the timer is updated with this really large value, the timer will not expire for a long time. This causes a problem to the whole funda-

mental idea of the algorithm, to use increasing timers to find the most stable node. The way this is solved is by introducing yet another constant, this time B, which is a upper bound of how long time a processor needs in order for a single message to be delivered. If a transient fault happens and some value in $timeout_i$ is bigger than B the variable is reset to start at β again (line 3).

Algorithm 2 Self-stabilizing Ω using $\diamond t$-assumption; code for p_i

```
1:  macro counts() := {count[k] : p_k ∈ P};
2:  macro check() := if max counts() - min counts() > δ then foreach p_k ∈ P do
    count[k] ← max {count[k], (max counts() − δ)};
3:  macro timeout_check() := foreach p_k ∈ P do
    if timeout_i[k] > B then timeout_i[k] ← β

4:  macro init(v) :=
5:      msgId_i[] ← [0,0]; //stores the current message id for both ALIVE (index 0)
        and SUSPECT-messages (index 1).
6:      const δ ← v;
7:      for each k do
8:          count_i[k] ← 0; suspect_i[k] ← ∅; timeout_i[k] ← β;
9:          recFrom_i[k] ← [0,0]; //Stored the last message ids (both ALIVE and SUSPECT)
            received from each processor.
10:         if (k ≠ i) then
11:             set timer_i[k] to timeout_i[k]
12:         end if
13:     end for

14: repeat every β time units
15:     for each j ≠ i do
16:         send ALIVE(count_i, msgId_i[0], recFrom_i[j][0] + 1) to p_j
17:     end for
18:     msgId_i[0] ← msgId_i[0] + 1;
19: end repeat

20: when ALIVE(count, id, nextId) is received from p_j do
21:     msgId_i[0] ← max (msgId_i[0], nextId);
22:     if recFrom_i[j][0] < id then
23:         for each k ∈ {1, ..., n} do
24:             count_i[k] ← max(count_i[k], count[k]);
25:         end for
26:         set timer_i[j] to timeout_i[j]
27:         recFrom_i[j][0] ← id;
28:         check();
29:     end if

30: when timer_i[k] expires do
31:     broadcast SUSPECT(k, msgId_i[1], recFrom_i[k][1] + 1);
32:     timeout_i[k] ← timeout_i[k] + 1;
33:     set timer_i[k] to timeout_i[k];
34:     msgId_i[1] ← msgId_i[1] + 1;
35:     timeout_check();

36: when SUSPECT(k, id, nextId) is received from p_j do
37:     msgId_i[1] ← max (msgId_i[1], nextId);
38:     if recFrom_i[j][1] < id then
39:         suspect_i[k] ← suspect_i[k] ∪ {j};
```

```
40:         if (|suspect_i[k]| ≥ (n - t)) then
41:             if [count_i[k] < δ + min counts()] then
42:                 count_i[k] ← count_i[k] + 1;
43:             end if
44:             suspect_i[k] ← ∅;
45:         end if
46:         [recFrom_i[j][1] ← id;]
47:         [check();]
48:     end if
49: when leader_i is read by upper layer do
50:     let (−, l) = min({(count_i[x], x)}_{1≤x≤n});
51:     return(l);
```

3.2.2 Self-stabilizing Ω failure detector using ◇MS_PAT assumption

Algorithm 3 builds an Ω failure detector by using the ◇MS_PAT assumption. Algorithm 3 is due to O. Lunström, M. Raynal, and E. Schiller [65] which is an self-stabilizing version of an algorithm by A. Mostéfaoui, E. Mouragaya, and M. Raynal [71]. The pseudo code of the algorithm can be found at 3

Underlying principle: Algorithm 3 have the assumption that eventually, each process always have the same process p_j in their winning response (the first majority of responses). Node p_i broadcast ALIVE message until the arrival of the corresponding RESPONSE messages from at least a majority of nodes called the winning response. Every node not included in any of the winning responses will be suspected (the count for the amount of times p_i suspected that node will increase). According to the assumption, eventually one node will always be in the winning response of all nodes and thus never be suspected by anyone again. The suspicion counter for that node will be stable and eventually this node will be elected leader. In order to use the Ω failure detector an receive a suggestion for a leader, another algorithm can call the public function *leader*() which will return the node with which is least suspected.

Local variables at each process: Algorithm 3 have the variable r_i that is the current round number of p_i, $recFrom_i$ that is the set of processors that replied to the most recent query, and $count_i$ that keep track of how many times each node have been suspected. p_i also have the constant *delta* which is the largest allowed gap between the biggest and smallest values in $count_i$.

Behavior of the process: Algorithm 3 repeatedly execute a do-forever-loop that broadcast ALIVE messages and collects their replies (lines 12-14). A message contains information about which nodes responded to a nodes last query. To identify corresponding messages, every process uses round numbers that is linked to each messages. From the responses received in line 13, p_i then create a set of all nodes they had as winning response during the last iteration (line 15). All nodes not in the set will be suspected with a increased *count* value (line 16).

When p_i receive an ALIVE message from p_j, we merge the two *count* variables

(line 20) and respond to the message with a RESPONSE. When a RESPONSE message arrive from p_j, p_i will again just merge the *count* variables.

Lastly, in case the operation $leader()$ is used by another algorithm, we return the node that have been suspected least amount of times, and thus have the lowest count value in $count_i[k]$ (lines 6).

Fault handling: This algorithm have a problem where a transient fault can change the variable $count_i$ to have a really large value. This could, in the worst possible scenario, lead to it taking a long time before the algorithm can elect a common stable leader. More information on how this problem is solved by using a constant δ and the function *check*() is already explained in section 4.3.1 in the paragraph about fault handling.

Algorithm 3 Self-stabilizing Ω using MS-PAT; code for p_i

```
1: local constant, variables and their initialization:
2: const δ;
3: r := 0;
4: recFrom := P;
5: count[0..n − 1] := [0, ..., 0];
6: operation leader() {let(−, x) := min {(count[k], k)} : p_k ∈ P; return (x)}
7: macro counts() := {count[k] : p_k ∈ P};
8: macro check() := if max counts() - min counts() > δ then foreach p_k ∈ P do
9:     count[k] ← max{count[k], (max counts() - δ)};
10: do forever begin
11:     r ← r + 1;
12:     repeat
13:         foreach j ≠ i do send ALIVE(r, count) to p_j;
14:     until RESPONSES(rJ, −, recFromJ) received from (n − t) processors;
15:     let prevRecFrom := ∪ sets of recFromJ received in line 13;
16:     foreach j ∉ prevRecFrom : count[j] < δ+ min counts() do count[j] ← count[j] + 1;
17:     recFrom ← {processors from which p_i has recived RESPONSE(rJ, −) in line 13};
18:     check();

19: upon ALIVE(rJ, countJ) arrival from p_j begin
20:     foreach p_k ∈ P do count[k] ← max(count[k], countJ[k]);
21:     check();
22:     send RESPONSE(rJ, count , recFrom) to p_j;

23: upon RESPONSE(rJ, countJ, recFromJ) arrival from p_j begin
24:     foreach p_k ∈ P do count[k] ← max(count[k], countJ[k]);
25:     check();
```

3.3 Self-stabilizing binary consensus using common coin

Algorithm 4 builds a binary consensus algorithm by using randomness with the common coin (CC) model. For this implementation, we integrate the CC-oracle into the

binary consensus algorithm since it is very easy to implement. The algorithm is a self-stabilizing version of an algorithm due to R. Friedman, A. Mostéfaoui, and M. Raynal [42]. The pseudo code of the algorithm can be seen in algorithm 4.

Underlying principle: The objective of the algorithm is for each process to start a round with the same estimated value. When this happen, the algorithm have a 0.5 probability of deciding that round and every round after that. If there is no common value during a round, each process will adopt the value proposed by the random oracle. In the round after, every processor will have the same estimated value.

Local variables at each process: Each processor manages the following local variables: est_i that stores the estimated value of each node at each round. The decided value of each node is also stored in this variable at index $est[M]$. r_i stores the current round number and M is a constant that is used to restrict the amount of memory the variable est_i is allowed to use. Finally each process have the variable s_i that contains the random bit associated with the current round.

Behavior of the process: A new round starts by increasing the round number and fetching the random bit from the round oracle (lines 10-11). Then each process broadcasts its current state with a message containing both its current estimation and its decided value (line 13) until p_i receives corresponding messages from at least a majority of nodes (line 14). At line 15 p_i checks if someone decided a value. If this is the case, we also decide the value. Else p_i checks if there exist a common value for this round. If no common value exists p_i adopt the random value of the random oracle as its estimation for the next round (line 16). Lastly, if a common value exists p_i uses the value as estimation for the next round and if the value is also the same as the random bit s_i, p_i decides the value (lines 17-19).

Upon the arrival of an EST-message, p_i stores the estimated value vJ and the decided value wJ from p_j in est_i and if aJ is true, it means the message requires a response, then p_i responds to the message with its own estimation for the round rJ and its decided value.

Fault handling: In algorithm 4, there are a few faults we must handle. The way we handle asynchrony and unreliable channels is by repeating a broadcast until we receive all answers needed in order to continue (line 12-14). When p_i sends a message via broadcast we also require a response from the nodes we sent the message to. By doing this, a slower node can still receive the other nodes estimation for that particular round even though all the other nodes are at a higher round (lines 13 and 23). One problem that is connected to this is that every node needs to store at least all the estimations between the round number of the slowest node up to its own round number. It is also advantageous to store the estimations of rounds from nodes who are at round above your own round as well. The best thing would be to store the whole range of estimations from the lowest to the highest round numbers. However, in a asynchronous system where each node have its own processing speed,

this range could theoretically be infinitely large. For a self-stabilizing algorithm, infinite memory is impossible.

In order to bound the amount of memory the estimation variable est_i use, we introduce the constant M that is the max gap between the lowest and the highest round number in the system. If any process is about to cross over the limit, it will be forced to wait (lines 7-9). When we limit the range of rounds by M, we can also bound the variable est_i by M. There is still one problem to solve however. In case a node fails at some round r, then no other process would be able to continue further than round $r + M$. The solution is to use a perfect failure detector to dismiss a any processor that have failed during this step. This way, when checking for all the current rounds in the system, we only consider nodes that are not failed (line 5).

Implementation of the common coin: For this algorithm the random-oracle is crucial. Fortunately it is also rather simple to implement and can easily be integrated into the binary consensus algorithm. We do this by providing each process with the same random generator, which is initialized with the same seed. If each process only get a new random value every new round, each processor will have the same random value for a round.

Algorithm 4 Self-stabilizing binary consensus using Common Coin; code for p_i

1: **variables**: $r := 0; est[0, ..., M][0, .., |\mathcal{P}| - 1] := [[\bot, ..., \bot], ..., [\bot, ..., \bot]]$
2: **operation** propose(v) **do** $\{(r, est) \leftarrow (0, [[\bot, ..., \bot], ..., [\bot, ..., \bot]]); est[0][i] \leftarrow (v, 0)\}$
3: **operation** result() **do** {if vals(ℓ) $= \{v\} : v \in \{0, 1\}$ **return** $est[M][i]$ **else return** $\bot$}
4: **macro** vals(ℓ) $:= \{w \in \{0, 1\} : (|\mathcal{P}|/2) < |\{p_k \in \mathcal{P} : est[\ell \bmod M][k] = (w, -) \vee est[M][k] = (w, -)\}|\}$
5: **macro** allRndNum() **return** $\{r'' : r' \in \{0, ..., M - 1\} \wedge k \in \text{trusted} \wedge est[r'][k] = (-, r'')\}$
6: **do forever begin**
7: **let** $(maxX, minX) \leftarrow (\max \text{allRndNum}(), \max\{\min \text{allRndNum}(), maxX - (M - 2)\})$
8: **foreach** $x \in (\{0, ..., M - 1\} \setminus \{y \bmod M : y \in \{minX, ..., maxX\}\})$ do $est[x] \leftarrow [\bot, ..., \bot]$;
9: **if** $(maxX - minX) \geq M - 1$ **then continue**;
10: $r \leftarrow r + 1$;
11: $s_i \leftarrow$ **randomBit()**;
12: **repeat**
13: **broadcast** EST(True, $r, est[r - 1 \bmod M][i], est[M][i]$)
14: **until** $n - t \leq |\{p_k \in \mathcal{P} : est[r \bmod M][k] \neq \bot \vee est[M][k] \neq \bot\}|$;
15: **if** $\exists p_k \in \mathcal{P} : est[M][k] = (w, -) \wedge est[M][i] = \bot$ **then** $est[M][i] \leftarrow (w, r)$;
16: **else if** vals(r) $\neq \{v\} : v \in \{0, 1\}$ **then** $est[r \bmod M][i] \leftarrow (s_i, r)$;
17: **else**
18: $est[r \bmod M][i] \leftarrow (v, r)$;
19: **if** $est[M][i] = \bot \wedge = s_i$ **then** $est[M][i] \leftarrow (v, r)$;

20: **upon** EST(aJ, rJ, vJ, wJ) **arrival from** p_j **do begin**
21: **if** $vJ = (w, rJ, -1) : w \in \{0, 1\}$ **then** $est[rJ \bmod M][j] \leftarrow (w, rJ)$;
22: **if** $wJ = (w, -) : w \in \{0, 1\}$ **then** $est[M][j] \leftarrow (w, rJ)$;
23: **if** aJ **then send** EST(False, $rJ, est[rJ \bmod M][i], est[M][i]$) to p_j;

3.4 Implementation of improvements

3.4.1 Ω failure detector-hybrid

The approach to build a hybrid Ω failure detector is easy to implement as long as you have access to two different Ω failure detectors algorithms. To achieve the hybrid we let the two algorithms (Ω failure detector using ◇*t*-SOURCE and Ω failure detector using ◇MS_PAT) run in parallel and when the variable *leader* is read by the upper layer we return the node id of the node with the lowest overall suspicion count from the two $count_i$ variables, as seen in algorithm 5.

Algorithm 5 Modification of Ω failure detector to create hybrid Ω failure detector

1: **when** $leader_i$ **is read by the upper layer do**
2: **for each** $x \in \{1, ..., n\}$ **do** $count_i[k] \leftarrow \min(mp_count_i[x], tsource_count_i[x])$ **end for**;
3: **let** $(-, \ell) = \min\ (\{(count_i[x], x)\}_{1 \leq x \leq n})$;
4: return ℓ;

3.4.2 The Look-Ahead technique

The Look-Ahead technique is unique in the way that it can be added to almost all binary consensus algorithms as long as the system is asynchronous. The most difficult task is to identify which future-message that can be useful and what actions can be allowed to not violate the safety requirements of binary consensus.

For the binary consensus algorithm that uses the Ω failure detector the following actions will be implemented for the following events:

- If node p_j receives a phase 1 message from p_k in higher round while p_j is in phase 0: stop wait and adopt estimated phase 1 value from p_k as estimation for phase 1.
- If node p_j receives a phase 1 message from node p_k in higher round while p_j is in phase 1: stop waiting and continue to the next round.

4

Evaluation

This chapter describes how the implementations was evaluated. Here we present the test environment, the evaluation criteria we used for the tests, and the experiments that was conducted. These three topics will be explained in their corresponding paragraph, but first a summary of the algorithms is presented.
In table 4.1 we present all versions of binary consensus algorithms that have been implemented and evaluated. To summarize: we have the three different ways of implementing an Ω oracle, MS-Pat and T-SOURCE and the hybrid that uses both of them and we have the algorithm that uses randomness with the CC-oracle. Lastly we improved the binary consensus algorithms that uses Ω with the Look-Ahead technique, and thus have three more version of algorithms.

nr.	Binary Consensus base	Oracle used	Additional upgrades
1.	Ω	MS-Pat	-
2.	Ω	MS-Pat	Look-Ahead
3.	Ω	T-SOURCE	-
4.	Ω	T-SOURCE	Look-Ahead
5.	Ω-hybrid	T-SOURCE & MS-Pat	-
6.	Ω-hybrid	T-SOURCE & MS-Pat	Look-Ahead
7.	Randomness	CC	-

Table 4.1: All versions of the algorithms that was evaluated, expressed in the type of binary consensus algorithm, the oracles used and additional upgrades.

4.1 Environment

The tests was completed in a local testing environment, using a single machine. By using a local environment it is possible to control every aspect of the system, like faulty nodes, message delays and so on in order to build reliable evaluation performances.

The environment operates by assigning each processor in the system to an individual socket on the local machine. The nodes can then communicate with each other on the loop-back interface address (127.0.0.1) by using an underlying message-passing system.

4.2 Evaluation criteria

In this section the evaluation criteria we use and why is presented. The criteria that are used are latency, idle time, number of rounds, and number of messages.

The first and most important evaluation criteria we consider is latency. In this context we define latency as the time it takes from when a node start the binary consensus algorithm by proposing a start value, until it receives a valid result back. Measuring latency is only useful for comparing algorithms as long as they are tested on the same system-setup. This is because latency is dependent on many different factors, for example, the computer or network setup. Therefore, we look at other parameters, such as idle time and number of messages in an attempt to trace the dominate factors.

The second measurement we use is the idle time i.e, the time spent waiting on the corresponding messages. This measurement is measured by computing the total time spent inside the repeat-until clause from when we start the binary consensus algorithm to the point we have a valid result. Lastly the number of rounds and the total number of messages is measured. Number of rounds simply count the number of rounds it take to reach consensus and number of messages count the number of messages a node receives/handles before reaching consensus. With these four criteria we are able to get a feel for how the different algorithms perform.

4.3 Experiments

One drawback of the algorithms is that the number of messages sent between the nodes can quickly become very large. Sometimes this results in nodes receiving a lot of duplicate messages that "hide" new messages. This could increase the latency since nodes cannot find the correct message. This problem is even worse when the number of nodes increases for the system. This is one aspect we think is necessary to examine and see how it affects the overall performance. In the experiment we conducted, we test the scalability by varying the number of nodes in the system. For this experiment the number of nodes are varying between 3 and 12.

In order to get a better overview of the evaluation, we divide it into two parts related to what type of binary consensus base is used (Ω or randomness).

Experiment 1: evaluation of random binary consensus (CC)

To make the evaluation easier, we conducted an experiment where all nodes where non-faulty, each node was proposed the same start value and the CC had the same seed to start with. This way, we got a result that is easily comparable between the different variations of the algorithm. For this experiment, algorithm 7 was ran 30 times each on all different system setting (where we varied the number of nodes between 3 and 12). The final measurement was calculated by taking the average of all 30 runs.

Experiment 2: evaluation of Ω binary consensus

This part of the evaluation was used for algorithms number 1-6. To make the eval-

uation easier, we conducted an experiment where all nodes where non-faulty, each node was proposed a random starting value (0 or 1) each time and the corresponding Ω failure detector was started before the binary consensus algorithm was started. Apart from that, each algorithm was ran 15 times each on all different system settings (where we varied the number of nodes between 3 and 12). From there, the final measurement was calculated by taking the average of the 15 runs.

4.4 Result of evaluation

We will look at the result of the evaluations of binary consensus using the common coin (CC) and binary consensus using Ω separately and then compare the best algorithm from both approaches.

4.4.1 Binary consensus using CC

We have a single version of the binary consensus algorithm that uses CC as assumption. It is the base algorithm seen in algorithm 4. The goal was to test how the algorithm will scale with the number of nodes in the system.

Expectations: When the scale of the system increases we can expect to see the number of messages, and thus also the latency, increase. This is due to the fact that when the system is larger, a node have to send messages to more nodes and more importantly, it needs to wait for more messages before continuing with the round. The round number however, is always expected to be at most four in this algorithm and should statistically stay between 1-4 rounds [78].

Observations: The result of the test is presented in figure 4.1. The first observation is to note the similarities between figures 4.1a and 4.1b. They are practically indistinguishable. This tells us that basically the whole latency of the algorithm will be dependent on how long the repeat-until-clause takes or how long we time a node spend waiting on messages.

We also note that the algorithm, for each measurement, advances in a zigzag pattern when the total number of nodes in the system increases. The explanation for this is that a majority of nodes need to cooperate in a round before continuing to the next one. This mean that if we have 6 nodes or 7 nodes in total in the system, we would for both cases only need 4 nodes to cooperate in order to get to a result. This could lead you too believe that the graph would advance in more of a stair shape where there is a plateau every other node. Instead we note that the uneven numbers actually have lower values than the smaller even numbers. This is due to the fact that the faster majority of nodes will decide a value before the slower nodes advance too far. The slower nodes will then receive a decide message before deciding themselves and will thus not need to work as long. When the system have an odd number of nodes, we have an extra slow node that bring average down.

The final thing to note is the inclination of the measurements. For these results

it seem like the graph is linear and with a quick calculation, we see that the latency is increasing by approximately 21 ms/node. Something to note is that, this slope would probably steepen if the round-trip-time was bigger. But more testing would be needed to see how much the round-trip-time affect the slope.

4.4.2 Binary consensus using Ω failure detector

For binary consensus that uses the Ω failure detector, we have six different algorithms. Two base algorithms that uses either the ◇MS-Pat assumption or the ◇T-SOURCE assumption. We also have the binary consensus algorithm that uses the hybrid Ω and lastly all the algorithms above, with the Look-Ahead technique added. To explain the results in a more clear way, this section will first look at the different variations and later look at how the Look-Ahead technique affected the algorithms.

Expectations: The goal of the experiment was, first and foremost, to see which one of the binary consensus variations is the fastest. We do this by running the same test on all the variations of the algorithm and then compare the latency. In advance, it is hard to guess which one of ◇MS-Pat and ◇T-SOURCE will be the fastest since it will depend on how fast they can find a common leader. However, we do hope that the hybrid version will be able to be faster than the two non-hybrid solutions. The reason for why this is not a matter of course is because there could be a scenario where one Ω failure detector always performs better than the other and thus make the second one redundant. The addition of the Look-Ahead technique to the algorithms is assumed to decrease the latency by reducing the number of messages received by each node in order to come to a consensus. The expectations of the algorithms when the system is scaled up with the number of total nodes in the system, is that each measurement should increase.

Observations: The results from the evaluation is presented in Figure 2.1. The first thing to note is that the results are rather incoherent and there is not as how the graph progresses. This could be because the algorithms was only ran 15 times each or just because of the more unpredictable nature of the Ω failure detector. When examining the graphs we note that the high point in the round value seen in 4.2d seem to be represented in all the other graphs as as a high value as well. From this we can assume that perhaps the incoherence of the graphs stems from the algorithms being more unpredictable. We, for example, can note the range between X= 8-12 for the rounds graph 4.2d, where the ◇*t*-SOURCE algorithm have high values which can also be seen represented in the message graph 4.2c for the same algorithm and the same range. In general, however, we actually see really low round values for all the Ω algorithms. Both algorithms no. 5 and no. 6 for example was able to decide in round 1 for each processor each time. This is especially spectacular since the binary consensus algorithm is not able to decide before the leader oracle provides a stable leader. If the round values are able to be 1 it means that, for this test the Ω failure detector was able to stabilize even before the binary consensus algorithm started.

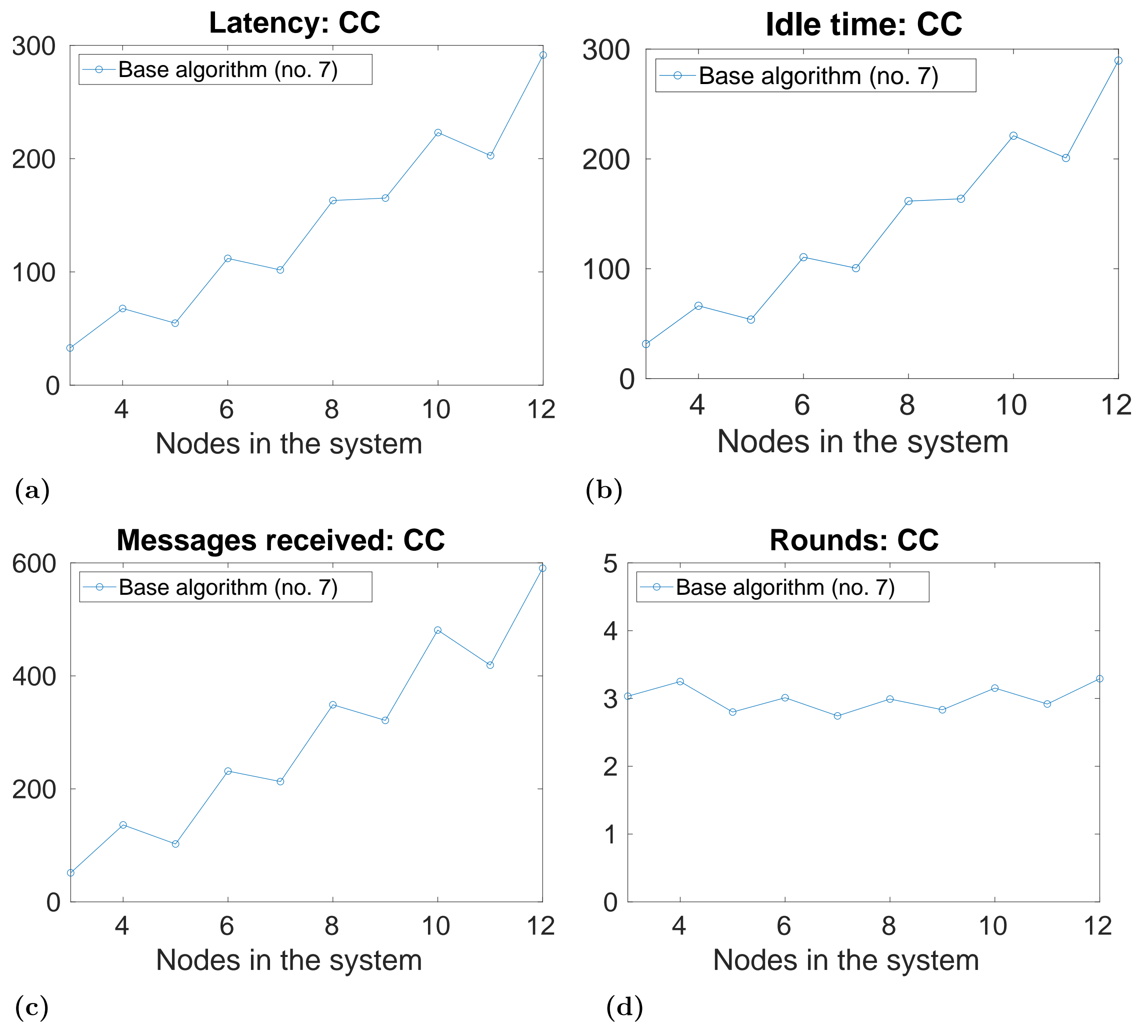

Figure 4.1: This figure show the results of the evaluation of the algorithm related to the binary consensus algorithm that use randomization and common coin (no. 7). The different sub-figures show the different criteria measured during the evaluation. Figure (a) shows the average latency in milliseconds for the algorithm. (b) shows the time spent waiting for messages in milliseconds. Figure (c) shows the average total number of messages received before coming to a consensus. Lastly (d) shows the average number of rounds for the algorithm.

For these algorithms, the prediction came true and we can see that in general both the improvements work. The hybrid solution is faster than the two base versions and it seems to originate from the fact that it could decide as early as round 1 and thus, not send as many messages, which is also reflected in the message measurement. Another thing to note however, is that the drop in messages received is actually larger than expected for the hybrid-version compared to the base-cases. We only assume that the messages should decrease when the number of rounds decreases, but, we can for example look at X=3 for both graph 4.2d and 4.2c. Here we note that the number rounds for all three algorithms no. 1, no. 3 and no. 5 is the same at 1 round, but the number of messages for the hybrid algorithm (no. 5), at 20 messages, is lower than the messages of the two base algorithms (no. 1 and 3), at 100 messages. This can probably be explained by the increase in work load the hybrid solution have compared to the base algorithm. The hybrid solution have at each processor three different algorithms running in parallel compared to the base version that have two algorithms running in parallel. This increase in work load affect the binary consensus algorithm because it does not get to run as much for the hybrid version. When the binary consensus algorithm does not run as much, it will not send as many duplication messages. We can also see evidence of this phenomenon in the latency graph 4.2a. At X = 3, we see that both algorithm no. 1 and no. 3 have the same value and no. 5 and no. 6 have the same value but a bit higher. Even when the hybrid-versions send less messages than the base-versions, the latency is larger due to the fact that it need to run an additional Ω failure detector in parallel.

Next we note that the solutions ◇MS-Pat and ◇T-SOURCE take turn in being the fastest in figure 4.2a, which was the criteria for the hybrid version to increase the performance of the algorithm in the first place.

Lastly, we can also take a look at figure 4.3 which shows the latency of the binary consensus algorithm that uses Ω failure detectors, with and without the Look-Ahead technique. Here we note that in general it seems like the Look-Ahead technique do decrease the overall latency for this specific implementation and evaluation.

4.4.3 Comparison between Ω and CC

The last thing we do in this section is to compare the different oracles used to solve binary consensus, CC and Ω. We do this by taking the best algorithm of both types and compare each measurement side by side.

Expectations: The initial hypothesis is that the binary consensus algorithm with Ω will be faster for the smaller systems but as the number of nodes in the system increases, the randomized binary consensus algorithm will be faster. The reason is that for a smaller system setting, it is easy for Ω to find a common leader and the binary consensus algorithm can then decide early. The randomized binary consensus algorithm on the other hand is bounded in that it will approximately take up to four rounds every time. When the system gets larger, however, the Ω failure detector

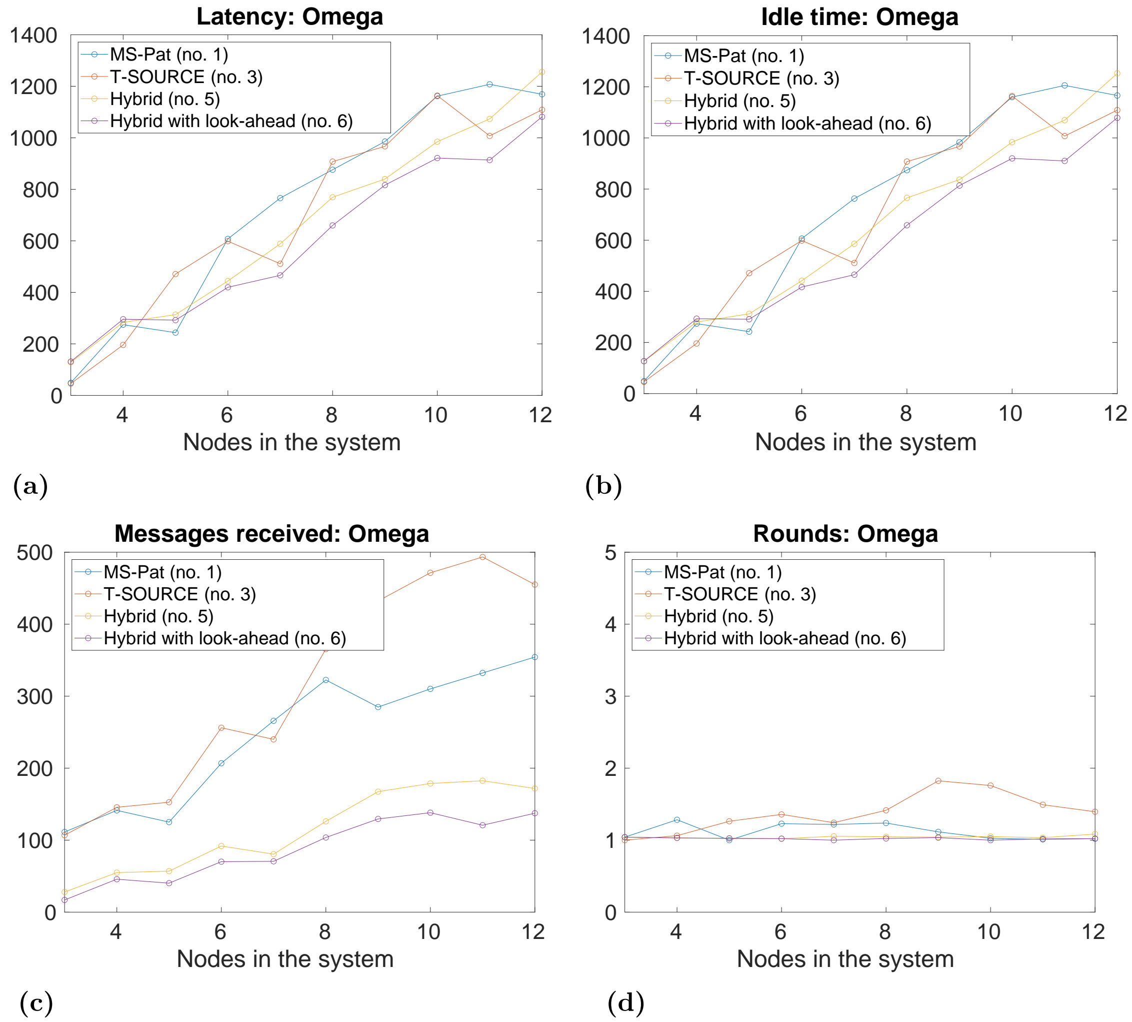

Figure 4.2: This figure shows the results of the evaluation of most of the algorithms related to the binary consensus algorithm that use failure detectors, specifically the Ω failure detector (no. 1, 3, 5, and 6). The different sub-figures shows the different criteria measured during the evaluation. Figure (a) shows the average latency in milliseconds for the algorithms. (b) shows the time spent waiting for messages in milliseconds. Figure (c) shows the average total number of messages received before coming to a consensus. Lastly (d) shows the average number of rounds for each algorithm.

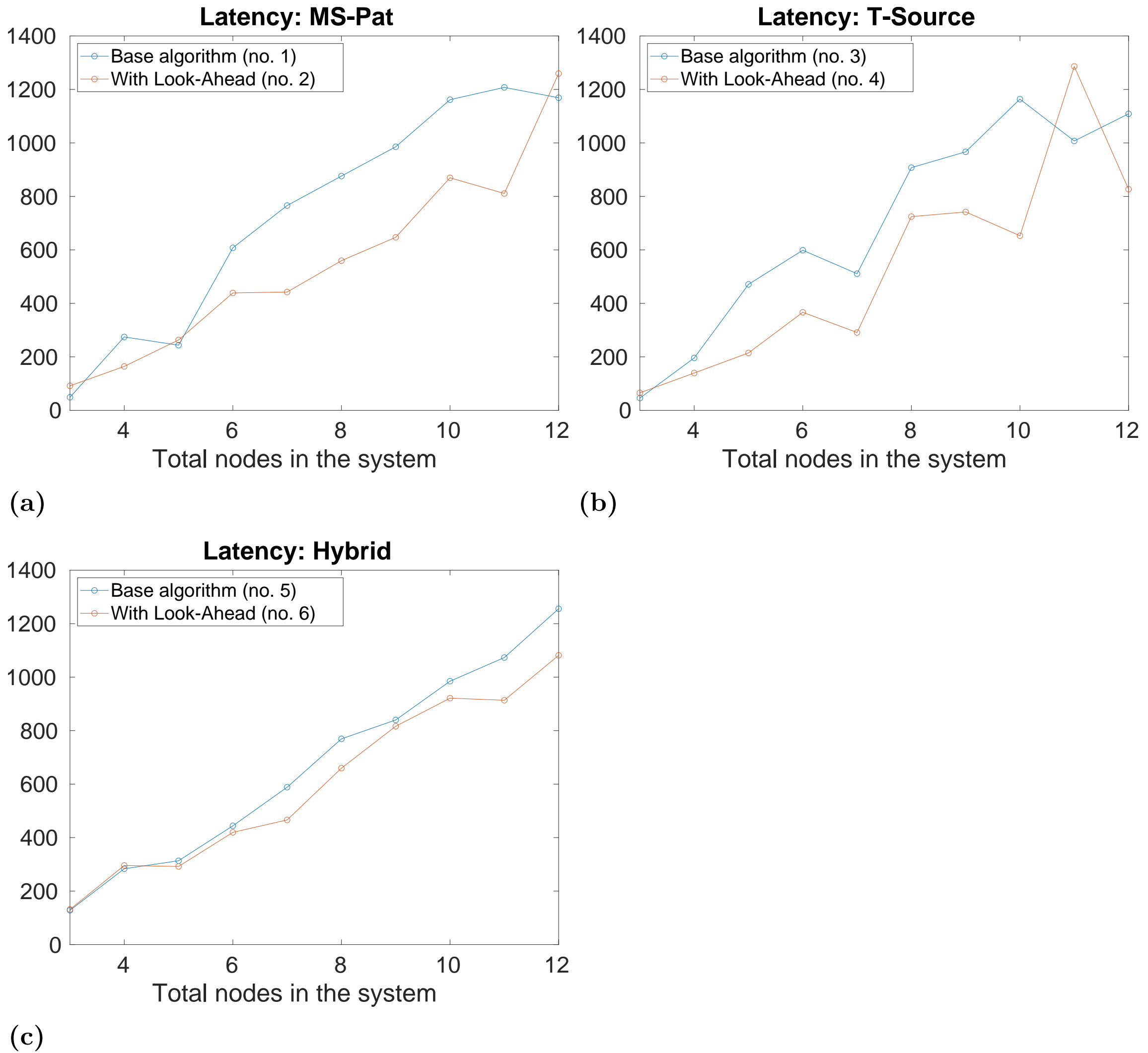

Figure 4.3: This figure shows the comparison of latency between the base algorithm and the version of the algorithm improved with the Look-Ahead technique of the algorithms that was based on the Ω failure detector (no. 1-6). The different sub-figures shows the different ways the Ω failure detector was implemented. Figure (a) shows the latency in millisecond of the two algorithms that used the ◇ MS_Pat assumption. Figure (b) shows the latency in millisecond of the two algorithms that used the ◇*t*-SOURCE assumption. Lastly (c) shows the latency in milliseconds of the two algorithms that used the hybrid version of Ω.

will have a harder time finding a common leader, and the binary consensus decision will be delayed.

Observations: The real result can be found in figure 4.4. Here we clearly see that randomized binary consensus is superior in comparison to the Ω solution, even for system settings with a small number of nodes. This is despite the Ω solution being able to decide in the first round every time, which is the optimal case. Not only that, the Ω solution also sends less messages on average than the randomized algorithm. However, the messages received measurement is not completely reliable to look at because the messages sent and received by the Ω failure detectors is not included in the value, only the messages sent and received by the binary consensus algorithm is measured.

So, how can the difference be so large? One explanation is that the Ω solution, especially the hybrid one, have many different variables to take into consideration when trying to optimize the outcome. For this evaluation, little time was spent trying to find the most optimal configuration before starting the test. The randomized solution basically only have a single variable that make a large difference on the latency. Another explanation could be that the randomized version only have a single algorithm running for each node and the algorithm itself only have a single phase. Meanwhile, the Ω solution have three different algorithms running in parallel.

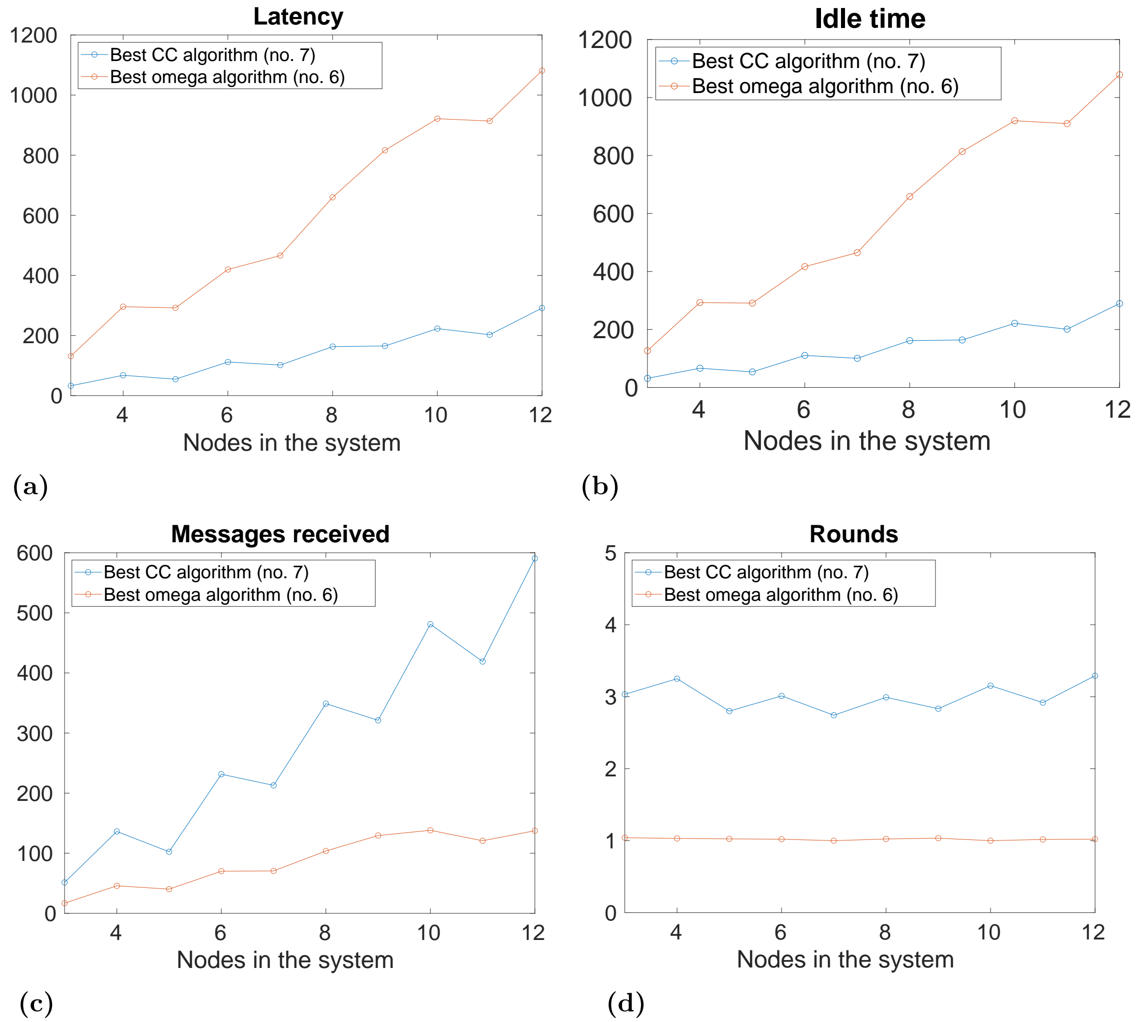

Figure 4.4: This figure show the evaluation of the binary consensus algorithm that used randomization (no. 7) and the the best binary consensus algorithm that used a failure detector (no. 6). Figure (a) shows the average latency in milliseconds for the algorithms. (b) show the time spent waiting for messages in milliseconds. Figure (c) show the average total number of messages received before coming to a consensus. Lastly (d) shows the average number of rounds for each algorithm.

5 Discussion and conclusion

Binary consensus, and particularly fault tolerant self-stabilizing binary consensus, is a complicated area to study. Embossed by impossibilities and their respective circumventions. Here we have explored two different ways of solving binary consensus in the following system model; a asynchronous message passing system with unreliable channels where at most a minority of nodes can fail-stop and arbitrary transient faults can happen. The two ways we solve binary consensus within this system is one using randomness and one using a eventual leader failure detector, both interesting and challenging in their own way. We have also used a technique to help improve the overall performance of the latter approach.

The goal of this project was to see if an binary consensus algorithm for such an extensive system and fault model can be efficient and useful for real-world applications. From the result section we can either confirm or deny if the algorithms was efficient enough. The conclusion we can draw is that both methods we used to solve binary consensus with do show promising results. Especially the randomized binary consensus algorithm. We can also conclude that both improvements added to the binary consensus algorithm that uses a Ω-failure detector seem to decrease the latency of the base-algorithms. The next step in evaluating the algorithms is to run the experiment in a real-world distributed system and see how the performance is in comparison. For this thesis we could only run the experiment on a local-setup and this will of course affect the latency by reducing the round-trip-time of messages. However, a local setup will probably also increase the latency as all nodes are forces to run in parallel on the same machine. We can probably assume that especially the binary consensus algorithm that used Ω was especially affected by the local setup since a single node runs 2-3 parallel thread by itself.

In order to answer the question of whether the algorithms are useful or not, a lot more evaluation is needed where the number of experiment must be increased and the testing environment should better representing a real-life system. One useful experiment could be to test how the performance of the algorithms change depending on the round-trip-time of the distributed system.

It would also be necessary to understand how the constants in correlation to the system setting affect the performance of the algorithms in order to find optimal performance of the algorithms. The implementations provided in this project can serve as a basis for a comparative study that, perhaps, can point out the best approach and suggest how to optimize the related parameters in order to get the best

performance for a specific system.

It is also important to note that there are still more binary consensus variations to try out. For example, we have the briefly mentioned binary consensus algorithm that uses an assumption about how messages are delivered. There also exist more types of failure detectors *e.g.,* $\diamond P$ eventual perfect, that also solve binary consensus. When it comes to improvements, for this thesis we only used the Look-Ahead technique, but there are also several, smaller improvements that are quite easy to implement as extensions to our implementation.

Bibliography

[1] Marcos K. Aguilera et al. "Communication-efficient leader election and consensus with limited link synchrony". In: *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing* 23 (2004), pp. 328–337. DOI: 10.1145/1011767.1011816.

[2] Marcos Kawazoe Aguilera and Sam Toueg. "Randomization and failure detection: A hybrid approach to solve consensus". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 1151 (1996), pp. 29–39. ISSN: 16113349. DOI: 10.1007/3-540-61769-8_3.

[3] Janhavi Baikerikar, Sunil Surve, and Sapna Prabhu. "Consensus Based Dynamic Load Balancing for a Network of Heterogeneous Workstations". In: *Advances in Computing, Communication and Control*. Ed. by Srija Unnikrishnan, Sunil Surve, and Deepak Bhoir. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 116–124. ISBN: 978-3-642-18440-6.

[4] Joffroy Beauquier, Thomas Hérault, and Elad Schiller. "Easy Stabilization with an Agent". In: *WSS*. Vol. 2194. Lecture Notes in Computer Science. Springer, 2001, pp. 35–50.

[5] Marco Canini et al. "A Self-Organizing Distributed and In-Band SDN Control Plane". In: *ICDCS*. IEEE Computer Society, 2017, pp. 2656–2657.

[6] Marco Canini et al. "Renaissance: A Self-Stabilizing Distributed SDN Control Plane". In: *ICDCS*. IEEE Computer Society, 2018, pp. 233–243.

[7] Marco Canini et al. "Renaissance: Self-Stabilizing Distributed SDN Control Plane". In: *CoRR* abs/1712.07697 (2017).

[8] António Casimiro, Emelie Ekenstedt, and Elad Michael Schiller. "Self-Stabilizing Manoeuvre Negotiation: The Case of Virtual Traffic Lights". In: *SRDS*. IEEE, 2019, pp. 354–356.

[9] Farah Habib Chanchary and Samiul Islam. "Strong Consensus in Cloud Computing". In: October (2012).

[10] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. "Weakest failure detector for solving Consensus". In: *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing* 43.4 (1992), pp. 147–158. DOI: 10.1145/135419.135451.

[11] *Consensus Algorithms in Blockchain*. URL: https://www.geeksforgeeks.org/consensus-algorithms-in-blockchain/. (accessed: 22.06.2021).

[12] Shlomi Dolev, Ronen I. Kat, and Elad Michael Schiller. "When Consensus Meets Self-stabilization". In: *OPODIS*. Vol. 4305. Lecture Notes in Computer Science. Springer, 2006, pp. 45–63.

[13] Shlomi Dolev, Ronen I. Kat, and Elad Michael Schiller. "When consensus meets self-stabilization". In: *J. Comput. Syst. Sci.* 76.8 (2010), pp. 884–900.

[14] Shlomi Dolev, Omri Liba, and Elad Michael Schiller. "Self-Stabilizing Byzantine Resilient Topology Discovery and Message Delivery". In: *CoRR* abs/1208.5620 (2012).

[15] Shlomi Dolev, Omri Liba, and Elad Michael Schiller. "Self-stabilizing Byzantine Resilient Topology Discovery and Message Delivery". In: *SSS*. Vol. 8255. Lecture Notes in Computer Science. Springer, 2013, pp. 351–353.

[16] Shlomi Dolev, Omri Liba, and Elad Michael Schiller. "Self-stabilizing Byzantine Resilient Topology Discovery and Message Delivery - (Extended Abstract)". In: *NETYS*. Vol. 7853. Lecture Notes in Computer Science. Springer, 2013, pp. 42–57.

[17] Shlomi Dolev, Thomas Petig, and Elad Michael Schiller. "Brief Announcement: Robust and Private Distributed Shared Atomic Memory in Message Passing Networks". In: *PODC*. ACM, 2015, pp. 311–313.

[18] Shlomi Dolev, Thomas Petig, and Elad Michael Schiller. "Self-Stabilizing and Private Distributed Shared Atomic Memory in Seldomly Fair Message Passing Networks". In: *CoRR* abs/1806.03498 (2018).

[19] Shlomi Dolev and Elad Schiller. "Communication Adaptive Self-Stabilizing Group Membership Service". In: *WSS*. Vol. 2194. Lecture Notes in Computer Science. Springer, 2001, pp. 82–97.

[20] Shlomi Dolev and Elad Schiller. "Communication Adaptive Self-Stabilizing Group Membership Service". In: *IEEE Trans. Parallel Distributed Syst.* 14.7 (2003), pp. 709–720.

[21] Shlomi Dolev and Elad Schiller. "Self-Stabilizing Group Communication in Directed Networks". In: *Self-Stabilizing Systems*. Vol. 2704. Lecture Notes in Computer Science. Springer, 2003, pp. 61–76.

[22] Shlomi Dolev and Elad Schiller. "Self-stabilizing group communication in directed networks". In: *Acta Informatica* 40.9 (2004), pp. 609–636.

[23] Shlomi Dolev, Elad Schiller, and Jennifer L. Welch. "Random walk for self-stabilitzing group communication in ad hoc networks". In: *PODC*. ACM, 2002, p. 259.

[24] Shlomi Dolev, Elad Schiller, and Jennifer L. Welch. "Random Walk for Self-Stabilizing Group Communication in Ad Hoc Networks". In: *IEEE Trans. Mob. Comput.* 5.7 (2006), pp. 893–905.

[25] Shlomi Dolev, Elad Schiller, and Jennifer L. Welch. "Random Walk for Self-Stabilizing Group Communication in Ad-Hoc Networks". In: *SRDS*. IEEE Computer Society, 2002, pp. 70–79.

[26] Shlomi Dolev et al. "Autonomous virtual mobile nodes". In: *DIALM-POMC*. ACM, 2005, pp. 62–69.

[27] Shlomi Dolev et al. "Autonomous virtual mobile nodes". In: *SPAA*. ACM, 2005, p. 215.

[28] Shlomi Dolev et al. "Brief announcement: virtual mobile nodes for mobile ad hoc networks". In: *PODC*. ACM, 2004, p. 385.

[29] Shlomi Dolev et al. "Practically Stabilizing Virtual Synchrony". In: *CoRR* abs/1502.05183 (2015).

[30] Shlomi Dolev et al. “Practically-self-stabilizing virtual synchrony”. In: *J. Comput. Syst. Sci.* 96 (2018), pp. 50–73.
[31] Shlomi Dolev et al. “Self-Stabilizing Automatic Repeat Request Algorithms for (Bounded Capacity, Omitting, Duplicating and non-FIFO) Computer Networks”. In: *CoRR* abs/2006.05901 (2020).
[32] Shlomi Dolev et al. “Self-stabilizing End-to-End Communication in (Bounded Capacity, Omitting, Duplicating and non-FIFO) Dynamic Networks - (Extended Abstract)”. In: *SSS*. Vol. 7596. Lecture Notes in Computer Science. Springer, 2012, pp. 133–147.
[33] Shlomi Dolev et al. “Self-stabilizing Reconfiguration”. In: *Middleware Posters and Demos*. ACM, 2016, pp. 13–14.
[34] Shlomi Dolev et al. “Self-stabilizing Reconfiguration”. In: *CoRR* abs/1606.00195 (2016).
[35] Shlomi Dolev et al. “Self-stabilizing Reconfiguration”. In: *NETYS*. Vol. 10299. Lecture Notes in Computer Science. 2017, pp. 51–68.
[36] Shlomi Dolev et al. “Self-stabilizing Virtual Synchrony”. In: *SSS*. Vol. 9212. Lecture Notes in Computer Science. Springer, 2015, pp. 248–264.
[37] Shlomi Dolev et al. “Strategies for repeated games with subsystem takeovers implementable by deterministic and self-stabilising automata”. In: *Int. J. Auton. Adapt. Commun. Syst.* 4.1 (2011), pp. 4–38.
[38] Shlomi Dolev et al. “Strategies for repeated games with subsystem takeovers: implementable by deterministic and self-stabilizing automata (extended abstract)”. In: *Autonomics*. ICST, 2008, p. 37.
[39] Shlomi Dolev et al. “Virtual Mobile Nodes for Mobile Ad Hoc Networks”. In: *DISC*. Vol. 3274. Lecture Notes in Computer Science. Springer, 2004, pp. 230–244.
[40] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. “Impossibility of Distributed Consensus with One Faulty Process”. In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382. ISSN: 1557735X. DOI: 10.1145/3149.214121.
[41] Felix C. Freiling, Rachid Guerraoui, and Petr Kuznetsov. “The failure detector abstraction”. In: *ACM Computing Surveys* 43.2 (2011). ISSN: 03600300. DOI: 10.1145/1883612.1883616.
[42] Roy Friedman, Achour Mostefaoui, and Michel Raynal. “Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems”. In: *IEEE Transactions on Dependable and Secure Computing* 2.1 (2005), pp. 46–56. ISSN: 15455971. DOI: 10.1109/TDSC.2005.13.
[43] Chryssis Georgiou, Oskar Lundström, and Elad Michael Schiller. “Self-Stabilizing Snapshot Objects for Asynchronous Fail-Prone Network Systems”. In: *CoRR* abs/1906.06420 (2019).
[44] Chryssis Georgiou, Oskar Lundström, and Elad Michael Schiller. “Self-Stabilizing Snapshot Objects for Asynchronous Failure-Prone Networked Systems”. In: *PODC*. ACM, 2019, pp. 209–211.
[45] Chryssis Georgiou, Oskar Lundström, and Elad Michael Schiller. “Self-stabilizing Snapshot Objects for Asynchronous Failure-Prone Networked Systems”. In:

NETYS. Vol. 11704. Lecture Notes in Computer Science. Springer, 2019, pp. 113–130.

[46] Chryssis Georgiou et al. "Loosely-self-stabilizing Byzantine-tolerant Binary Consensus for Signature-free Message-passing Systems". In: *CoRR* abs/2103.14649 (2021).

[47] Chryssis Georgiou et al. "Self-stabilization Overhead: A Case Study on Coded Atomic Storage". In: *NETYS*. Vol. 11704. Lecture Notes in Computer Science. Springer, 2019, pp. 131–147.

[48] Chryssis Georgiou et al. "Self-stabilization Overhead: an Experimental Case Study on Coded Atomic Storage". In: *CoRR* abs/1807.07901 (2018).

[49] Zacharias Georgiou et al. "A Self-stabilizing Control Plane for Fog Ecosystems". In: *UCC*. IEEE, 2020, pp. 13–22.

[50] Zacharias Georgiou et al. "A Self-stabilizing Control Plane for the Edge and Fog Ecosystems". In: *CoRR* abs/2011.02190 (2020).

[51] Jaap-Henk Hoepman et al. "Secure and Self-stabilizing Clock Synchronization in Sensor Networks". In: *SSS*. Vol. 4838. Lecture Notes in Computer Science. Springer, 2007, pp. 340–356.

[52] Jaap-Henk Hoepman et al. "Secure and self-stabilizing clock synchronization in sensor networks". In: *Theor. Comput. Sci.* 412.40 (2011), pp. 5631–5647.

[53] Daniel Karlberg and Daniel Kem. "Self-Stabilizing Emulation of State-Machine Replication". MA thesis. Gothenburg, Sweden: Computer Science and Engineering, Chalmers University of Technology, 2021.

[54] Olaf Landsiedel, Thomas Petig, and Elad Michael Schiller. "DecTDMA: A Decentralized-TDMA - With Link Quality Estimation for WSNs". In: *SSS*. Vol. 10083. Lecture Notes in Computer Science. 2016, pp. 231–247.

[55] Pierre Leone, Marina Papatriantafilou, and Elad Michael Schiller. "Relocation Analysis of Stabilizing MAC". In: *SSS*. Vol. 5873. Lecture Notes in Computer Science. Springer, 2009, pp. 791–792.

[56] Pierre Leone, Marina Papatriantafilou, and Elad Michael Schiller. "Relocation Analysis of Stabilizing MAC Algorithms for Large-Scale Mobile Ad Hoc Networks". In: *ALGOSENSORS*. Vol. 5804. Lecture Notes in Computer Science. Springer, 2009, pp. 203–217.

[57] Pierre Leone and Elad Schiller. "Self-Stabilizing TDMA Algorithms for Dynamic Wireless Ad Hoc Networks". In: *Int. J. Distributed Sens. Networks* 9 (2013).

[58] Pierre Leone and Elad Michael Schiller. "Interacting Urns Processes for Clustering of Large-Scale Networks of Tiny Artifacts". In: *Int. J. Distributed Sens. Networks* 6.1 (2010).

[59] Pierre Leone and Elad Michael Schiller. "Interacting urns processes: for clustering of large-scale networks of tiny artifacts". In: *SAC*. ACM, 2008, pp. 2046–2051.

[60] Pierre Leone and Elad Michael Schiller. "Self-Stabilizing TDMA Algorithms for Dynamic Wireless Ad-hoc Networks". In: *CoRR* abs/1210.3061 (2012).

[61] Pierre Leone and Elad Michael Schiller. "Self-stabilizing TDMA Algorithms for Dynamic Wireless Ad-Hoc Networks". In: *ALGOSENSORS*. Vol. 7718. Lecture Notes in Computer Science. Springer, 2012, pp. 105–107.

[62] Pierre Leone and Elad Michael Schiller. “Self-stabilizing TDMA Algorithms for Dynamic Wireless Ad-hoc Networks”. In: *SENSORNETS*. SciTePress, 2013, pp. 119–124.

[63] Pierre Leone et al. “Chameleon-MAC: Adaptive and Self-* Algorithms for Media Access Control in Mobile Ad Hoc Networks”. In: *SSS*. Vol. 6366. Lecture Notes in Computer Science. Springer, 2010, pp. 468–488.

[64] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. “Self-Stabilizing Indulgent Zero-degrading Binary Consensus”. In: *CoRR* abs/2010.05489 (2020).

[65] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. “Self-Stabilizing Indulgent Zero-degrading Binary Consensus”. In: *ACM International Conference Proceeding Series* (2021), pp. 106–115. DOI: 10.1145/3427796.3427836. arXiv: 2010.05489.

[66] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. “Self-Stabilizing Indulgent Zero-degrading Binary Consensus”. In: *ICDCN*. ACM, 2021, pp. 106–115.

[67] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. “Self-stabilizing Multivalued Consensus in Asynchronous Crash-prone Systems”. In: *CoRR* abs/2104.03129 (2021).

[68] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. “Self-Stabilizing Set-Constrained Delivery Broadcast (extended abstract)”. In: *ICDCS*. IEEE, 2020, pp. 617–627.

[69] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. “Self-stabilizing Uniform Reliable Broadcast”. In: *NETYS*. Vol. 12129. Lecture Notes in Computer Science. Springer, 2020, pp. 296–313.

[70] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. “Self-stabilizing Uniform Reliable Broadcast”. In: *CoRR* abs/2001.03244 (2020). arXiv: 2001.03244. URL: https://arxiv.org/abs/2001.03244.

[71] Achour Mostefaoui, Eric Mourgaya, and Michel Raynal. “Asynchronous Implementation of Failure Detectors”. In: *Proceedings of the International Conference on Dependable Systems and Networks* 00.c (2003), pp. 351–360. DOI: 10.1109/DSN.2003.1209946.

[72] Achour Mostefaoui, Michel Raynal, and Frédéric Tronel. “The best of both worlds: A hybrid approach to solve consensus”. In: *Proceedings of the 2002 International Conference on Dependable Systems and Networks* (2000), pp. 513–522. DOI: 10.1109/ICDSN.2000.857584.

[73] Achour Mostéfaoui and Michel Raynal. “Solving Consensus Using Chandra-Toueg’s Unreliable Failure Detectors: A General Quorum-Based Approach”. In: *Distributed Computing*. Ed. by Prasad Jayanti. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 49–63. ISBN: 978-3-540-48169-0.

[74] Thomas Petig, Elad Schiller, and Philippas Tsigas. “Self-stabilizing TDMA algorithms for wireless ad-hoc networks without external reference”. In: *Med-Hoc-Net*. IEEE, 2014, pp. 87–94.

[75] Thomas Petig, Elad Michael Schiller, and Philippas Tsigas. “Self-stabilizing TDMA Algorithms for Wireless Ad-Hoc Networks without External Reference”. In: *SSS*. Vol. 8255. Lecture Notes in Computer Science. Springer, 2013, pp. 354–356.

[76] Thomas Petig, Elad Michael Schiller, and Philippas Tsigas. “Self-stabilizing TDMA Algorithms for Wireless Ad-hoc Networks without External Reference”. In: *CoRR* abs/1308.6475 (2013).

[77] Michael O. Rabin. “Randomized Byzantine Generals.” In: *Annual Symposium on Foundations of Computer Science (Proceedings)* (1983), pp. 403–409. ISSN: 02725428. DOI: 10.1109/sfcs.1983.48.

[78] Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems*. 2018. ISBN: 9783319941400. DOI: 10.1007/978-3-319-94141-7.

[79] Iosif Salem and Elad Michael Schiller. “Practically-Self-Stabilizing Vector Clocks in the Absence of Execution Fairness”. In: *CoRR* abs/1712.08205 (2017).

[80] Iosif Salem and Elad Michael Schiller. “Practically-Self-stabilizing Vector Clocks in the Absence of Execution Fairness”. In: *NETYS*. Vol. 11028. Lecture Notes in Computer Science. Springer, 2018, pp. 318–333.

[81] Sun Wanlu. “On Clock Synchronization in Wireless Networks Using Parameter Estimation and Consensus Techniques”. In: (2013).

[82] Axel Wegener et al. “Hovering Data Clouds: A Decentralized and Self-organizing Information System”. In: *IWSOS/EuroNGI*. Vol. 4124. Lecture Notes in Computer Science. Springer, 2006, pp. 243–247.

[83] Weigang Wu et al. “Using asynchrony and zero degradation to speed up indulgent consensus protocols”. In: *Journal of Parallel and Distributed Computing* 68.7 (2008), pp. 984–996. ISSN: 07437315. DOI: 10.1016/j.jpdc.2008.02.007.