University of Cyprus

DEPARTMENT OF COMPUTER SCIENCE

Self-Stabilizing State Machine Replication in Static and Reconfigurable Asynchronous Message-Passing Systems

Ioannis Marcoullis

A dissertation submitted to the University of Cyprus

in partial fulfillment of the requirements

for the degree of Doctor of Philosophy

July, 2018

VALIDATION PAGE

Doctoral Candidate: Ioannis Marcoullis

Doctoral Dissertation Title: Self-Stabilizing State Machine Replication in Static and Reconfigurable Asynchronous Message-Passing Systems

The present Doctoral Dissertation was submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy at the Department of Computer Science and was approved on ***July 4th, 2018*** *by the members of the Examination Committee.*

Examination Committee:

Research Supervisor _____
Associate Professor Chryssis Georgiou

Committee Chair _____
Associate Professor Anna Philippou

Committee Member _____
Assistant Professor George Pallis

Committee Member _____
Professor Maria Potop-Butucaru

Committee Member _____
Professor Stefan Schmid

DECLARATION OF DOCTORAL CANDIDATE

The present Doctoral Dissertation was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy of the University of Cyprus. It is a product of original work of my own, unless otherwise mentioned through references, notes, or any other statements.

Ioannis Marcoullis

. . . . . .

Περίληψη

Η μέθοδος Αναπαραγωγής Μηχανής Καταστάσεων (ΑΜΚ) (state machine replication) στον κατανεμημένο υπολογισμό είναι θεμελιώδης στη χρήση πλεονασμού πόρων για την επίτευξη ανοχής σφαλμάτων. Στοχεύει στη συνεχή διατήρηση της συνέπειας πολλών αντιγράφων (replicas) ενός –πιθανώς δυναμικού– κατανεμημένου αντικειμένου, παρέχοντας έτσι αυξημένη διαθεσιμότητα (availability), προσπαθώντας όμως παράλληλα να περιορίσει, κατά το δυνατόν, τις επιπτώσεις στην επίδοση του συστήματος. Δύο καταξιωμένα μοντέλα παροχής ΑΜΚ είναι η ομοφωνία (consensus) και ο εικονικός συγχρονισμός (virtual synchrony). Αλγόριθμοι ΑΜΚ βάσει ομοφωνίας έχουν μελετηθεί και χρησιμοποιηθεί ευρέως. Πρόσφατα, η ΑΜΚ με ανοχή αυθαίρετων σφαλμάτων προτάθηκε ως εναλλακτική του μοντέλου Απόδειξης-Εργασίας (Proof-of-Work) για τον ορισμό της σειράς των "blocks" στα "blockchains". Κυριότερος λόγος είναι η δυνατότητα του πιο πάνω μοντέλου να υπερβαίνει το πρόβλημα της φθίνουσας επίδοσης που προκαλούν οι αυξανόμενες ανάγκες για συναλλαγές, αλλά και της υψηλής κατανάλωσης ενέργειας. Στον αντίποδα, η ΑΜΚ με βάση τον εικονικό συγχρονισμό απευθύνεται σήμερα προς υπηρεσίες νέφους (cloud services) με ανάγκες για υψηλές ταχύτητες.

Τα συστήματα που υλοποιούν ΑΜΚ παρέχουν εγγυήσεις επί τῇ βάσει παραδοχών όπως η ύπαρξη ανώτατου ορίου στον ρυθμό εισόδου/εξόδου των αντιγράφων (επεξεργαστών ή εξυπηρετητών) στο σύστημα, καθώς και των σφαλμάτων τα οποία μπορούν να επισυμβούν, οι αλάνθαστοι πιθανοτικοί έλεγχοι σφαλμάτων, αλλά και η υπόθεση αρχικοποίησης όλων των συστημικών μεταβλητών σε μια ορθή κατάσταση. Εάν όμως τα πιο πάνω, έστω και προσωρινά, παραβιαστούν από παροδικά σφάλματα, τότε αυτό μπορεί να αλλοιώσει την κατάσταση ενός ή περισσοτέρων αντιγράφων, περιλαμβανομένου του μετρητή του

προγράμματός τους. Συνεπώς, το σύστημα οδηγείται σε μια αυθαίρετη κατάσταση που το εξαναγκάζει σε διαρκώς εσφαλμένη υπηρεσία, ή σε μόνιμη διακοπή της υπηρεσίας, έως ότου υπάρξει ανθρώπινη διορθωτική παρέμβαση. Τα αυτοσταθεροποιούμενα συστήματα (self-stabilizing systems) ενισχύουν τα ανεκτικά σφαλμάτων συστήματα, επιτρέποντάς τους να ανακάμπτουν αυτόματα από παροδικά σφάλματα. Έτσι, και σε συνδυασμό με άλλες τεχνικές ανοχής σφαλμάτων, η αυτοσταθεροποίηση παρέχει μια περιεκτική και εύρωστη στρατηγική ανοχής σφαλμάτων και ανάκαμψης.

Η παρούσα διατριβή προτείνει αυτοσταθεροποιούμενες αλγοριθμικές λύσεις με αποδεδειγμένες εγγυήσεις σε προβλήματα που σχετίζονται με την AMK. Στο πλαίσιο αυτό, παρουσιάζουμε την πρώτη πρακτικά-αυτοσταθεροποιούμενη (practically-self-stabilizing) AMK που εδράζεται στο μοντέλο εικονικού συγχρονισμού για ένα στατικό σύνολο αντιγράφων που δύνανται να καταρρεύσουν. Έπειτα, εισαγάγουμε δυναμικότητα στο σύνολο των αντιγράφων, και παρέχουμε το πρώτο αυτοσταθεροποιούμενο σύστημα αναδιαμόρφωσης (reconfiguration) που μπορεί να οδηγήσει σε διαμορφοποιήσιμο AMK, βασισμένο είτε σε μοντέλο εικονικού συγχρονισμού, είτε ομοφωνίας. Τέλος αντιμετωπίζουμε το πιο δύσκολο μοντέλο σφαλμάτων, τα αυθαίρετα σφάλματα (arbitrary/Byzantine faults), και προτείνουμε έναν ανθεκτικό σε αυθαίρετα σφάλματα αυτοσταθεροποιούμενο αλγόριθμο AMK, που βασίζεται σε υλοποιήσιμους ανιχνευτές σφαλμάτων. Η διατριβή συνεισφέρει επίσης ορισμένα επιμέρους αποτελέσματα, όπως ένα αυτοσταθεροποιούμενο κατανεμημένο μετρητή που μπορεί να υλοποιήσει αυτοσταθεροποιούμενη κατανεμημένη κοινή μνήμη πολλαπλών γραφέων και αναγνωστών (multi-writer multi-reader shared memory emulation), καθώς και καινοτόμες τεχνικές σχεδιασμού και αποδείξεων. Σε γενική θεώρηση τα τρία αποτελέσματα καλύπτουν ένα σημαντικό κενό στον δρόμο για την υλοποίηση αυτοσταθεροποιούμενων συστημάτων "blockchain" βασισμένων σε AMK, αλλά και αυτοσταθεροποιούμενης διαμορφοποιήσιμης ανεκτικής σε αυθαίρετα σφάλματα AMK.

Abstract

State machine replication (SMR) in distributed computing is fundamental when employing redundancy of storage to facilitate fault-tolerance. It aims at maintaining the consistency of the state of several copies of a -possibly dynamic- distributed object, thus, providing increased availability. At the same time, it strives to reduce the impact on the system's performance. Research on cloud systems has significantly benefited from accumulated knowledge on SMR to quickly progress towards making cloud services efficient and reliable. Two established paradigms providing SMR are consensus and virtual synchrony (VS). Consensus-based SMR algorithms are widely-studied and used, and well-understood. Recently, Byzantine-tolerant SMR was proposed as an alternative to the Proof-of-Work (PoW) paradigm of block ordering in blockchains. This is because it is suggested to overcome the poor performance scalability of PoW and avoid the high energy consumptions. On the other hand, systems that provide the VS property are today directed towards high-speed cloud services.

Systems implementing SMR provide guarantees based on assumptions like bounded churn rates, bounded replica failures, failure-free probabilistic error detection mechanisms, or that system variables are started in a consistent state. If these are, even temporarily, violated due to the occurrence of transient faults, then this may corrupt the state of a single or multiple replicas, including its program counters. Subsequently, this leads the system to an arbitrary state, causing it to become possibly unavailable, unless there is human intervention. Self-stabilizing systems enhance existing fault tolerant systems to allow them to automatically recover from

transient failures. In this way, and combined with other fault-tolerance techniques, self-stabilization provides a comprehensive and robust fault-resilience and recovery strategy.

This thesis proposes self-stabilizing algorithmic solutions with proven guarantees to several SMR-related problems. In this framework, we initially present the first practically-self-stabilizing VS-based SMR for a static crash-prone replica set. We then introduce dynamicity in the membership of the replica set, and provide a modular self-stabilizing reconfiguration scheme that can lead to self-stabilizing reconfigurable SMR (based on either VS or consensus). We then address the more challenging failure model of Byzantine faults, and propose a malicious-tolerant self-stabilizing SMR algorithm based on implementable failure detectors. The thesis also bears several by-products such as a self-stabilizing counter that can be incremented in distributed fashion, and novel design and proof techniques. Considered together, the three results cover a significant gap towards achieving self-stabilizing versions of SMR-based blockchain frameworks or of reconfigurable Byzantine-tolerant SMRs.

Acknowledgments

I would like to express my indebtedness to my supervisor Chryssis Georgiou for his firm and inspiring guidance, indefatigable support and patient encouragement in all of the years of my doctoral studies. His inspiring example of persistence and excellence in research, combined with his integrity of character have positively impacted my development as a researcher and as a person. I would like to thank him for all the practical and financial support, for keeping his office door always open, and particularly, for the precious time that he invested in me, often taken from his personal/family time (for which I apologize).

My deep gratitude goes towards Elad Schiller for significantly enhancing my understanding of the area of self-stabilization, and generally my approach towards problem solving. His persistence in clear, rigorous and accurate statements of research results have greatly improved the presentation of this dissertation, but have also enriched my research and proof techniques over the past five years.

It was an honor collaborating with Shlomi Dolev and being introduced, first hand, to the principles of self-stabilization. His insightful advices have guided the presented research in many ways that I could never imagine. I would like to thank him for the meetings we had, which were always vibrant and joyful, and his remarks and comments, which were persistently to the point even at times when I could not realize this.

I am grateful that, over all the years of my studies, I had splendid labmates at the Foundations of Computing Systems and Theoretical Computer Science laboratory. I am thankful to all of them for being around at times of very intense effort, expressing

their support, and providing useful feedback. I hope that I have contributed to their efforts in a useful and constructive way as well.

I am thankful to the members of the faculty of the Computer Science Department with whom I collaborated over these years as part of my teaching assistance duties, and for always displaying a constructive high standard of professionalism in their duties. Similarly, I would also like to thank the Department's staff who willingly and patiently provided their important services whenever required.

I would like to take this opportunity to acknowledge the support provided by the University, especially through the two doctoral scholarships that proved very helpful in bringing this work to end.

Rightfully, my deepest gratitude and indebtedness goes to my parents, Christos and Andreani, and my sisters for their care and patience over these long years of my studies. Without their ethical and material support this would have been impossible.

Finally, I would like to thank all the people who have at various times and places taken interest and encouraged me to follow doctoral studies. To this end, I thank Fr. Andreas for his encouragement ensuing this degree, and all the people who have, for months, lifted my own duties and responsibilities to grant me the required time to complete this dissertation.

Thesis Contributions

This thesis is founded on the knowledge acquired by the author's involvement in the authorship of the following journal articles and conference papers:

Journal Articles

1. Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad M. Schiller, Practically-Self-Stabilizing Virtual Synchrony. *Journal of Computer and System Sciences*, Vol. 96, pp. 50–73, 2018.

Conference and Workshop Proceedings

2. Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad M. Schiller. Self-stabilizing Byzantine Tolerant Replicated State Machine Based on Failure Detectors. In *Proc. of the 2nd International Symposium on Cyber Security Cryptography and Machine Learning (CSCML 2018)*. pp. 84-100, Be'er Sheva, Israel, 2018.

3. Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad M. Schiller, Self-stabilizing Reconfiguration. In *Proc. of the 5th International Conference on Networked Systems (NETYS 2017)*. pp. 51-68, Marrakech, Morocco, 2017.

4. Ioannis Marcoullis, Self-stabilizing Middleware Services. In *Proc. of the Doctoral Symposium of the 17th International Middleware Conference*. Article 2, Trento, Italy, 2016.

5. Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad M. Schiller. Poster Abstract: Self-stabilizing Reconfiguration. In *Proc. of the Posters and Demos Session of the 17th International Middleware Conference*. pp. 13-14, Trento, Italy, 2016.

6. Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad M. Schiller. Self-stabilizing Virtual Synchrony. In *Proc. of the 17th International Symposium on Stabilization, Safety and Security of Distributed Systems (SSS 2015)*. pp. 248-264, Edmonton, Canada, 2015.

7. Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad M. Schiller. Brief Announcement: Self-stabilizing Virtual Synchrony. In *Proc. of the 29th International Symposium on Distributed Computing (DISC 2015)*. pp. 655-656, Tokyo, Japan, 2015.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

This thesis provides algorithmic solutions to problems in distributed computing that employ the state machine replication approach and are proved to recover to their designed behavior from any deviant system state. The solutions consider asynchronous static and dynamic message-passing systems under various failure models.

1.1 Motivation

Distributed computing, distributed data storage and more recently, the emergence of cloud computing, has rendered *fault-tolerance* a vital and highly desired system property. System failures, let them be power failures, hardware failures, communication interruptions or DoS attacks, can have such adverse repercussions on industries, services, and governments, that the commodity of fault-tolerance has major day-to-day gains. To this end, introducing *redundancy*, namely keeping multiple copies of a distributed object, is a standard technique that renders systems more robust by masking replica (or *processor*) failures and providing availability and performance in the case of queries. Redundancy, though, gives rise to an important problem; that of *consistency*. How can one retain consistency of state in every single replica when the distirbuted object is being changed possibly concurrently? The user of a distributed system must be able to receive a consistent, up-to-date view of the distributed object it is querying or processing.

Replication [1] is a well known and widely studied paradigm in distributed computing for creating multiple copies of a possibly dynamic object. The aim is to ensure that redundancy is leveraged towards masking failures and providing availability,

but not at the expense of the reliability of data, and also attempting to preserve performance during replica updates [2]. There are several approaches, and, depending on the task, one may choose to relax the need for strong consistency, for example, to achieve better performance and availability [3].

Several methods have been suggested over the years to retain the consistency of replicated objects. Transactional replication is a well-studied paradigm [4] on which many such replication services are built especially for carrying out operations on databases. *State machine replication* (SMR) [5,6] emulates the state transitions of one replica—possibly of a leader also known as primary or coordinator—in every other replica in the system through the execution of the same commands, with the same input, on the same state variables, and in the same order. Two of the standard and well-established techniques to implement SMR are *(i) consensus*-based techniques (e.g., Paxos [7]), and *(ii)* via Group Communication Systems (GCS) that implement the *virtual synchrony* (VS) paradigm [8–10].

Nowadays, consensus-based *Byzantine-fault-tolerant* (BFT) SMR is a hot topic because of its use in implementing distributed ledgers, a form of which is the blockchain [11–14]. On the other hand, the virtual synchrony paradigm can be found in the structure of several existing replication systems, and recent implementations of such systems are directed towards high-speed cloud services [15,16]. In general, BFT is a highly researched topic, since every large scale network will at some point (in many cases continuously) experience more severe failures than crashes. This may be due to intended or unintended malicious behavior.

Systems that facilitate replication are often found struggling with the changes in the membership of the replica set, which is often referred to as the *configuration*. The environments in which they function tend to be dynamic with processors joining and departing either graciously or by crashing. As time passes, the configuration composition perishes and application support becomes problematic. For long-lived systems, it is imperative to have a *reconfiguration* mechanism to introduce new replicas to the configuration and exclude departed or inefficient ones.

Fault-tolerant systems provide guarantees, given that some assumptions hold. It is possible that in some instances system assumptions are violated. For example, a system might be tolerant to a minority of failures of its processing entities, or to less than a third of those to exhibit malicious behavior. This is something that cannot be actually controlled. The same holds for assumptions about bounded churn rates,

i.e., bounded rates of processor joins and leaves/crashes. A common assumption of many reconfiguration services is that a majority or quorum[1] of the configuration is never lost [17, 18]. Another example is systems that offer guarantees *with high probability*, e.g., error detection. While such assumptions may be sufficient for a long period of a system's lifetime, some rare violations cannot be excluded, and these may have detrimental effects.

For example, a soft error (some accidental bit-flip) may force a counter to acquire its maximal value, and thus drive the system to either non-progress or to a permanent violation of the system's safety properties. A corrupt program counter or program variable can bring the system to an arbitrary state from which it cannot recover, since it was not anticipated by the system's designers. The system remains useless, requiring human intervention to recover and personnel to be always on-call.

Self-stabilizing systems [19, 20] are designed to automatically recover the system back to its working state and desired behavior. Such systems have a comprehensive approach towards faulty states that usually system designers consider as impossible to reach. In this way, self-stabilizing systems guarantee *convergence* to a legitimate system state starting from any possible system state, and *closure* when this legitimate state has been reached, and until the guarantees of the system are violated again.

Impressively, this approach can even cope with the almost universal designers' assumption that the system and its variables start in a consistent initial state. In general, automatic recovery reduces the cost of recovery, and possibly the off-time, making systems more available, and provides added-value as far as their maintainability is concerned. The self-stabilization research community has produced many results for a plethora of problems that are basic to distributed computing. State machine replication, as might be expected, has drawn the attention of many members of the self-stabilization community that aim to enhance existing solutions with the stabilization property.

An overview of the current state of the art, as Section 1.2 and to a greater extend Section 2 present, reveals several challenging open questions on fault-tolerant self-stabilizing replication services. Contrary to consensus, virtual synchrony has not up to this day received the attention of the self-stabilizing community, neither have important elements enabling dynamic participation in replication services such as the

[1]Sets of pairwise interconnected servers (e.g., majority of servers form a quorum system).

reconfiguration service. Moreover, while the use of implementable failure detectors is a well known technique to circumvent impossibility results that otherwise require stronger asynchrony or non-determinism, it is not obvious that they have been used in the types of problems that we consider. This thesis seeks to address these problems while moving in the direction of a deployable self-stabilizing, reconfigurable, fault-tolerant SMR system; desirably a Byzantine-tolerant one.

1.2 Prior Work

In this section we present some background knowledge of the topics and a selection of important (for this thesis) related works. This serves as a prelude to presenting the thesis's contributions.

Self-stabilization. A *self-stabilizing* algorithm is defined as an algorithm, which, started from any initial arbitrary state, *automatically* converges to the system's desired behavior, i.e., to a legal system state, within bounded time [6,19,21,22]. Edsger Dijkstra was the first to identify the self-stabilization property for distributed algorithms while solving the mutual exclusion task [20]. Leslie Lamport [23] commenting on Dijkstra's paper, held it to be Dijkstra's "most brilliant work", and "a milestone in work on fault tolerance", as well as a "very fertile field for research", a characterization that is today justified given the mounting number of published research papers on self-stabilization (including an annual conference on self-stabilization).

For asynchronous message-passing systems, it may be proved impossible to bound the stabilization time (as per the traditional self-stabilization definition). This is because predictions on *when* stale information will appear at a given processor to cause a safety violation are impossible to make, e.g., they are dependent on asynchrony. This motivates the study of weaker forms of stabilization [24]. Although we cannot bound the stabilization time, we can leverage upon the possibility to bound the number of possible safety violations. This approach is called pseudo-stabilization [25]. A similar, but more recent, approach is *practically-self-stabilization* [24,26–28], which is more inclusive in terms of the problems that it can tackle. In particular, it requires a bounded number of safety violations in a number of computation steps that is big enough to be considered infinite. We elaborate, further motivate, and more formally define the notion in Section 4, but for a more

complete account see [24] [2].

A necessary element to quantify the contributions of this work is the metric of an *asynchronous round*. Most of the guarantees of our self-stabilization results are using this metric. An asynchronous round for an algorithm $\mathcal{A}$ includes the complete execution of $\mathcal{A}$ from every processor in the system from the first line to the last, and the receipt of all the messages that every such processor sent by other processors. It is intuitive that faster processors may complete many rounds, but the metric considers the last processor to achieve the propagation of its information to every other processor. Note that for the case where there are failures, only correct processors are considered in the definition of the asynchronous rounds.

Practically-Self-Stabilizing Counters. Many systems like the ones performing replication (e.g., GCSs requiring group identifiers, of Paxos implementations requiring ballot numbers) assume access to an infinite (unbounded) counter. Practically, no integer counter can be implemented as unbounded, since hardware has finite capabilities. A *practically infinite* counter implemented as a τ-bit counter (e.g., 64-bit with $\tau = 64$) is not truly infinite, but it is large enough to provide counters for the lifetime of most conceivable systems when started at 0. Indeed, it is not difficult to verify that a 2^{64}-bit counter incremented per nanosecond can last for around 500 years, essentially an infinity for most of today's running systems. Nevertheless, transient faults can corrupt the counter to spontaneously attain its maximal value.

The bounded labeling scheme and the use of practically unbounded sequence numbers proposed by Alon et al. [27], allow the creation of practically-self-stabilizing bounded-sized solutions to the never-exhausted counter problem in the restricted case of a single writer. In [26], a practically-self-stabilizing version of Paxos was developed, which led to a practically-self-stabilizing consensus-based SMR implementation. To this end, they extended the labeling scheme of [27] to allow for multiple counter writers, since unbounded counters are required for ballot numbers. Extracting this scheme for other uses does not seem intuitive. We further detail the scheme in Section 4.3.

[2] This paper was only very recently accepted at the NETYS 2018 conference, where it received the Best Paper Award. This indicates an appreciation of the community to this notion as a potentially fruitful research direction in the area of stabilizing algorithms. The paper makes considerable use of the extended labeling scheme presented in this thesis and detailed in Section 4.

Virtual Synchrony. *Virtual Synchrony* (VS) is an important property provided by several Group Communication Systems (GCSs) that has proved to be valuable in the scope of fault-tolerant distributed systems where communicating processors are organized in groups with changing membership [8,29,30]. During the computation, groups change allowing an outside observer to track the history (and order) of the groups, as well as the messages exchanged within each group.

The VS property guarantees that any two processors that both participate in two consecutive such groups, should deliver the same messages in their respective group. Group communication systems that realize the VS abstraction provide services, such as *group membership* and *reliable group multicast*. The group membership service is responsible for providing the current *group view* of the recently live and connected group members, i.e., a processor set and a unique *view identifier*, which is a sequence number of the view installation. The group multicast service facilitates reliable delivery of messages (all-or-none delivers them), although the delivery order may be specified by the service. Thus, a virtually synchronous multicast is one which guarantees delivery of a message within the view it was sent in, unless the sender crashes [31, p.350]. We will mainly be referring to reliable totally-ordered delivery (also known as atomic message delivery) that requires all non-crashed processors of a view to deliver the same messages to the upper layers in the same order. This is necessary to facilitate SMR, albeit it faces the restrictions of the FLP result when implemented in asynchrony in the presence of failures.

Shared Memory Emulation. The shared memory primitive considers a shared object called "register", which is accessible through the operations read and write [32]. Concurrent reads and writes give rise to the challenge of preserving data consistency, although the primitive is simpler to conjecture about in relation to the message-passing one. Leslie Lamport [33] defined several notions of consistency (like safety and regularity), but the strictest one is *atomicity*. We only refer to this notion of atomicity as this is compatible to the guarantees that our services provide. The shared memory model may be emulated on a message-passing system through algorithms like the seminal atomic shared memory emulation protocol of Attiya, Bar-Noy and Dolev (known as ABD) [32]. This tolerates f crash failures that are up to a minority of the whole set, i.e., $f < n/2$. Here, a single processor is the *writer* that can write a value to the register. An extension by [34] allows for multiple writers with an extra

communication round for the writer. The service can be run in a static environment on a fixed set of processors, but also in a dynamic environment using a reconfiguration service to provide a set of reliable processors to act as service providers. We see this in the sequel.

Reconfiguration. Dynamic environments in which the participating processors (servers) that form the configuration join and depart continuously are always a challenging problem to every such system. Reconfiguration is a fundamental and indispensable component to any large scale system that provides long-lived services. Existing solutions for providing reconfiguration in dynamic systems, such as [17] and [18], do not consider transient faults and self-stabilization, because their correctness proofs (implicitly) depend on a coherent start [35], and also assume that crash failures can never prevent the quorum configuration to facilitate configuration updates, i.e., the quorum system never collapses before a reconfiguration completes. They also often use unbounded counters for ordering consensus messages, or for shared memory emulation and by that facilitate configuration updates, e.g., [17]. As we saw before, such counters can occasionally be subject to corruption with adverse results.

Byzantine Fault Tolerance. The most severe failure model of distributed computing is the Byzantine one [36], in which some processors may act arbitrarily by not following the protocol. This model abstracts the intention of the faulty processor, by including all such errors, e.g., malicious agent actions, a processor with a virus, or a communication fault in some processor, causing packet corruption of outgoing data [31]. State machine replication in the presence of Byzantine processors is a long studied problem [37, 38]. A milestone of this research line is Lamport et al.'s result that agreement is achievable only if faulty processors constitute less than a third of the processor set [36].

A seminal paper on BFT replication is Practical Byzantine Fault Tolerance (PBFT) by Castro and Liskov [38]. It achieves Byzantine-tolerant replication that is optimal in the number of Byzantine processors, i.e., $n = 3f + 1$. The service requires a consistent initial state, in which processors have a common integer *view*, and an unbounded local memory, although garbage collection procedures are well-defined. The processor whose identifier coincides with the view is the primary that coordinates the

view. It is responsible to assign sequence numbers to client requests, so that the other servers execute these in a common order, thus achieving SMR. Cryptography is used to ensure the authenticity of client requests sent by the (possibly malicious) primary to the other processors. It is also used to verify the authenticity of the other servers' messages. The process is then a three-phase protocol to establish that the sequence number assigned by the primary will be accepted by $2f + 1$ processors. This ensures that at least $f + 1$ correct processors accepted this, which is a majority of correct processors, and ensures that eventually every other correct processor will attain this request with the same request number.

Liveness is maintained using timeouts to check whether the primary is progressing the state machine. If this is not the case (although detection may be imperfect), a view change procedure commences. Correct processors in a view v await from the new primary (who has the identifier equal to $v + 1 \mod n$) to send a set of pending requests and a stable checkpoint of previous executed requests. If the other processors accept this, based on the encrypted-proved messages sent, they proceed to the next view. Otherwise, the processor with the next identifier will be expected to send this information and become the primary. We later review several optimization done to the above basic protocol, or other approaches. Nevertheless, PBFT remains a landmark in the research of BFT replication.

The only work available on self-stabilizing Byzantine-Fault-Tolerant SMR is by Binun et al. [39] that assumes a semi-synchronous setting and employs a self-stabilizing byzantine-tolerant clock synchronization algorithm [40] to enforce a new instance of Byzantine agreement upon every clock pulse. In particular, upon a clock pulse the system initiates $n + 1$ instances of a leaderless version of BFT for each processor to obtain a common vector of message histories. This defines a sequence of executions.

1.3 Contributions

This thesis contributes three main novel results by providing algorithmic self-stabilizing solutions to significant problems in distributed computing. It initially provides the first practically-self-stabilizing SMR service based on virtual Synchrony, which runs on a static set of crash-prone processors. It proceeds to provide the first self-stabilizing reconfiguration scheme that allows services such as the virtually synchronous SMR to run on a dynamic membership. The third work is the first

failure-detector-based Byzantine-Fault-Tolerant replication, that is essentially a self-stabilizing version of the seminal paper on Practical Byzantine Fault Tolerance [38]. We proceed to detail the contributions, without neglecting that these are not confined to the main results themselves. Indeed, en route to presenting the results, we also expose important by-products of this work that are possibly of independent interest and of general use.

Practically-self-stabilizing virtually synchronous SMR. The virtually synchronous SMR scheme is built with the stabilization notion of practically-self-stabilization that provides guarantees on the number of possible safety violations that an asynchronous system may experience starting in an arbitrary state. We consider a system with a fixed set of n crash-prone processing entities called *processors*.

The system employs a practically-self-stabilizing counter increment algorithm that is an extension of the one of Alon et al. [27] that facilitates the increment of the distributed counter by any processor in the system and not just the writer. Extending the scheme is not a straight-forward exercise. This is because stale messages in the communication links from a corrupt initial state may cause a phenomenon of endless cyclic adoptions of the crashed processors' counters. There is need to clean the system from this information debris before converging to a global maximal view. We prove that the system can clean corrupt labels with $O(n^3)$ violations of safety, which is a rather extreme case scenario. The space complexity (i.e., memory usage) in labels is also $O(n^3)$. The extended scheme proves its value by being modular and of general use as a practically-self-stabilizing black-box tool. It also lends itself easily to a straight-forward implementation of a Multi-writer Multi-reader (MWMR) shared memory emulation by the simple attachment of a value to the maximal counter. The rest is merely applying the multiple-writer version [34] of the ABD algorithm [32].

The SMR algorithm is constructed around a reliable multicast service realizing the virtual synchrony property. Data links, via packet exchange, implement a token passing abstraction used to implement a heartbeat, and in extent, a failure detector to provide the membership of the current multicast processor group (also known as the *view*). This is the (Θ)–failure detector presented for the self-stabilizing Paxos scheme [26]. The system is coordinator-based with the coordinator being the processor that draws the highest counter. The coordinator is "reused" and is the one to change the view if it detects changes to the composition of the current view, by

incrementing the counter and using it as the new view identifier. The reuse of the coordinator over several multicast rounds proves efficient in relation to consensus-based approaches. We prove our algorithm to be practically-self-stabilizing and require at most $O(n)$ forced view creations, i.e., safety violations, in order to guarantee the existence of a non-changing non-faulty coordinator. Such a coordinator is defined with respect to our failure detector with the liveness assumption that less than half of the processors suspect this processor in a computation that is practically infinite. This work is detailed in Chapter 4, and parts of it appear in [41–43].

Self-stabilizing Reconfiguration service. Departing from the previous system setting, we consider a dynamically changing set of crash-prone processors, of which at most N are live and connected at any time in the computation. Of the N participants, some or all constitute the set of configuration members. A reconfiguration service enhances the longevity of any application running on the configuration. As such, it is not confined to the application of replication, although, as we detail in Section 2, reconfiguration is mostly related to replication and shared memory emulation. The reconfiguration service proposed is the first, to our knowledge, self-stabilizing reconfiguration service. As such, it can recover from more adverse situations like the collapse of the configuration quorum system (e.g., of the majority), or from conflicts of the system's participants about the current configuration. The system can even bootstrap in cases where the configuration known by processors in the system's initial state is completely strange to the system's current membership.

The service comprises three modules. These grant to the application considerable control over customized configuration and reconfiguration decisions. In particular, the system is composed of a lower layer algorithm responsible to monitor the common knowledge about the current configuration. To this end, it either follows a *brute-force* reconfiguration technique that is a hard reset of the configuration, or a *delicate* form of reconfiguration. While brute reconfiguration allows every processor, even joining ones to enter the configuration, delicate reconfiguration only allows the application participants to decide and become members of the configuration. Both involve the convergence to a common processor set provided by a proposed eventually perfect failure detector, although the delicate form has the weaker requirement that only joined participants need to agree on the configuration membership. We use an enhanced version of the (Θ)-failure detector to get the (N, Θ)–failure detector

(an approach suggested in [44, 45]).

In the case of conflicting configuration knowledge, the lower layer is proved to self-stabilize and converge back to a state with a common configuration within $O(N)$ asynchronous rounds. The design and proof of the algorithm is not trivial, and employs an automaton-based lockstep progression of configuration convergence and installation.

The upper layer sits on top of the lower layer. and is responsible to check the well-being of the configuration and the existence of a majority or in general some quorum of processors of the configuration. It has a black box evaluation function that captures the general view that the question of *when* a reconfiguration is initiated is an application-based decision [35]. Self-stabilization is proved to take place within an additional $O(1)$ asynchronous rounds after the lower layer has stabilized. The technical difficulty lies in keeping the upper layer modular, yet unable to hinder the progress of a pending reconfiguration at the lower layer.

The joining mechanism completes the system and is the access point of joining processors to the application. It ensures that their local state is cleaned from any pre-existing stale information that may pollute the system upon induction. The approach is again application-based, since the joining mechanism gives to the application the right to allow or deny access. In this way one can control the churn (and thus the predefined system upper bound N). This work is detailed in Chapter 5, and preliminary versions appear in [43, 46, 47].

Self-stabilizing Byzantine-Fault-Tolerance. Looking towards more severe forms of failures in the SMR problem, a natural first step is to provide self-stabilizing Byzantine-fault-tolerant replication. Following the seminal paper by Castro and Liskov [38] on Practical Byzantine Fault Tolerance (PBFT), we build a self-stabilizing version. This guarantees convergence and retains the safety of the replication task in asynchrony, with progress being provided under given liveness assumptions. The solution is detailed for a fixed processor set of $n = 5f + 1$ processors where f may act maliciously at any time and not follow the system's protocol. We later explain how our solution works for optimal resilience, i.e., for $n = 3f + 1$. The scheme is composed of three modules.

Following the PBFT approach of using the view to define the primary processor, our first module "View Establishment" has the task to ensure that correct processors

have a consistent view that cannot be overthrown by malicious behavior. This is the most critical part, since, having a primary, correct processors can then converge to a single replica state. Managing convergence to a consistent view in the presence of Byzantine processors injecting arbitrary messages, and in the existence of other stale information in local states and communication channels is very demanding, and it is impossible without a series of assumptions [48–51]. To this respect, we present an automaton-based solution where convergence requires a fragment of the computation to be free of failures (even under this constraint, view establishment is still very challenging as one infers from Section 6.3). The automaton-based solution is similar to the one of the self-stabilizing reconfiguration work, but here it is presented in a modular way, in an effort to demonstrate that this automaton coordinating approach is applicable to other problems beyond the one considered. In Section 6.6, we relax this constraint by introducing a novel *event-driven* (unreliable) failure detector that can be tuned to ensure (in all reasonable executions) that enough responses from non-byzantine processors are received.

The second module offers the replication service following the three-phase protocol of PBFT with the following differences. We provide bounds on the message queues as this is vital for self-stabilization to be possible. Also, as we prefer to use information theoretically secure schemes, rather than computationally cryptographic secure schemes based on message signing [52], we require that clients contact all replicas. The primary is still the one to decide the order, but the replicas, through a self stabilizing all-to-all exchange procedure, validate the requests suggested to be processed by the primary. In the existence of a primary, the replication service requires $O(n)$ asynchronous rounds to converge to a unique replica state.

The third component is the primary monitoring module that ensures the liveness of the system by checking that the primary is making progress by ordering requests. We substitute the timer-based approach of Liskov with a suitable failure detector. In particular, we follow the approach of Baldoni et al. [53], to propose a self-stabilizing implementation of a failure detector that checks both the *responsiveness* of the replicas (including the primary), and whether the primary is progressing the state machine. This module stabilizes from an arbitrary initial state within $O(n)$ asynchronous rounds, although after a view exists, it only needs $O(1)$ rounds.

Diverging from the approach of the self-stabilizing BFT of [39], we do not use clock synchronization and timeouts, but rather, we base our solution on the self-

stabilizing failure detector and automaton-based coordination technique mentioned above. This approach encapsulates weaker synchronization guarantees than [39]. In view of [11], this result is an important step towards realizing self-stabilizing BFT-based infrastructure for blockchain systems. This work is detailed in Chapter 6 and a preliminary version appears in [54].

As a general contribution, our systems, when compared with corresponding non-stabilizing services, require bounded local memory and bounded-sized messages. Additionally, they do not require a consistent initial state. These properties are innate to stabilizing protocols.

1.4 Document Structure

The remaining parts of this thesis are structured as follows.

In Chapter 2 we study prior work and related literature.

In Chapter 3 we define the system settings for the different problems that we tackle, the self-stabilization notions that appear, and the metrics that we use. Given the diversity of the tasks that we tackle within the vicinity of SMR, we defer discussion of some system-specific details to the dedicated chapters. We also keep task definitions to their respective chapters.

Chapter 4 presents the practically-self-stabilizing virtually synchronous SMR scheme. In particular, we first present the practically-self-stabilizing version of the labeling scheme and counter increment counter with their correctness proofs (Section 4.3), and then detail the virtual synchrony algorithm (Section 4.4).

Chapter 5 considers the self-stabilizing reconfiguration scheme, the upper layer under the name of Reconfiguration Stability and Assurance (Section 5.3), the upper layer under the name of Reconfiguration Management (Section 5.4) and the joining mechanism (Section 5.5).

The Self-stabilizing Byzantine Fault Tolerance service in Chapter 6 is composed of the View Establishment module (Section 6.3), the Replication module (Section 6.4) and the Primary Monitoring module (Section 6.5).

We conclude with Chapter 7 where we overview the thesis work, and discuss future research directions of the presented line of work.

Chapter 2

Related Work

We overview related work in the research areas considered by this thesis. Research on self-stabilization is considered in the presentation of every corresponding problem that we study.

2.1 State Machine Replication

Leslie Lamport was the first to introduce *State Machine Replication* (SMR), presenting it as an example in [5]. Schneider [55] gave a more systematic approach to the design and implementation of SMR protocols. A server (replica) defines a state machine with state variables that are modified by operations. When a client issues a request to a server with a specific operation, the server, under certain conditions, executes the operation and a state transition takes place. SMR requires replicas to define a common order of execution of such requests, in order to generate the same state transitions, and thus retain the same state [56]. This approach has the underlying assumption of a consistent initial state. In the case of self-stabilization this is not something given, but rather it is what is being asked for, namely, for the system to be able to converge to a consistent legal system state that also defines a legal replication state from which to continue replication.

2.1.1 Consensus and State Machine Replication

The *consensus* problem [7, 57] requires that a set of processors, each with an initial value, to eventually agree on a single such value. Any solution to consensus must provide the following three properties: the decided value is one of the processors' initial values (*validity*), all correct processors decide on the same value (*agreement*),

and all correct processors eventually decide (*termination*). It was shown that there is no deterministic algorithm providing termination (which is the liveness property of consensus) in any asynchronous system with faulty processors. This holds even in the case where only a single processor is crash-prone. This result is known as the "FLP impossibility", or simply the "FLP" [58]. Fortunately, there are several approaches to circumvent this impossibility [37]. Popular ways to do this, are to introduce synchrony and assume a known delay to the system's communication, or to use randomization and thus sacrifice determinism [59]. Another way is to employ a failure detection mechanism [60], while some works move on to modify the problem and ask for weaker guarantees.

Intuitively, a consensus-based replicated state machine runs repeated instances of consensus to allow replicas to reach an agreed order of execution of state operations. Indeed, consensus is probably the most well-established and favored technique to achieve SMR. The best-known consensus algorithm is the Paxos protocol by Lamport [7]. The protocol considers replicas that maintain the application state and communicate with clients, but also leaders and acceptors that coordinate the replica. They communicate to reach to a common value and announce this to the replicas that are responsible to execute/store and report on the result.

Paxos has seen a series of optimizations and specializations tackling different failure models [61–63]. Google Chubby [64], Google Spanner [65] and more recently Google Cloud Spanner, all employ some implementation of the Paxos algorithm to achieve consensus and also to cater for replication. Raft [66], a Paxos alternative also implements consensus-based SMR, and it is suggested to be easier to understand (than Paxos) from an engineering point of view.

Dolev et al. [28] consider the shared memory model to give practically-self-stabilizing consensus. (Although this appears to be the first work on practically-self-stabilization, the term was not established at the time). They then build a consensus algorithm for a rotating coordinator that works with bounded memory, and is used to implement a replicated state machine. The replicated state machine is composed of multiple instances (incarnations) of the consensus algorithm that are identified and ordered with a counter. To this end, they propose a self-stabilizing wait-free reset in order to deal with counter exhaustion, the result of a transient fault. Liveness is provided by a self-stabilizing version of the $\Diamond S$ failure detector which is the proven weakest failure detector to solve consensus [60, 67].

More recently, a practically-self-stabilizing version of Paxos [26], made use of the practically-self-stabilizing labeling scheme of Alon et al. [27] to acquire ballot numbers. To achieve the result it was required to extend the scheme to allow for multiple writers to increment the counters that the labels implement. The paper follows the Paxos protocol, deviating only when self-stabilization issues arise. Liveness is provided by the (Θ)-failure detector that we detail in Section 2.5.1.

Extracting the extended version of Alon et al.'s labeling scheme from the proposed Paxos protocol does not seem intuitive. In this thesis we extend Alon et al.'s scheme in a way that it remains decoupled from the application, which in our case is the virtually synchronous SMR. Our version of the scheme also requires smaller messages, since it only sends pairs of counters, whilst the one of [26] sends vectors.

2.1.2 Virtual Synchrony

A different approach to consensus when aiming for SMR is to use systems that implement the *virtual synchrony* (VS) paradigm, commonly provided by Group Communication Systems (GCS) [8–10, 68, 69]. GCSs [10, 30, 69, 70] can implement SMR based on reliable multicast within process groups, also catering for some delivery order guarantees [71]. Two fundamental components of these systems is the *membership service*, and the *reliable group multicast service* [10]. A group membership service is responsible to provide a functioning set of processors upon which applications can run. A reliable group multicast service is a service providing reliable message delivery (i.e., either all correct processors deliver a message or none), and possibly giving some other delivery guarantees (e.g., FIFO or causal ordering).

Birman et al. [29] were the first to suggest VS by implementing Isis[1]. This initiated a line of work with improvements in the efficiency of ordering protocols [9,16,72–74]. The project that started with Isis proceeded with similar systems implementing virtually synchronous reliable multicast, such as the Isis Toolkit [9], Horus [30] and Ensemble [75]. These were used in air-traffic control (in France), chemical refinery plants, stock-exchange markets (of New York and Switzerland) and elsewhere [8]. While Isis and its direct descendants work in the primary partition (i.e., with a majority group among the system's set of processors), other research groups developed

[1]The name originates from the Egyptian goddess Isis, and has nothing to do with the so-called "Islamic State of Iraq and Syria", a synonymy that has forced the rebranding of Isis2 to "Vsync"!

solutions that can cope with partitionable and merging groups (e.g., Totem [69] and Transis [68]). A concise account of the evolution and use of the VS model for SMR, as well as a comparison with consensus-based SMR (e.g., Paxos) is given in [8]. For a comparison of Virtual Synchrony with the database transactional model (a different approach to replication) see [76].

The group membership and the identifier of such a group constitute the *view*. In general, VS attempts to make an asynchronous execution appear as synchronous to the user (hence the name), by forcing that the views are totally ordered in the computation, and all members of a view v that move to the consecutive view v' will have delivered the same messages. This distinguishes the time at which a message is *received* by the communication layer, from the time it is *delivered* to the upper layers. In general, virtual synchrony requires reliable delivery of messages within a view, and thus, all non-crashed members or a view deliver a message of none delivers it. Different implementations of VS provide a range of guarantees on the ordering of messages within the same view. One may want to relax the strict guarantees provided by the atomic broadcast (known as ABCAST) to achieve better performance with FIFO or causal ordering [8]. In general, the synchronization of the views that VS imposed is aimed at helping developers reason about their system without becoming entangled to its asynchrony.

A self-stabilizing GCS is suggested by [45]; it is based on a self-stabilizing group membership service in [77] and on token circulation on a virtual ring of processors, and does not adopt a virtually synchronous approach. To our knowledge there is no work on self-stabilizing virtual synchrony, besides the one considered in this thesis.

2.1.3 Consensus-based and VS-based SMR

Comparing the two models, namely consensus and VS, is a rather intriguing task due to the fact that historically the notions have both initially diverged from the primal meaning of state machine replication, and have today converged in many ways [8]. Multicasting with the virtual synchrony approach often progresses more efficiently than the consensus-based solutions [8], although theoretically implementing strong virtual synchrony guarantees can be harder than consensus [78]. Virtual Synchrony is deemed to have a simpler group membership protocol compared to Paxos, and it runs this as a separate service rather than as part of the protocol.

Nowadays, one can find systems that draw paradigms from both models, such as Google Chubby [64] and Apache ZooKeeper, which is currently employed by Yahoo! for many of its services [79]. Descendants of Isis such as Vsync [15] and Derecho [16] are directed towards high-speed cloud services and offer implementations of Paxos, while virtual synchrony handles membership [74]. In fact, they are suggested to be among the most efficient implementations of Paxos [16].

2.2 Shared Memory Emulation

In Section 1.2 we discussed *Shared Memory Emulation* (SME), i.e., how to emulate shared memory model operations over a message-passing system. We continue by further reviewing this area, with special attention given to self-stabilizing SME literature.

2.2.1 Non-Self-Stabilizing SME

The seminal result on atomic shared memory emulation by Attiya, Bar-Noy and Dolev (known as ABD) [32] provides guarantees about the consistency of a distributed shared object. In particular, it guarantees *atomicity* [33, 80]. The atomic single-writer multiple-reader protocol proposed is simple to understand, and this explains part of its success and influence[2]. The approach has generated work in both static settings and dynamic ones, and in both the crash failure and the Byzantine failure model. The service can be run in a static environment on a fixed set of processors, but also in a dynamic environment using a reconfiguration service to provide a set of reliable processors to act as service providers. We see this in the sequel.

ABD works as follows. The single processor that can write a value to the register, called the *writer*, maintains an integer counter from which it draws numbers to timestamp the version of any value written to the register. Whenever a write is required, the writer increments the counter, and together with the value it writes it to the quorum by sending it to the processors, and then waiting for a response from some quorum (e.g., a majority) before returning. The quorum pair-wise intersection property ensures that whenever a reader performs a read and gathers a majority of responses, it will certainly recover the value of a previous completed write. The two-

[2]It was awarded the Edsger W. Dijkstra Paper Prize in Distributed Computing in 2011.

phase read (composed of a read and a write back to the processors) ensures atomicity. An extension by [34] allows for multiple writers with an extra communication round for the writer, since the timestamp in this case is not maintained only by a single writer.

Attiya and Bar-Noy [81] propose a single-writer multi-reader register emulation in the presence of Byzantine servers and *semi-Byzantine* clients (that may either stop-fail or perform a memory access operations incorrectly, but these operations are confined to specific memory objects). Malkhi and Reiter [82] consider shared memory emulation as an example service when they propose quorum systems that are Byzantine-tolerant. On the practical side, DepSky [83] is an implementation of a single-writer multi-reader shared register emulation on a set of untrusted storage clouds that can fail in an arbitrary way.

2.2.2 Self-Stabilizing SME

In recent years, a line of work has produced results on practically- and pseudo- self-stabilizing versions of shared memory emulation, in both synchronous and asynchronous message passing systems [26, 27, 84]. Alon et al. [27] propose a bounded labeling scheme that they use to gain practically unbounded sequence numbers to use as timestamps. In this way they leverage the creation of a practically-self-stabilizing single-writer multiple-reader (SWMR) shared memory emulation closely following the idea of the ABD algorithm [32]. The labels and the idea above were used to build a pseudo-stabilizing version of the *SWMR* [84].

Bonomi et al. [85] provide a pseudo-stabilizing Byzantine-tolerant algorithm for a multi-writer multi-reader *regular* register. The result considers a system of crash-prone clients performing reads and writes on a set of $n > 5f$ servers where at most f may be malicious. This upper bound on the number of malicious servers is proved for building stabilizing regular registers. The approach uses the labeling scheme by Alon et al. [27] to timestamp the write operations, and also provides a finite set of labels for readers to identify their read operations and verify whether they are missing responses by using the guarantees of a FIFO data-link implementation. The multi-writer register is built on top of a single-writer one, and it is suggested to be the first work on shared memory emulation to tackle both Byzantine behavior and transient faults.

In [86] the proposed stabilizing Byzantine-tolerant single-writer single-reader regular register is enhanced with the labeling scheme by Alon et al. [27] to provide atomic single-writer multi-writer *atomic* registers. These are practically-self-stabilizing. The setting is asynchronous and at most $f < n/8$ processors may exhibit malicious behavior.

A more challenging failure model is the *mobile* Byzantine one, that allows Byzantine agents to move to different servers. This is a realistic approach for a long-lived system where compensated processors are repaired possibly by running an automated recovery routine, while others that were not among the initial set of f possibly faulty, may become malicious. The approach has received theoretical interest [87,88].

In a continuation of this line of work, Bonomi et al. [89] consider the synchronous round-based setting where any number of clients may fail by crashing. Servers are subject to mobile Byzantine failures. The results consider four different models of mobile Byzantine failures that are defined in the related literature. These vary as to whether, after recovery, servers can detect that they were compensated or not, and as to the instance of a round that a faulty agent is permitted to move. The self-stabilizing multi-writer multi-reader parametric algorithm of this work is suggested to tackle all four models for different bounds on the number of mobile agents f for each of the mobile-Byzantine models. The bounds are proven and the algorithm is said to be tight in its complexity with respect to each of these.

2.3 Reconfiguration

Distributed systems that work in dynamic asynchronous environments often employ quorum configurations [90, 91], i.e., interconnected sets of active processors (servers or replicas), to provide services (such as shared storage [35]) to the system's participants. Over time, the configuration may gradually lose active participants due to voluntary leaves and stop failures. There is, therefore, the need to allow the participation of newly arrived processors and, from time to time, to *reconfigure* so that the new configuration is built on a more recent participation group. We overview such reconfiguration services for the crash-tolerant model, and we look into several attempts to address Byzantine-tolerant reconfiguration.

2.3.1 Crash-Tolerant Reconfiguration

Over the last years, a number of reconfiguration techniques have been proposed, mainly for state machine replication, and for the emulation of atomic shared memory, e.g., RAMBO [17], DynaStore [18], Geoquorums [92], and others [93–102]. These works concern the crash-failure model. We review some of the best known.

RAMBO [17] combines a reconfiguration mechanism with a quorum-based replication service to provide atomic SME that, as suggested, guarantees high reliability and availability. The techniques involved are: (i) *replication* to provide service over short intervals of time where crashes/departures from the configuration are expected to be low, and (ii) *reconfiguration* in the long run when the configuration accumulates significant losses. It guarantees consistency for a variety of system failures, with the assumption that the quorum system does not collapse before a reconfiguration finishes. Service does not stop even when replication and reconfiguration run concurrently, although this comes with the overhead of garbage collecting the histories of past configurations that have completed all the operations required to preserve the consistency of the distributed object.

While RAMBO uses consensus to allow participants to agree on a configuration, DynaStore [18] is suggested to avoid this through the use of non-atomic snapshots that generate a directed acyclic graph, the vertexes of which correspond to the different configurations that can be produced. Under suitable assumptions, the DynaStore algorithm achieves reconfiguration. The read and write operations are similar to those of ABD. The provided algorithm, i.e., DynaStore, is also deemed to be evidence that dynamic read/write storage is a weaker problem than dynamic consensus.

The work on GeoQuorums [92] takes a different approach. It builds atomic memory on top of an ad hoc network of mobile nodes that gather within geographic areas called "focal points". It assumes the existence of a GPS service to provide real-time timestamps for the reads and writes. For a theoretical comparison of RAMBO, DynaStore, GeoQuorums and others see [35], and for a practical evaluation based on implementation performances see [103].

More recently, Nogueira et al. [104] attempted to make reconfiguration more scalable and faster with "fast elasticity". To this end, they propose a modular partition transfer protocol for creating and destroying state partitions. This is suggested to

make the service more suitable for cloud systems.

When considering reconfiguration, two important questions are posed:
(i) *How does the system choose the next configuration?* This question drove research efforts to characterize the fault-tolerance guarantees that can be provided by different quorum system designs. For an in-depth discussion see [91].
(ii) *When does the system reconfigure?* One simple decision would be to reconfigure when a fraction (e.g., $1/4^{th}$) of the members of a configuration appear to have failed. More complex decisions could use prediction mechanisms (possibly based on statistics), but generally this is regarded as an application-orientated decision. For a discussion on this topic see the related discussion in [35].

To our knowledge, there is no work tackling the self-stabilizing reconfiguration task besides the one considered in this thesis. There are several reasons why existing reconfiguration techniques do not claim to be self-stabilizing. One of their basic assumptions is that the system starts in a consistent configuration, in which all processors are in their initial state, and that all processors are aware of a single configuration. Starting from that state, the system must strive to preserve consistency as long as a predefined churn rate of processors' joins and leaves is not violated and unbounded storage is available. Systems offering reconfiguration also often assume unbounded counters to order consensus messages that facilitate configuration updates, or to timestamp writes for shared memory emulation (e.g., [17]). Transient faults can lead such counters to become exhausted, or be otherwise corrupted. Another downside of existing systems is that they usually do not address the issue of recovering after the quorum system has collapsed. All of the above make such systems unable to automatically recover from the arbitrary system states forced by transient failures.

2.3.2 Byzantine-Tolerant Reconfiguration

As Lamport, Malkhi and Zhou note in [97], the vast body of the work in reconfiguration mostly considers fail-stop failures. They themselves provide some intuition as to how their reconfigurable replicated state machine could tolerate malicious faults, by using a malicious tolerant consensus protocol, but also by having *enough* clients requesting to stop the current configuration and choosing the new one. Unfortunately, there is no more discussion on this.

BFT-SMART [105] is suggested to support reconfigurable Byzantine-Fault-Tolerance. In spite of the existence of an experimental evaluation of their reconfiguration service, its correctness is not theoretically proved. Moreover, it is assumed that a special *trusted* client provides the configuration to be installed, and this removes a part of the difficulty of the reconfiguration task. Even though the problem they solve still does not appear to be trivial, it does not seem to be as difficult as the Byzantine-Tolerant reconfiguration that we discuss about.

In general, it remains a challenging open question whether a reconfigurable BFT service is possible to construct, and under what guarantees this is attainable. The only certain thing is that there is scarce evidence of work on reconfigurable BFT that provides provable guarantees [106]. There is therefore some way to go before proceeding to enhance such a service with the stabilization property.

2.4 Byzantine Fault Tolerance

We now review Byzantine Fault Tolerant replication with and without the self-stabilization property.

2.4.1 Non-Self-Stabilizing BFT

Replication in Byzantine asynchronous message-passing environments has received much attention with the classic result by Castro and Liskov [38] known as Practical Byzantine Fault Tolerance (PBFT). PBFT achieves replication by imposing a total order of execution on clients' requests. To this end, it employs a primary replica (server) to monitor the allocation of request identifiers for the $n = 3f + 1$ secondary replicas that execute. Since f of them might be faulty, execution is expected by only $2f + 1$. The problem studies malicious behavior by the servers and does not directly address such behavior by clients.

Briefly, a client request received by the primary is allocated a sequence number and sent as a *pre-prepare* message to all other servers. Note that the authenticity of a client's request is guaranteed due to the assumption of strong encryption primitives that prevent the message forgery. If servers receive a legitimate copy of the *pre-prepare* message, they issue a *prepare* message and this is sent to all the other servers, while waiting for $2f + 1$ such *prepare* messages to move to a similar *commit* phase

and then execute. Liveness is conditioned by timeouts to ensure that a primary not progressing the replication is ousted and a new one is installed. The paper has many optimizations to practically enhance performance[3], e.g., checkpoints, and garbage collection.

PBFT was followed by several other works that took different approaches or achieved optimizations [107], but it does not seem that many diverged from PBFT's general idea. In Zyzzyva [108], requests are executed in a speculative order and clients agree to commit changes based on history. In [109] the Spinning algorithm suggests that a version of [38] with a rotating primary can be more efficient, especially when facing a malicious primary that consistently delays, but not to the point of forcing its change. In [105], BFT-SMART proceeds in providing a more efficient implementation of BFT replication while also catering for state transfer. Aublin et al. [110] propose ABSTRACT, an abstraction for developing replication protocols and with this they develop a new optimized BFT protocol. An important work in tackling byzantine consensus (and thus Byzantine SMR) in the asynchronous setting, is Byzantine Paxos [63] with its optimizations, e.g., [111,112]. Adapt [113] reconciles different existing BFT protocols by adapting itself to use the "most suitable" protocol according to an evaluation mechanism.

Over the years, research has circumvented the impossibility results for the asynchronous BFT state machine problem by means of introducing non-determinism (e.g., [114]), restricting the asynchrony, adding failure detectors (e.g., Peer-review [115] and Baldoni et al. [53]) or using wormholes in hybridized architectures. A different approach assumes the use of some local trusted components that may only crash (but are assumed to not exhibit malicious behavior). Such are MinBFT and MinZyzzyva [116], which also manage to reduce throughput requirements in relation to PBFT. For a survey on related literature one can refer to [37]. Lately, BFT has also been studied due to the relation and use of BFT consensus in blockchain consensus models [117,118], e.g., by the Hyperledger blockchain platform [119].

2.4.2 Self-Stabilizing BFT

Self-stabilizing algorithms that tolerate both transient failures and Byzantine failures as well, are very challenging as we discussed in Section 1.3. The Byzantine agreement

[3]This is the "practical" aspect of *P*BFT.

algorithm by Daliot and Dolev [120] assumes a bounded message transmission delay, to achieve agreement of all correct processors to a value suggested by a possibly faulty initiator. It does so by associating the servers' local-time with the protocol initiation of the initiator. The message exchange takes place with a proposed reliable broadcast primitive, while the solution is suggested to be optimal in time.

As we have already documented, a newer work available on self-stabilizing Byzantine tolerant replication is by Binun et al. [39] that presents the first self-stabilizing BFT replication service assuming semi-synchrony and known bounds on message delivery. In particular, they employ the self-stabilizing Byzantine-tolerant clock synchronization algorithm of [40] so that upon every clock pulse, $n+1$ instances of leaderless Byzantine consensus are initiated, one for the system state and n for each of the processors' histories, so that processors will agree on a common message history and replica state. The work concludes with an evaluation of a prototype of this algorithm, implemented as a self-stabilizing Byzantine-tolerant replicated Hadoop master node. The evaluation suggests that the cost of self-stabilization is not prohibitive, and that building efficient fault-tolerant self-stabilizing systems is both practical and possible.

Generally, in tackling the problem of BFT, one may employ the *authenticated* model with unforgeable digital signatures, or the *unauthenticated*, where the source of a message can be verified [121]. To our knowledge, self-stabilizing solutions that assume public key cryptography are very limited in number (e.g., [122]) and come under the assumption that correct processors do not leak their private keys. The unauthenticated model is the one usually employed, but with the assumption of existence of self-stabilizing authenticated links that guarantee the identity of the sender of a message. Such an implementation of data-links is proposed by [123].

The issue is twofold. Primarily, one cannot guarantee that during transient faults the primary keys have not been exposed to other processors [52]. Indeed, in [121] it is assumed that this does not happen in order to probabilistically build self-stabilizing and Byzantine-tolerant overlay networks. Secondly, self-stabilization is particularly suited to *mobile* Byzantine failures [87], where the malicious agents may move to different processes, since it can model such failures as transient faults. When considering such mobility, assuming public key cryptography is not useful, since the private key of a now correct processor may have leaked when it was compromised. This can be exploited by currently malicious agents. The line of work on mobile

Byzantine failures and self-stabilizing atomic memory [86, 89, 123] was reviewed in Section 2.2.

2.5 Data-link Protocols and Failure Detectors

Here we review data-link and failure detector protocols that are relevant to our work.

2.5.1 Self-Stabilizing Data-Link Protocols

There is considerable work on self-stabilizing data-link protocols [124], e.g., for the alternating bit protocol [125] and generally for the sliding window protocol [126]. For our purposes, we assume that our system runs on a stabilizing data-link layer that provides reliable FIFO communication over unreliable bounded capacity channels as the ones of [127, 128].

The general idea of [127] and of [128], which also handles duplication errors in the channels, is the following. A processor that sends a packet π to another processor, inserts a copy of π to the FIFO queue that represents the communication channel to the receiver. Since links have bounded capacity, respecting the capacity implies that there are possible omissions of either new packets, or one of the already sent packets. When π is received, it is dequeued from the queue representing the channel. Data packets are retransmitted until more acknowledgments than the total capacity *cap* arrive. Intuitively, this ensures that the (at most *cap*) stale packets from an arbitrary state are not considered, although the actual details of the papers handle several subtle issues.

This data-link enables the two connected processors to constantly exchange a "token". Specifically, the sender constantly sends packet π_1 until it receives enough acknowledgments (more than the capacity). Then, it constantly sends packet π_2, and so on and so forth. This assures that the receiver has received packet π_1 before the sender starts sending packet π_2. This can be viewed as a token exchange. We use this abstraction of the token that carries messages back and forth between any two communication entities, to implement a *heartbeat* to (imperfectly) detect whether a processor is communicating or not; when a processor in no longer active, the token will not be returned back to the other processor. We discuss this further in the next section.

2.5.2 Failure Detectors

Failure detection enables asynchronous fault-tolerant distributed systems to provide liveness guarantees [67]. There is a huge literature on the topic of failure detection. The discussion will focus on the results that are more relevant to this thesis.

The (Θ)-failure detector. The (Θ)-failure detector was introduced in [129], but a self-stabilizing version was produced by Blanchard et al. [26]. It is implemented as follows. Every processor p uses the token-based mechanism described above to implement a heartbeat with every other processor. In particular, for every processor q in the system, p maintains a heartbeat integer counter. Whenever p receives the token from q over their data link, p resets the counter of q to zero, and increments all the integer counters associated with every other processor by one, up to a predefined threshold value W. If the heartbeat counter of a processor q' reaches W, the failure detector of p considers q as "suspected" (and thus possibly inactive or crashed). In other words, the failure detector at processor p considers processor q' to be active if and only if the heartbeat associated with q is strictly less than W. In this thesis we employ the self-stabilizing version of the (Θ)-failure detector either as it is, or enhanced.

In particular, for the virtual synchrony algorithm (Chapter 4), we employ the (Θ)-failure detector, but with weaker requirements than [26], because in [26] it is required to solve consensus, and naturally they resort to a failure detector at least as strong as Ω [60]. The VS algorithm does not require an eventually perfect failure detector that ensures that after a certain time, no active processor suspects any other active processor. Our requirements, on the other hand, are stronger than the weakest failure detector required to implement atomic registers (where at most a minority of failures are assumed), namely the Σ failure detector [130], since virtual synchrony is a more difficult task.

For the self-stabilizing reconfiguration service (Chapter 5) we enhance the (Θ)-failure detector to form the (N, Θ)-failure detector. This reports on the N most responsive processors, where N is an upper bound to the number of processors that are live and connected at any time in the computation.

For the self-stabilizing BFT service that we propose in Chapter 6 we have greater requirements. We need to detect both *responsiveness* (i.e., whether processors are

actively communicating), and also *progression* of computation by the primary processor. Following the approach of Baldoni et al. [53], we use *both* the (Θ)-failure detector to check the responsiveness of the primary and the heartbeat along with the threshold to produce an enhanced version of the (Θ)-failure detector to check that the primary is progressing the state machine. In particular, every non-primary processor uses the heartbeat to monitor the client requests for which a primary's action is pending. If the heartbeat exceeds the threshold, meaning that the primary has not processed requests for a given number of token exchanges, then the primary is suspected as faulty (more details in Chapter 6).

Byzantine Failure Detectors. Byzantine failure detectors identify and permanently suspect malicious processors. This simplifies the design of state machine replication, since the actions and messages of detected processors are ignored. Baldoni et al. [53] use a pair of Byzantine failure detectors. The muteness failure detector $\Diamond\mathcal{M}$ detects non-responsiveness, and another component to detect Byzantine behavior. Alvisi at el. [131] use a statistical approach to estimate the risk posed by faulty processors. PeerReview [115] implements the notion of accountability by storing secure records of messages sent and received to ensure that correct processors are never suspected.

Building self-stabilizing Byzantine failure detectors with proven guarantees and strength is another open research direction. For the purposes of our BFT implementation in Chapter 6 this was not necessary, as we only required checking responsiveness and progress of the state machine by the primary processor.

Chapter 3

System Settings and Definitions

We present the general system settings and definitions. Any system settings and definitions that are specific to the work of each of the next chapters are given at the beginning of each chapter.

3.1 Distributed Setting

We consider an asynchronous message-passing system. The system is composed of communicating entities named *processors*. Every processor joins the system carrying a unique integer identifier from an ordered set $\mathcal{I}$, such that p_i is the processor with identifier $i \in \mathcal{I}$.

In Chapters 4 and 6 we consider a fixed set of processors P, such that $|P| = n$; in particular, $\mathcal{I} = \{1, 2, \cdots, n\}$. In Chapter 5 we consider the number of live and connected processors at any point in the computation to be bounded by some integer N such that $N \ll |\mathcal{I}|$.

Since we tackle problems with both static (fixed) and dynamic processor sets, we define this in the problems' dedicated sections.

3.2 Failure Model

In Chapters 4 and 5 any processor may fail by crashing at any point in the computation, and from this point it takes no further steps. A processor that has failed by crashing is referred to as *crashed*, whereas a non-crashed processor is called *active* or *correct*.

In Chapter 6 at most f processors may (intentionally or not) exhibit Byzantine (malicious) behavior, i.e., fail to follow the defined protocol. They are referred to as *faulty*. Processors that fail by crashing are included in f, since a crashed processor is modeled as a malicious one that does not perform the communication part of the protocol. A non-faulty processor, i.e., one that does not exhibit malicious behavior is called *correct*. We define the size of f in Chapter 6.

Additionally, the system may also suffer *transient faults*, which are short-term violations of the system's design assumptions. These may corrupt the system's state and introduce *stale information* in local variables and program counters of any number of processors, as well as in the communication links. We assume that the hardware, the program's code and any hard-coded system parameters are left intact.

3.3 Communication and Data Link Implementation

The network topology is that of a fully connected graph. We assume that the system runs on top of a stabilizing data-link layer that provides reliable FIFO communication over unreliable bounded capacity channels such as the ones of [127, 128]. Every pair of processors exchange low-level messages called *packets* to enable a reliable delivery of high-level messages. When no confusion is possible, we use the term "messages" for packets.

Communication links have bounded capacity, so that the number of messages in any link at every given instance is bounded by a constant *cap*; a parameter known to processors. Packets sent may be lost, reordered, or duplicated but not arbitrarily created, although the channels may initially (after transient faults) contain stale packets. Due to the boundedness of the channels, the total number of stale packets in the communication links is also bounded $O(n^2 cap)$ (and $O(N^2 cap)$ for Chapter 5) system-wide. Although packets may be spontaneously omitted (lost) from the channels, we assume that communication channels are fair, thus, a packet sent infinitely often is received infinitely often.

When processor p_i sends a packet, π, to processor p_j, the operation *send* inserts a copy of π to the FIFO queue that represents the communication channel from p_i to p_j, while respecting an upper bound on the number of packets in the channel, possibly omitting the new packet or one of the already sent packets. When p_j receives π from p_i, π is dequeued from the queue representing the channel.

One version of a self-stabilizing FIFO data link implementation that we can use, is based on the fact that communication links have bounded capacity. Packets are retransmitted until more than the total capacity acknowledgments arrive. It is assumed that acknowledgments are sent only when a packet arrives (i.e., not spontaneously) [127, 128]. Over this data-link, the two connected processors can constantly exchange a "token". Specifically, the sender (possibly the processor with the highest identifier among the two) constantly sends packet π_1 until it receives enough acknowledgments (more than the capacity). Then, it constantly sends packet π_2, and so on and so forth. This ensures that the receiver has received packet π_1 before the sender starts sending packet π_2. This can be viewed as a token exchange. We use the abstraction of the token carrying messages back and forth between any two communication entities, to implement a *heartbeat* to (imperfectly) detect whether a processor is active or not; when a processor in no longer active, the token will not be returned back to the other processor.

Some specific communication requirements are given in the corresponding specific settings section of each chapter.

3.4 The Interleaving Model

Every processor p_i executes a program that is a sequence of *(atomic) steps*. Each atomic step starts with local computations and ends with a communication operation, i.e., packet *send* or *receive*. We assume the standard interleaving model where at most one step is executed in every given moment. An input event can either be the arrival of a packet or a periodic timer triggering p_i to (re)send. Note that the system is asynchronous and the rate of the timer is totally unknown.

The *state* σ_i, of an active processor p_i, consists of p_i's variable values and the content of p_i's incoming communication channels. Whenever p_i executes a step, this can change the σ_i. The tuple of the states σ_i of all active processors p_i that are active at a specific instance of the computation defines the *system state* at that instance. An *execution (or run)* $R = c_0, a_0, c_1, a_1, \ldots$ is an alternating sequence of system states c_x and steps a_x, such that each state c_{x+1}, except the initial system state c_0, is obtained from c_x by the execution of step a_x. A computation step a is *applicable* if there exists a state c' which is reached after step a is taken in state c. An execution is *fair* when every correct processor that has an applicable step a_i infinitely often, executes a_i infinitely

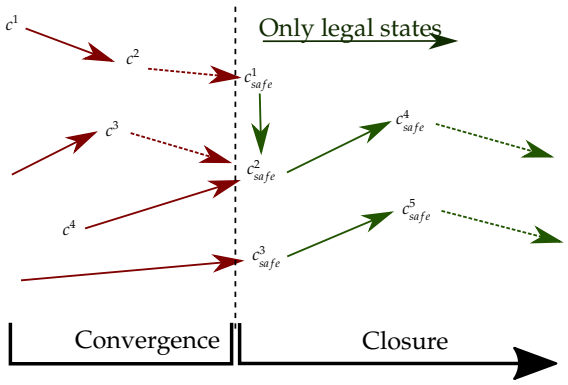

Figure 3.1: Stabilizing algorithms guarantee that after the period of convergence from any system state (even illegal ones such as $c^1, ..., c^4$), we are lead to a set of safe states (e.g., $c^1_{safe}, ..., c^5_{safe}$), and we never deviate from these unless a new transient fault takes place.

often. The system's task is a set of executions called *legal executions* (*LE*) in which the task's requirements hold.

3.5 Self-Stabilization

An algorithm is *self-stabilizing* with respect to *LE* when every (unbounded) execution R of the algorithm has a suffix R' that is in *LE*. Consider a state c_s such that any fair execution of an algorithm starting from c_s belongs to *LE*. We say that c_s is a *safe* state with respect to *LE* and the executed algorithm. We thus require that any self-stabilizing algorithm guarantees *convergence* to a such state that is safe within bounded time (polynomial w.r.t. to the system's dimensions), and also *closure* to the set of safe states. See also Figure 3.1.

The code of a self-stabilizing algorithm reflects the requirement for non-termination in that it usually consists of a `do – forever` loop that contains communication operations with the neighbors and validation that the system is in a consistent state as part of the transition decision. An *iteration* of an algorithm formed as a `do – forever` loop is a complete run of the algorithm starting in the loop's first line and ending at the last line, regardless of whether it enters branches.

Notation. We indicate a variable *var* in the local state of a processor p_i by var_i, i.e., with a subscript of the processor's identifier. Similarly, an execution of a function/macro $fun()$ by p_i with $fun_i()$. When it is clear from the context that *var* (or $fun()$) is owned (called) by p_i, then the subscript may be omitted. We may use the same notation with predicates defined for proofs.

Chapter 4

Practically-Self-Stabilizing Virtual Synchrony

A relatively new self-stabilization paradigm is *practically-self-stabilization* [24, 26–28]. Consider an asynchronous system with bounded memory and data link capacity, which contains stale information due to a transient fault. Such corrupt data may appear *unexpectedly* at any processor as they lie in communication links, or may (indefinitely) remain "hidden" in some processor's local memory until they are added to the communication links as a response to some other processor's input. Whilst these are bounded in number due to the boundedness of the links and local memory, they can eventually force the system to lose its safety guarantees. Such corrupt information may repeatedly drive the system to an undesired state of non-functionality. This is true for all systems and self-stabilizing systems are required to eradicate all corrupted information. In fact, whenever they appear, the self-stabilizing system is required to regain consistency and in some sense stabilize. One can consider this as an adversary with a limited number of chances to interrupt the system, but only itself knows *when* it will do this.

In this perspective, self-stabilization, as it was proposed by Dijkstra [20], is not the best design criteria for asynchronous systems for which we cannot specifically define *when* stabilization is expected to finish (in some metric like asynchronous rounds, for example). The newer criterion of practically-stabilizing systems is closely related to pseudo-self-stabilizing systems [25], as we explain next. Burns, Gouda and Miller [25] deal with the above challenge by proposing the design criteria of *pseudo-self-stabilization*, which merely bounds the number of possible safety violations. Namely, their approach is to abandon Dijkstra's seminal proposal [20] to bound the period in which such violations occur (using some metric like asynchronous cycles). We consider a variation on the design criteria for pseudo-self-stabilization

systems that can address additional challenges that appear when implementing a decentralized shared counter that uses a constant number of bits.

Self-stabilizing systems can face an additional challenge due to the fact that a single transient fault can cause the counter to attain its maximum possible value and still (it is often the case that) the system needs to be able to increment the counter for an unbounded number of times. The challenge becomes greater when there is no elegant way to show that the system can always maintain an order among the different values of the counter by, say, wrapping to zero in such integer overflow events. Arora, Kulkarni and Demirbas [132] overcome the challenge of integer overflow by using non-blocking resets in the absence of faults described [132]. In case faults occur, the system recovery requires a blocking operation, which performs a distributed global reset. This work considers a design criteria for message-passing systems that perform in a wait-free manner even when recovering from transient faults.

From the theoretical point of view, systems that take an extraordinary large number of steps (that exceeds the counter maximum value, or even an infinite number of steps) are bound to violate any ordering constraints. This is because of the asynchronous nature of the studied system, which could arbitrarily delay a node from taking steps or defer the arrival of a message until such violations occur after, say, a counter wraps around to zero. Having practical systems in mind, we consider systems for which the number of sequential steps that they can take throughout their lifetime is not greater than an integer that can be represented using a constant number of bits. For example, Dolev, Kat and Schiller [28] assume that counting from zero to $2^{64} - 1$ using sequential steps is not possible in any practical system and thus consider only a *practically infinite period*, of 2^{64} sequential steps, that the system takes when demonstrating that safety is not violated. The design criteria of practically-self-stabilizing systems [24, 26, 27] requires that there is a bounded number of possible safety violations during any practically infinite period of the system execution.

We proceed to present the practically-self-stabilizing Virtual Synchrony scheme that we already discussed and motivated in Chapters 1 and 2. We start with the required specific definitions and system settings.

4.1 Specific System Settings and Definitions

We present the required definitions specific to this setting.

4.1.1 Practically-Self-Stabilization

An execution R_p is a *practically infinite execution* if it contains a chain of steps ordered according to Lamport's happened-before relation [5] that are longer than 2^τ (τ being, for example, 64); namely they are practically infinite for any given system [28] (see discussion in Section 1.2). Similar to an infinite execution, a processor that fails by crashing, stops taking steps, and any processor that does not crash, eventually takes a practically infinite number of steps.

We define the system's abstract task $\mathcal{T}$ by a set of variables (of the processor states) and constraints, which we call the system requirements, in a way that implies the desired system behavior [19]. Note that an execution R can satisfy the abstract task and still not belong to LE, because R considers only a subset of variables, whereas the states of executions that are in LE consider every variable in the processor states and message fields. An algorithm is *practically-self-stabilizing* (or just *practically-stabilizing*) with relation to the task $\mathcal{T}$ if in any practically infinite execution it has a bounded number of deviations $\mathcal{T}$ [24].

4.1.2 Complexity Measures

The above definition of practically-stabilizing algorithms, suggests that there are a bounded number of corrupt elements in the state (either messages or local stale corruptions) that might force the system to *deviate* from its task even if these may or may not appear due to asynchrony. Whenever a deviation happens, a number of algorithmic operations are required to satisfy $\mathcal{T}$ once again. This defines a natural measure of complexity, in particular, we bound the deviations as the total number of times recovery operations take place throughout an execution. These operations differ by algorithm, i.e., it is label creations in the labeling scheme, counter increments for the counter increment algorithm and view creations in the virtual synchrony algorithm.

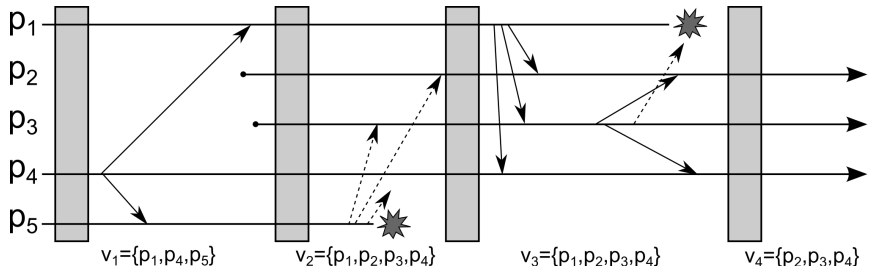

Figure 4.1: An execution satisfying the VS property. The grey boxes indicate a new view installation, and the example shows four views. View v_1 initially with membership $\{p_1, p_4, p_5\}$. The reliable multicast reaches all members of the group. Two new processors p_2 and p_3 join the group, forming view v_2. In this view, p_5 crashes before completing its multicast which is ignored (dashed lines). The new view v_3 is formed to exclude p_5, and in it, p_1 manages a successful multicast before crashing. The multicast of p_3 is reliable and guaranteed to be delivered to all non-crashed within the view, that is excluding p_1 which might or might not have received it (dotted line). A new view is then formed to encapture the failure of p_1.

4.1.3 The Virtual Synchrony Task

The *virtual synchrony task* uses the notion of a *view*, a group of processors that perform multicast within the group and is uniquely identified, to ensure that any two processors that belong to two views that are consecutive according to their identifier, *deliver* identical message sets in these views. The legal execution of virtual synchrony is defined in terms of the input and output sequences of the system with the environment. When a majority of processors are continuously active, every external input (and only the external inputs) should be atomically accepted and processed by the majority of the active processors. The system works in the primary majority, i.e., it does not deal with partitions and requires that a view contains a majority of the system's processors, i.e., its membership size is always greater than $n/2$. Therefore, there is no delivery and processing guarantee in executions in which there is no majority, still in these executions any delivery and processing is due to a received environment input. Figure 4.1 is an example of a virtually synchronous execution.

Notation. Throughout the chapter we use the following notation. Let y and y' be two objects that both include the field x. We denote $(y =_x y') \equiv (y.x = y'.x)$.

4.2 Solution Outline

The virtually synchronous SMR algorithm presented in this chapter as Algorithm 4 is built incrementally. We start by building a counter algorithm to provide view identifiers as required by the task. At the heart of our counter algorithm is the underlying labeling algorithm (Algorithm 2) that extends the labeling scheme of Alon et al. [27] to support *multiple* writers, whilst the algorithm specifies how processors exchange their label information in the asynchronous system and how they maintain proper label bookkeeping so as to "discover" the greatest label and discard all obsolete ones.

By extending the labels with integer counters, we provide a multi-purpose practically-self-stabilizing counter algorithm (Algorithm 3) using only bounded memory and communication bandwidth. With this, many writers can increment the counter concurrently for an unbounded number of times in the presence of processor crashes and unbounded communication delays. An immediate application of our counter algorithm, as we explain in Section 4.3.3, is a practically-self-stabilizing MWMR register emulation.

The practically-self-stabilizing counter algorithm, together with implementations of a practically-self-stabilizing reliable multicast service and membership service that we propose, are composed to yield a practically-self-stabilizing coordinator-based Virtual Synchrony solution. Our Virtual Synchrony solution yields a practically-self-stabilizing State Machine Replication (SMR) implementation (Algorithm 4). As this implementation is based on virtual synchrony rather than consensus, the system can progress in more extreme asynchronous executions than consensus-based SMR.

In the sequel, Section 4.3 details the practically-self-stabilizing Labeling Scheme and Increment Counter algorithms. Section 4.4 presents the practically-self-stabilizing Virtual Synchrony algorithm and the resulting replicate state machine emulation.

4.3 Practically-Self-Stabilizing Labeling Scheme and Counter Algorithm

Many system like the ones performing replication (e.g. GCSs requiring group identifiers, Paxos implementations requiring ballot numbers) assume access to an infinite (unbounded) counter. We proceed to give a practically-stabilizing, practically infinite counter based on a bounded labeling scheme. Note that by a *practically infinite (*or *unbounded) counter* we imply that a τ-bit counter (e.g., 64-bit) is not truly infinite (since this is anyway not implementable on hardware), but it is large enough to provide counters for the lifetime of most conceivable systems when started at 0. We refer the reader to the example provided by Blanchard et al. [26], where a 64-bit counter initialized at 0 and incremented per nanosecond is calculated to last for around 500 years, essentially an infinity for most of today's running systems.

The task of a practically-self-stabilizing labeling scheme is for every processor that takes an infinite number of steps to reach to a label that is maximal for all active processors in the system. The task of maintaining a practically infinite counter, is for every processor that takes an infinite yet bounded number of steps, to eventually be able to monotonically increment the counter from 0 to 2^{τ}. The latter task depends on the former to provide the maximal label in the system to be used as a sequence number epoch, so that within the same epoch, the integer sequence number is incremented as a practically infinite counter. It is implicit that the tasks are performed in the presence of corrupt information that might exist due to transient faults.

Our solutions are *practically infinite*, in the following way. A bounded amount of stale information from the corrupt initial state, may unpredictably corrupt the counter. In such cases, processors are forced to change their labels and restart their counters. A processor cannot predict whether a corrupt piece of information exists, or *when* will it make its appearance as this is essentially the work of asynchrony. Our solutions guarantee that only a bounded number of labels will need to change, or that only a bounded number of counter increments will need to take place before we reach to one that is eligible to last its full 2^{τ} length, less the fact that this maximal value is practically unattainable.

We first present and prove the correctness of a practically-stabilizing labeling algorithm, and then explain how this can be extended to implement practically

stabilizing, practically unbounded counters in Section 4.3.3.

4.3.1 Labeling Algorithm for Concurrent Label Creations

Preliminaries

Bounded labeling scheme. The bounded labeling scheme of Alon et al. [27] implements an SWMR register emulation in a message-passing system. The *labels* (also called *epochs*) allow the system to stabilize, since once a label is established, the integer counter related to this label is considered to be practically infinite, as a 64-bit integer is practically infinite and sufficient for the lifespan of any reasonable system. We extend the labeling scheme of [27] to support multiple writers, by including the epoch creator (writer) identity to break symmetry, and decide which epoch is the most recent one, even when two or more creators concurrently create a new label.

Formally defined, we consider the set of integers $D = [1, k^2 + 1]$ such that $k \in \mathbb{N}$ a known constant to the processors, which we determine in Corollary 4.3.2. A *label* (or *epoch*) is a triple $\langle lCreator, sting, Antistings \rangle$, where *lCreator* is the identity of the processor that established (created) the label, $Antistings \subset D$ with $|Antistings| = k$, and $sting \in D$. Given two labels ℓ_i, ℓ_j, we define the relation $\ell_i \prec_{lb} \ell_j \equiv (\ell_i.lCreator < \ell_j.lCreator) \vee (\ell_i.lCreator = \ell_j.lCreator \wedge ((\ell_i.sting \in \ell_j.Antistings) \wedge (\ell_j.sting \notin \ell_i.Antistings)))$; we use $=_{lb}$ to say that the labels are identical. Note that the relation $\prec_{lb}$ does not define a total order. For example, when $\ell_i =_{lCreator} \ell_j$ and $(\ell_i.sting \notin \ell_j.Antistings)$ and $(\ell_j.sting \notin \ell_i.Antisting)$ these labels are *incomparable*.

As an example, consider the situation with $k = 3$, and $D = \{1, 2, \ldots, 10\}$. Assume the existence of three labels $\ell_1 = \langle i, 2, \langle 3, 5, 9 \rangle \rangle$, $\ell_2 = \langle i, 1, \langle 2, 9, 10 \rangle \rangle$, and $\ell_3 = \langle i + 1, 1, \langle 3, 5, 9 \rangle \rangle$. In this case, $\ell_1 \prec_{lb} \ell_3$ and $\ell_2 \prec_{lb} \ell_3$, since the creator of ℓ_3 has a greater identity than the creator of ℓ_1 and ℓ_2. We can also see that $\ell_1 \prec_{lb} \ell_2$, since the sting of ℓ_1, namely 2, belongs to the antistings set of ℓ_2 (which is $\langle 2, 9, 10 \rangle$) while the opposite is not true for the sting of ℓ_2. This makes ℓ_2 "immune" to the sting of ℓ_1.

As in [27], we demonstrate that one can still use this labeling scheme as long as it is ensured that eventually a label greater than all other labels in the system is introduced. We say that a label ℓ ***cancels*** another label ℓ', either if they are incomparable or they have the same *lCreator* but ℓ is greater than ℓ' (with respect to *sting* and *Antistings*). A label with creator p_i is said to belong to p_i's domain.

Algorithm 1: The *nextLabel*() function; code for p_i

```
1 For any non-empty set X ⊆ D, function pick(d, X) returns d arbitrary elements of X;
2     input S = ⟨ℓ_1, ℓ_2 ..., ℓ_k⟩ set of k labels;
3     output ⟨i, newSting, newAntistings⟩;
4 let newAntistings = {ℓ_j.sting : ℓ_j ∈ S};
5 newAntistings ← newAntistings ∪ pick(k − |newAntistings|, D \ newAntistings);
6 return ⟨i, pick(1, D \ (newAntistings ∪ {∪_{ℓ_j∈S} ℓ_j.Antistings})), newAntistings⟩;
```

Creating a largest label. Function *nextLabel*(), Algorithm 1, gets a set of at most k labels as input and returns a new label that is greater than all of the labels of the input, given that all the input labels have the same creator i.e., the same *lCreator*. This last condition is imposed by the labeling algorithm that calls *nextLabel*(), as we will see further down with a set of labels from the same processor. It has the same functionality as the function called $Next_b()$ in [27], but it additionally appends the label creator to the output. The function essentially composes a new *Antistings* set from the stings of all the labels that it receives as input, and chooses a *sting* that is in none of the *Antistings* of the input labels. In this way it ensures that the new label is greater than any of the input. Note that the function takes k *Antistings* of k labels that are not necessarily distinct, implying at most k^2 distinct integers and thus the choice of $|D| = k^2 + 1$ allows to always obtain a greater integer as the *sting*. For the needs of our labeling scheme, $k = 4(n^3cap + 2n^2 - 2n) + 1$ (Corollary 4.3.2).

Scheme idea and challenges. When all processors are active, the scheme can be viewed as a simple extension of the one of [27]. Informally speaking, the scheme ensures that each processor p_i eventually "cleans up" the system from obsolete labels of which p_i appears to be the creator (for example, such labels could be present in the system's initial arbitrary state). Specifically, p_i maintains a bounded FIFO history of such labels that it has recently learned, while communicating with the other processors, and creates a label greater than all that are in its history; call this p_i's *local maximal label*. In addition, each processor seeks to learn the *globally maximal label*, that is, the label in the system that is the greatest among the local maximal ones.

We note here that compared to Alon et al. [27], which only had a single writer upon the failure of whom there would be no progress thus stabilization would not be the main concern, we have *multiple* label creators. If these creators were not allowed to crash then the extension of the scheme would be a simple exercise. Nevertheless,

when some processors can crash the problem becomes incrementally more difficult as we now explain. The problem lies in cleaning the system of these crashed processors' labels since they will not "clean up" their local labels. Each active processor needs to do this itself, indirectly, without knowing which processor is inactive, i.e., we do not employ any form of failure detection for this algorithm. To overcome this problem, each processor maintains bounded FIFO histories on labels appearing to have been created by other processors. These histories eventually accumulate the obsolete labels of the inactive processors. The reader may already see that maintaining these histories, also creates another source of possible corrupt labels. We show that even in the presence of (a minority of) inactive processors, starting from an arbitrary state, the system eventually converges to use a global maximal label.

Let us explain why obsolete labels from inactive processors can create a problem when no one ever cleans (cancels) them up. Consider a system starting in a state that includes a cycle of labels $\ell_1 \prec \ell_2 \prec \ell_3 \prec \ell_1$, all of the same creator, say p_x, where $\prec$ is a relation between labels. If p_x is active, it will eventually learn about these labels and introduce a label greater than them all. But if p_x is inactive, the system's asynchronous nature may cause a repeated cyclic label adoption, especially when p_x has the greatest processor identifier, as these identifiers are used to break symmetry. Say that an active processor learns and adopts ℓ_1 as its global maximal label. Then, it learns about ℓ_2 and hence adopts it, while forgetting about ℓ_1. Then, learning of ℓ_3 it adopts it. Lastly, it learns about ℓ_1, and as it is greater than ℓ_3, it adopts ℓ_1 once more, as the greatest in the system; this can continue indefinitely. By using the bounded FIFO histories, such labels will be accumulated in the histories and hence will not be adopted again, ending this vicious cycle.

The Labeling Algorithm

The labeling algorithm (Algorithm 2) specifies how the processors exchange their label information in the asynchronous system and how they maintain proper label bookkeeping so as to "discover" their greatest label and cancel all obsolete ones. Specifically, we define the abstract task of the algorithm as one that lets every node to maintain a variable that holds the local maximal label. We require that, after the recovery period and as long as there are no calls to *nextLabel*() (Algorithm 1), these local maximal label actually refer to the same global maximal label.

As we will be using pairs of labels with the *same* label creator, for the ease of presentation, we will be referring to these two variables as the *(label) pair*. The first label in a pair is called *ml*. The second label is called *cl* and it is either $\perp$, or equal to a label that cancels *ml* (i.e., *cl* indicates whether *ml* is an obsolete label or not). The variables and operators employed by Algorithm 2 are presented in Figure 4.2.

The processor state. Each processor stores an array of label pairs, $max_i[n]$, where $max_i[i]$ refers to p_i's maximal label pair and $max_i[j]$ considers the most recent value that p_i knows about p_j's pair. Processor p_i also stores the pairs of the most-recently-used labels in the array of queues $storedLabels_i[n]$. The j-th entry refers to the queue with pairs from p_j's domain, i.e., that were created by p_j. The algorithm makes sure that $storedLabels_i[j]$ includes only label pairs with unique *ml* from p_j's domain and that at most one of them is *legitimate*, i.e., not canceled. Queues $storedLabels_i[j]$ for $i \neq j$, have size $n + m$ whilst $storedLabels_i[i]$ has size $2(mn + 2n^2 - 2n)$ where m is the system's total link capacity in labels. We later show (c.f. Lemmas 4.3.3 and 4.3.4) that these queue sizes are sufficient to prevent overflows of useful labels.

High level description. Each pair of processors periodically exchange their maximal label pairs and the maximal label pair that they know of the recipient. Upon receipt of such a label pair couple, the receiving processor starts by checking the integrity of its data structures and upon finding a corruption it flushes its label history queues. It then moves to see whether the two labels that it received can cancel any of its non-canceled labels and if the received labels themselves can be canceled by labels that it has in its history. Upon finishing this label housekeeping, it tries to find its local maximal view, first among the non-cancelled labels that other processors report as maximal, and if not such exist among its own labels. In latter case, if no such label exists, it generates a new one with a call to Algorithm 1 and using its own label queue as input. At the end of the iteration the processor is guaranteed to have a maximal label, and continues to receive new label pair couples from other processors.

Information exchange between processors. Processor p_i takes a step whenever it receives two pairs $\langle sentMax, lastSent \rangle$ from some other processor. We note that in a legal execution p_j's pair includes both *sentMax*, which refers to p_j's maximal label

Variables:
$max[n]$ of $\langle ml, cl \rangle$: $max[i]$ is p_i's largest label pair, $max[j]$ refers to p_j's label pair (canceled when $max[j].cl \neq \perp$).
$storedLabels[n]$: an array of queues of the most-recently-used label pairs, where $storedLabels[j]$ holds the labels created by $p_j \in P$. For $p_j \in (P \setminus \{p_i\})$, $storedLabels[j]$'s queue size is limited to $(n + m)$ w.r.t. label pairs, where $n = |P|$ is the number of processors in the system and m is the maximum number of label pairs that can be in transit in the system. The $storedLabels[i]$'s queue size is limited to $(n(n^2 + m))$ pairs.
Operators:
The operator $add(\ell)$ adds lp to the front of the queue, and $emptyAllQueues()$ clears all $storedLabels[]$ queues. We use $lp.remove()$ for removing the record $lp \in storedLabels[]$. Note that an element is brought to the queue front every time this element is accessed in the queue.

Figure 4.2: Variables and Operators for the Labeling Scheme (Algorithm 2); code for p_i.

pair $max_j[j]$, and *lastSent*, which refers to a recent label pair that p_j received from p_i about p_i's maximal label, $max_j[i]$ (line 12).

Whenever a processor p_j sends a pair $\langle sentMax, lastSent \rangle$ to p_i, this processor stores the value of the arriving *sentMax* field in $max_i[j]$ (line 15). However, p_j may have local knowledge of a label from p_i's domain that cancels p_i's maximal label, *ml*, of the last received *sentMax* from p_i to p_j that was stored in $max_j[i]$. Then p_j needs to communicate this canceling label in its next communication to p_i. To this end, p_j assigns this canceling label to $max_j[i].cl$ which stops being $\perp$. Then p_j transmits $max_j[i]$ to p_i as a *lastSent* label pair, and this satisfies $lastSent.cl \not\preceq_{lb} lastSent.ml$, i.e., $lastSent.cl$ is either greater or incomparable to $lastSent.ml$. This makes *lastSent* illegitimate and in case this still refers to p_i's current maximal label, p_i must cancel $max_i[i]$ by assigning it with *lastSent* (and thus $max_i[i].cl = lastSent.cl$) as done in line 16. Processor p_i then processes the two pairs received (lines 17 to 24).

Label processing. Processor p_i takes a step whenever it receives a new pair message $\langle sentMax, lastSent \rangle$ from processor p_j (line 13). Each such step starts by removing *stale* information, i.e., misplaced or doubly represented labels (line 5). In the case that stale information exists, the algorithm empties the entire label storage. Processor p_i then tests whether the arriving two pairs are already included in the label storage ($storedLabels[]$), otherwise it includes them (line 18). The algorithm continues to see whether, based on the new pairs added to the label storage, it is possible to cancel a non-canceled label pair (which may well be the newly added pair). In this case, the algorithm updates the canceling field of any label pair lp (line 19) with the canceling

Algorithm 2: Practically-Self-Stabilizing Labeling Algorithm; code for processor p_i

```
1  Macros:
2  legit(lp) = (lp = ⟨•, ⊥⟩)
3  labels(lp) : return (storedLabels[lp.ml.lCreator])
4  double(j, lp) = (∃lp′ ∈ storedLabels[j] : ((lp ≠ lp′) ∧ ((lp =_ml
   lp′) ∨ (legit(lp) ∧ legit(lp′)))))
5  staleInfo() = (∃p_j ∈ P, lp ∈ storedLabels[j] : (lp ≠_lCreator j) ∨ double(j, lp))
6  recordDoesntExist(j) = (⟨max[j].ml, •⟩ ∉ labels(max[j]))
7  notgeq(j, lp) = if (∃lp′ ∈ storedLabels[j] : (lp′.ml ⋠_lb lp.ml)) then return(lp′.ml)
   else return(⊥)
8  canceled(lp) = if (∃lp′ ∈ labels(lp) : ((lp′ =_ml lp) ∧ ¬legit(lp′))) then return(lp′)
   else return(⟨⊥, ⊥⟩)
9  needsUpdate(j) = (¬legit(max[j]) ∧ ⟨max[j].ml, ⊥⟩ ∈ labels(max[j]))
10 legitLabels() = {max[j].ml : ∃p_j ∈ P ∧ legit(max[j])}
11 useOwnLabel() = if (∃lp ∈ storedLabels[i] : legit(lp)) then max[i] ← lp
       else storedLabels[i].add(max[i] ← ⟨nextLabel(), ⊥⟩) // For every
       lp ∈ storedLabels[i], we pass in nextLabel() both lp.ml and lp.cl.
12 upon transmitReady(p_j ∈ P \ {p_i}) do transmit(⟨max[i], max[j]⟩)
13 upon receive(⟨sentMax, lastSent⟩) from p_j
14 begin
15     max[j] ← sentMax;
16     if ¬legit(lastSent) ∧ max[i] =_ml lastSent then max[i] ← lastSent;
17     if staleInfo() then storedLabels.emptyAllQueues();
18     foreach p_j ∈ P : recordDoesntExist(j) do labels(max[j]).add(max[j]);
19     foreach p_j ∈ P, lp ∈ storedLabels[j] : (legit(lp) ∧ (notgeq(j, lp) ≠ ⊥)) do
       lp.cl ← notgeq(j, lp);
20     foreach p_j ∈ P, lp ∈ labels(max[j]) : (¬legit(max[j]) ∧ (max[j] =_ml lp) ∧ legit(lp))
       do lp ← max[j];
21     foreach p_j ∈ P, lp ∈ storedLabels[j] : double(j, lp) do lp.remove();
22     foreach p_j ∈ P : (legit(max[j]) ∧ (canceled(max[j]) ≠ ⟨⊥, ⊥⟩)) do
       max[j] ← canceled(max[j]);
23     if legitLabels() ≠ ∅ then max[i] ← ⟨max_{≺_lb}(legitLabels()), ⊥⟩;
24     else useOwnLabel();
```

label of a label pair lp' such that $lp'.ml \not\preceq_{lb} lp.ml$ (line 19). It is implied that since the two pairs belong to the same storage queue, they have the same processor as creator. The algorithm then checks whether any pair of the $max_i[]$ array can cause canceling to a record in the label storage (line 20), and also line 21 removes any canceled records that share the same creator identifier. The test also considers the case in which the above update may cancel any arriving label in $max[j]$ and updates this entry accordingly based on stored pairs (line 22).

After this series of tests and updates, the algorithm is ready to decide upon a maximal label based on its local information. This is the $\preceq_{lb}$-greatest legit label pair

among all the ones in $max_i[]$ with respect to their *ml* label (line 23). When no such legit label exists, p_i requests a legit label in its own label storage, $storedLabels_i[i]$, and if one does not exist, will create a new one if needed (line 24). This is done by passing the labels in the $storedLabels_i[i]$ queue to the *nextLabel*() function. Note that the returned label is coupled with a $\perp$ as the *cl* and the resulting label pair is added to both $max_i[i]$ and $storedLabel_i[i]$.

4.3.2 Labeling Algorithm Correctness Proof

We now show the correctness of the algorithm starting with a proof overview.

Proof overview. The proof considers a execution R of Algorithm 2 that may initiate in an arbitrary configuration (and include a processor that takes practically infinite number of steps). It starts by showing some basic facts, such as: (1) stale information is removed, i.e., $storedLabels_i[j]$ includes only unique copies of p_j's labels, and at most one legitimate such label (Corollary 4.3.1), and (2) p_i either adopts or creates the $\preceq_{lb}$-greatest legitimate local label (Lemma 4.3.2). The proof then presents bounds on the number adoption steps (Lemmas 4.3.3 and 4.3.4), that define the required queue sizes to avoid label overflows.

The proof continues to show that active processors can eventually stop adopting or creating labels, by tackling individual cases where canceled or incomparable label pairs may cause a change of the local maximal label. We show that such labels eventually disappear from the system (Lemma 4.3.5) and thus no new labels are being adopted or created (Lemma 4.3.6), which then implies the existence of a global maximal label (Lemma 4.3.7). Namely, there is a legitimate label $\ell_{\max}$, such that for any processor $p_i \in P$ (that takes a practically infinite number of steps in R), it holds that $max_i[i] = \ell_{\max}$. Moreover, for any processor $p_j \in P$ that is active throughout the execution, it holds that p_i's local maximal (legit) label pair $max_i[i] = \ell_{max}$ is the $\preceq_{lb}$-greatest of all the label pairs in $max_i[]$ and there is no label pair in $storedLabels_i[j]$ that cancels ℓ_{max}, i.e., $((max_i[j].ml \preceq_{lb} \ell_{\max}.ml) \wedge ((\forall \ell \in storedLabels_i[j] : legit(\ell)) \Rightarrow (\ell.ml \preceq_{lb} \ell_{\max}.ml)))$. We then demonstrate that, when starting from an initial arbitrary configuration, the system eventually reaches a configuration in which there is a global maximal label (Theorem 4.3.3).

Before we present the proof in detail, we provide some helpful definitions and

notation.

Definitions. We define $\mathcal{H}$ to be the set of all label pairs that can be in transit in the system, with $|\mathcal{H}| = m$. So in an arbitrary configuration, there can be up to m corrupted label pairs in the system's links. We also denote $\mathcal{H}_{i,j}$ as the set of label pairs that are in transit from processor p_i to processor p_j. The number of label pairs in $\mathcal{H}_{i,j}$ obeys the link capacity bound. Recall that the data structures used (e.g., max_i[], $storedLabels_i$[], etc) store label pairs. For convenience of presentation and when clear from the context, we may refer to the *ml* part of the label pair as "the label". Note that in this algorithm, we consider an *iteration* as the execution of lines 13–24, i.e., the receive action.

No stale information

Lemma 4.3.1 says that the predicate $staleInfo()$ (Fig. 4.2, line 5) can only hold during the first execution of the *receive*() event (line 13).

Lemma 4.3.1. *Let $p_i \in P$ be a processor for which $\neg staleInfo_i()$ (line 5) does not hold during the k-th step in R that includes the complete execution of the receive() event (from line 13 to 24). Then $k = 1$.*

Proof. Since R starts in an arbitrary configuration, there could be a queue in $storedLabels_i$[] that holds two label records from the same creator, a label that is not stored according to its creator identifier, or more than one legitimate label. Therefore, $staleInfo_i()$ might hold during the first execution of the *receive*() event. When this is the case, the $storedLabels_i$[] structure is emptied (line 17). During that *receive*() event execution (and any event execution after this), p_i adds records to a queue in $storedLabels_i$[] (according to the creator identifier) only after checking whether *recordDoesntExist*() holds (line 18).

Any other access to $storedLabels_i$[] merely updates cancelations or removes duplicates. Namely, canceling labels that are not the $\preceq_{lb}$-greatest among the ones that share the same creating processors (line 19) and canceling records that were canceled by other processors (line 20), as well as removing legitimate records that share the same *ml* (line 21). It is, therefore, clear that in any subsequent iteration of *receive*() (after the first), $staleInfo()$ cannot hold. □

Lemma 4.3.1 along with the lines 5 and 22 of the Algorithm, imply Corollary 4.3.1.

Corollary 4.3.1. *Consider a suffix R' of execution R that starts after the execution of a receive() event. Then the following hold throughout R': (i) $\forall p_i, p_j \in P$, the state of p_i encodes at most one legitimate label, $\ell_j =_{lCreator} j$ and (ii) ℓ_j can only appear in $storedLabels_i[j]$ and $max_i[]$ but not in $storedLabels_i[k] : k \neq j$.*

Local $\preceq_{lb}$-greatest legitimate local label

Lemma 4.3.2 considers processors for which $staleInfo()$ (Fig. 4.2, line 5) does not hold. Note that $\neg staleInfo()$ holds at any time after the first step that includes the *receive*() event (Lemma 4.3.1). Lemma 4.3.2 shows that p_i either adopts or creates the $\preceq_{lb}$-greatest legitimate local label pair and stores it in $max_i[i]$.

Lemma 4.3.2. *Let $p_i \in P$ be a processor such that $\neg staleInfo_i()$ (Fig. 4.2, line 5), and $L_{pre}(i) = \{max_i[j].ml : \exists p_j \in P \wedge legit(max_i[j]) \wedge (\exists \langle max_i[j].ml, x\rangle \in (labels(max_i[j]) \setminus \{max_i[j]\}) \Rightarrow (x = \perp))\}$ be the set of $max_i[]$'s labels that, before p_i executes lines 17 to 24, are legitimate both in $max_i[]$ and in $storedLabels_i[]$'s queues. Let $L_{post}(i) = \{max_i[j].ml : \exists p_j \in P \wedge legit(max_i[j])\}$ and $\langle \ell, \perp\rangle$ be the value of $max_i[i]$ immediately after p_i executes lines 17 to 24. The label $\langle \ell, \perp\rangle$ is the $\preceq_{lb}$-greatest legitimate label in $L_{post}(i)$. Moreover, suppose that $L_{pre}(i)$ has a $\preceq_{lb}$-greatest legitimate label pair, then that label pair is $\langle \ell, \perp\rangle$.*

Proof. **$\langle \ell, \perp\rangle$ is the $\preceq_{lb}$-greatest legitimate label pair in $L_{post}(i)$.** Suppose that immediately before line 23, we have that $legitLabels_i() \neq \emptyset$, where $legitLabels_i() = \{max_i[j].ml : \exists p_j \in P \wedge legit(max_i[j])\}$ (Fig. 4.2, line 10). Note that in this case $L_{post}(i) = legitLabels_i()$. By the definition of $\preceq_{lb}$-greatest legitimate label pair and line 23, $max_i[i] = \langle \ell, \perp\rangle$ is the $\preceq_{lb}$-greatest legitimate label pair in $L_{post}(i)$. Suppose that $legitLabels_i() = \emptyset$ immediately before line 23, i.e., there are no legitimate labels in $\{max_i[j] : \exists p_j \in P\}$. By the definition of $\preceq_{lb}$-greatest legitimate label pair and line 11 (Fig. 4.2), $max_i[i] = \langle \ell, \perp\rangle$ is the $\preceq_{lb}$-greatest legitimate label pair in $L_{post}(i)$.

Suppose that rec $= \langle \ell', \perp\rangle$ is a $\preceq_{lb}$-greatest legitimate label pair in $L_{pre}(i)$, then $\ell = \ell'$. We show that the record *rec* is not modified in $max_i[]$ until the end of the execution of lines 17 to 24. Moreover, the records that are modified in $max_i[]$, are not included in $L_{pre}(i)$ (it is canceled in $storedLabels_i[]$) and no records in $max_i[]$ become legitimate. Therefore, *rec* is also the $\preceq_{lb}$-greatest legitimate label pair in $L_{post}(i)$, and thus, $\ell = \ell'$.

Since we assume that $staleInfo_i()$ does not hold, line 17 does not modify *rec*. Lines 18, 19 and 21 might add, modify, and respectively, remove $storedLabels_i$'s

records, but it does not modify $max_i[]$. Since *rec* is not canceled in $storedLabels_i[]$ and the $\preceq_{lb}$-greatest legitimate label pair in $max_i[]$, the predicate $(legit(max[j]) \wedge notgeq(j))$ does not hold and line 19 does not modify *rec*. Moreover, the records in $max_i[]$, for which that predicate holds, become illegitimate. □

Bounding the number of labels

Lemmas 4.3.3 and 4.3.4 present bounds on the number of adoption steps. These are $n + m$ for labels by labels that become inactive in any point in R and $(mn + 2n^2 - 2n)$ for any active processor. Following the above, choosing the queue sizes as $n + m$ for $storedLabels_i[j]$ if $i \neq j$, and $2(nm + 2n^2 - 2n) + 1$ for $storedLabels_i[i]$ is sufficient to prevent overflows given that m is the system's total link capacity in labels.

Maximum number of label adoptions in the absence of creations. Suppose that there exists a processor, p_j, that has stopped adding labels to the system (the else part of line 24), say, because it became inactive (crashed), or it names a maximal label that is the $\preceq_{lb}$-greatest label pair among all the ones that the network ever delivers to p_j. Lemma 4.3.3 bounds the number of labels from p_j's domain that any processor $p_i \in P$ adopts in R.

Lemma 4.3.3. *Let $p_i, p_j \in P$, be two processors. Suppose that p_j has stopped adding labels to the system configuration (the else part of line 24), and sending (line 12) these labels during R. Processor p_i adopts (line 23) at most $(n + m)$ labels, $\ell_j : (\ell_j =_{lCreator} j)$, from p_j's unknown domain ($\ell_j \notin labels_i(\ell_j)$) where m is the maximum number of label pairs that can be in transit in the system.*

Proof. Let $p_k \in P$. At any time (after the first step in R) processor p_k's state encodes at most one legitimate label, ℓ_j, for which $\ell_j =_{lCreator} j$ (Corollary 4.3.1). Whenever p_i adopts a new label ℓ_j from p_j's domain (line 23) such that $\ell_j : (\ell_j =_{lCreator} j)$, this implies that ℓ_j is the only legitimate label pair in $storedLabels_i[j]$. Since ℓ_j was not transmitted by p_j before it was adopted, ℓ_j must come from p_k's state delivered by a transmit event (line 12) or delivered via the network as part of the set of labels that existed in the initial arbitrary state. The bound holds since there are n processors, such as p_k, and m bounds the number of labels in transit. Moreover, no other processor can create label pairs from the domain of p_j. □

Maximum number of label creations. Lemma 4.3.4 shows a bound on the number of adoption steps that does not depend on whether the labels are from the domain of an active or (eventually) inactive processor.

Lemma 4.3.4. *Let $p_i \in P$ and $L_i = \ell_{i_0}, \ell_{i_1}, \ldots$ be the sequence of legitimate labels, $\ell_{i_k} =_{lCreator} i$, from p_i's domain, which p_i stores in $max_i[i]$ through the reception (line 13) or creation of labels (line 24), where $k \in \mathbb{N}$. It holds that $|L_i| \leq n(n^2 + m)$.*

Proof. Let $L_{i,j} = \ell_{i_0,j}, \ell_{i_1,j}, \ldots$ be the sequence of legitimate labels that p_i stores in $max_i[j]$ during R and $C_{i,j} = \ell^c_{i_0,j}, \ell^c_{i_1,j}, \ldots$ the sequence of legit labels that p_i receives from p_j's domain. We consider the following cases in which p_i stores L's values in $max_i[i]$.

(1) When $\ell_{\mathbf{i_k}} = \ell_{\mathbf{j_0,j'}}$, where $\mathbf{p_j}, \mathbf{p_{j'}} \in \mathbf{P}$ and $\mathbf{k} \in \mathbb{N}$. This case considers the situation in which $max_i[i]$ stores a label that appeared in $max_j[j']$ at the (arbitrary) starting configuration, (i.e. $\ell_{j_0,j'} \in L_{j,j'}$). There are at most $n(n-1)$ such legitimate label values from p_i's domain, namely $n-1$ arrays $max_j[]$ of size n.

(2) When $\ell_{\mathbf{i_k}} = \ell_{\mathbf{j_{k'},j'}} = \ell^{\mathbf{c}}_{\mathbf{j_0,j'}}$, where $\mathbf{p_j}, \mathbf{p_{j'}} \in \mathbf{P}$, $\mathbf{k}, \mathbf{k'} \in \mathbb{N}$ and $\ell_{\mathbf{j_{k'},j'}} \neq \ell_{\mathbf{j_{k'},j}}$. This case considers the situation in which $max_i[i]$ stores a label that appeared in the communication channel between p_j and $p_{j'}$ at the (arbitrary) starting configuration, (i.e. $\ell^c_{j_0,j'} \in C_{j,j'}$) and appeared in $max_j[j']$ before p_j communicated this to p_i. There are at most m such values, i.e., as many as the capacity of the communication links in labels, namely $|\mathcal{H}|$.

(3) When $\ell_{\mathbf{i_k}}$ is the return value of nextLabel() (the else part of line 24). Processor p_i aims at adopting the $\preceq_{lb}$-greatest legitimate label pair that is stored in $max_i[]$, whenever such exists (line 23). Otherwise, p_i uses a label from its domain; either one that is the $\preceq_{lb}$-greatest legit label pair among the ones in $storedLabels_i[i]$, whenever such exists, or the returned value of *nextLabel*() (line 24).

The latter case (the else part of line 24) refers to labels, ℓ_{i_k}, that p_i stores in $max_i[i]$ only after checking that there are no legitimate labels stored in $max_i[]$ or $storedLabels_i[i]$. Note that every time p_i executes the else part of line 24, p_i stores the returned label, ℓ_{i_k}, in $storedLabels_i[i]$. After that, there are only three events for ℓ_{i_k} not to be stored as a legitimate label in $storedLabels_i[i]$:

(*i*) execution of line 17, (*ii*) the network delivers to p_i a label, ℓ', that either cancels ℓ_{i_k} ($\ell'.cl \not\preceq_{lb} \ell_{i_k}.ml$), or for which $\ell'.ml \not\preceq_{lb} \ell_{i_k}.ml$, and (*iii*) ℓ_{i_k} overflows from $storedLabels_i[i]$ after exceeding the $(n(n^2 + m) + 1)$ limit which is the size of the queue.

Note that Lemma 4.3.1 says that event (*i*) can occur only once (during p_i's first

step). Moreover, only p_i can generate labels that are associated with its domain (in the else part of line 24). Each such label is $\preceq_{lb}$-greater-equal than all the ones in $storedLabels_i[i]$ (by the definition of *nextLabel*() in Algorithm 1).

Event (*ii*) cannot occur after p_i has learned all the labels $\ell \in remoteLabels_i$ for which $\ell \notin storedLabels_i[i]$, where $remoteLabels_i = (((\cup_{p_j \in P} localLabels_{i,j}) \cup \mathcal{H}) \setminus storedLabels_i[i])$ and $localLabels_{i,j} = \{\ell' : \ell' =_{lCreator} i, \exists p_j \in P : ((\ell' \in storedLabels[i]) \vee (\exists\, p_k \in P : \ell' = max_j[k].ml))\}$. During this learning process, p_i cancels or updates the cancellation labels in $storedLabels_i[i]$ before adding a new legitimate label. Thus, this learning process can be seen as moving labels from $remoteLabels_i$ to $storedLabels_i[i]$ and then keeping at most one legitimate label available in $storedLabels_i[i]$. Every time $storedLabels_i[i]$ accumulates a label ℓ that was unknown to p_i, the use of *nextLabel*() allows it to create a label ℓ_{i_k} that is $\preceq_{lb}$-greater than any label pair in $storedLabels_i[i]$ and eventually from all the ones in $remoteLabels_i$.

Note that $remoteLabels_i$'s labels must come from the (arbitrary) start of the system, because p_i is the only one that can add a label to the system from its domain and therefore this set cannot increase in size. These labels include those that are in transit in the system and all those that are unknown to p_i but exist in the $max_j[\bullet]$ or $storedLabels_j[i]$ structures of some other processor p_j. By Lemma 4.3.3 we know that $|storedLabels_j[i]| \leq n + m$ for $i \neq j$. From the three cases of L_i labels that we detailed at the beginning of this proof ((1)–(3)), we can bound the size of $remoteLabels_i$ as follows: for $p_j \in P : j \neq i$ we have that $|remoteLabels_i| \leq (n-1)(|max[]| + |storedLabels_j[i]|) + |\mathcal{H}| = (n-1)(n + (n+m)) + m = mn + 2n^2 - 2n$. Since p_i may respond to each of these labels with a call to *nextLabel*(), we require that $storedLabels_i[i]$ has size $2|remoteLabels_i| + 1$ label pairs in order to be able to accommodate all the labels from $|remoteLabels_i|$ and the ones created in response to these, plus the current greatest. Thus, what is suggested by event (*ii*) of p_i, i.e., receiving labels from $remoteLabels_i$, stops happening before overflows (event (*iii*)) occurs, since $storedLabels_i[i]$ has been chosen to have a size that can accommodate all the labels from $remoteLabels_i$ and those created by p_i as a response to these. This size is $2(mn + 2n^2 - 2n) + 1 = 2(n^3 cap + 2n^2 - 2n) + 1$ (since $m = n^2 cap$) which is $O(n^3)$. □

From the end of the proof of Lemma 4.3.4, we get Corollary 4.3.2.

Corollary 4.3.2. *The number k of antistings needed by Algorithm 1 is* $2 \cdot (2(n^3 cap + 2n^2 - 2n) + 1)$ *(twice the queue size).*

Pair diffusion

The proof continues and shows that active processors can eventually stop adopting or creating labels. We are particularly interested in looking into cases in which there are canceled label pairs and incomparable ones. We show that they eventually disappear from the system (Lemma 4.3.5) and thus no new labels are being adopted or created (Lemma 4.3.6), which then implies the existence of a global maximal label (Lemma 4.3.7).

Lemmas 4.3.5 and 4.3.6, as well as Lemma 4.3.7 and Theorem 4.3.3 assume the existence of at least one processor, $p_{unknown} \in P$ whose identity is unknown, that takes practically infinite number of steps in R. Suppose that processor $p_i \in P$ takes a bounded number of steps in R during a period in which $p_{unknown}$ takes a practically infinite number of steps. We say that p_i has become inactive (crashed) during that period and assume that it does not resume to take steps at any later stage of R (in the manner of fail-stop failures, as defined in Chapter 3).

Consider a processor $p_i \in P$ that takes any number of (bounded or practically infinite) steps in R and two processors $p_j, p_k \in P$ that take a practically infinite number of steps in R. Given that p_j has a label pair ℓ as its local maximal, and there exists another label pair ℓ' such that $(\ell'.ml \not\preceq_{lb} \ell.ml) \vee \ell'.cl \not\preceq_{lb} \ell.ml$ and they have the same creator p_i. Algorithm 2 suggests only two possible routes for some label pair ℓ' to find its way in the system through p_j. Either by p_j adopting ℓ' (line 23), or by creating it as a new label (the else part of line 24). Note, however, that p_j is not allowed to create a label in the name of p_i and since $\ell' =_{lCreator} i$, the only way for ℓ' to disturb the system is if this is adopted by p_j as in line 23. We use the following definitions for estimating whether there are such label pairs as ℓ and ℓ' in the system.

There is a *risk* for two label pairs from p_i's domain, ℓ_j and ℓ_k, to cause such a disturbance when either they cancel one another or when it can be found that one is not greater than the other. Thus, we use the predicate $risk_{i,j,k}(\ell_j, \ell_k) = (\ell_j =_i \ell_k) \wedge legit(\ell_j) \wedge (notGreater(\ell_j, \ell_k) \vee canceled(\ell_j, \ell_k))$ to estimate whether p_j's state encodes a label pair, $\ell_j =_{lCreator} i$, from p_i's domain that may disturb the system due to another label, ℓ_k, from p_i's domain that p_k's state encodes, where $canceled(\ell_j, \ell_k) = (legit(\ell_j) \wedge \neg legit(\ell_k) \wedge \ell_j =_{ml} \ell_k)$ refers to a case in which label ℓ_j is canceled by label ℓ_k, $notGreater(\ell_j, \ell_k) = (legit(\ell_j) \wedge legit(\ell_k) \wedge \ell_k.ml \not\preceq_{lb} \ell_j.ml)$ that refers to a case in which label ℓ_k is not $\preceq_{lb}$-greater than ℓ_j and $(\ell_j =_i \ell_k) \equiv (\ell_j =_{lCreator} \ell_k =_{lCreator} i)$.

Notation	Definition	Remark
$hName_{i,j,k}$	$\{(\ell_j, \ell_k) : \ell_j = max_j[j] \wedge (\exists \langle \ell_k, \bullet \rangle \in \mathcal{H}_{k,j})\}$	In transit from p_k to p_j as *sentMax* feedback about $max_k[k]$
$hAck_{i,j,k}$	$\{(\ell_j, \ell_k) : \ell_j = max_j[k] \wedge (\exists \langle \bullet, \ell_k \rangle \in \mathcal{H}_{k,j})\}$	In transit from p_k to p_j as *lastSent* feedback about $max_k[j]$
$max_{i,j,k}$	$\{(max_j[j], max_k[k])\}$	Local maximal labels of p_j and p_k
$ack_{i,j,k}$	$\{(max_j[j], max_k[j])\}$	ℓ_j is p_j's local maximal label and $\ell_k = max_k[j]$
$stored_{i,j,k}$	$\{\{max_j[j]\} \times storedLabels_k[i]\}$	A label ℓ_k in $storedLabels_k[i]$ that can cancel $\ell_j = max_j[j]$

Table 4.1: The notation used to identify the possible positions of label pairs ℓ_j and ℓ_k that can cause canceling as used in Lemmas 4.3.5 to 4.3.7 and in Theorem 4.3.3.

These two label pairs, ℓ_j and ℓ_k, can be the ones that processors p_j and p_k name as their local maximal label, as in $max_{i,j,k} = \{(max_j[j], max_k[k])\}$, or recently received from one another, as in $ack_{i,j,k} = \{(max_j[j], max_k[j])\}$. These two cases also appear when considering the communication channel (or buffers) from p_k to p_j, as in $hName_{i,j,k} = \{(\ell_j, \ell_k) : \ell_j = max_j[j] \wedge (\exists \langle \ell_k, \bullet \rangle \in \mathcal{H}_{k,j})\}$ and $hAck_{i,j,k} = \{(\ell_j, \ell_k) : \ell_j = max_j[k] \wedge (\exists \langle \bullet, \ell_k \rangle \in \mathcal{H}_{k,j})\}$. We also note the case in which p_k stores a label pair that might disturb the one that p_j names as its (local) maximal, as in $stored_{i,j,k} = \{\{max_j[j]\} \times storedLabels_k[i]\}$ We define the union of these cases to be the set $risk = \{(\ell_j, \ell_k) \in max_{i,j,k} \cup ack_{i,j,k} \cup hName_{i,j,k} \cup hAck_{i,j,k} \cup stored_{i,j,k} : \exists p_i, p_j, p_k \in P \wedge stopped_j \wedge stopped_k \wedge risk_{i,j,k}(\ell_j, \ell_k)\}$, where $stopped_i = true$ when processor p_i is inactive (crashed) and $false$ otherwise. The above notation can also be found in Table 4.1.

Lemma 4.3.5. *Suppose that there exists at least one processor, $p_{unknown} \in P$ whose identity is unknown, that takes practically infinite number of steps in R during a period where p_j never adopts labels (line 23), $\ell_j : (\ell_j =_{lCreator} i)$, from p_i's unknown domain ($\ell_j \notin labels_j(\ell_j)$). Then eventually $risk = \emptyset$.*

Proof. Suppose this Lemma is false, i.e., the assumptions of this Lemma hold and yet in any configuration $c \in R$, it holds that $(\ell_j, \ell_k) \in risk \neq \emptyset$. We use *risk*'s definition to study the different cases. By the definition of *risk*, we can assume, without the loss of generality, that p_j and p_k are active throughout R.

Claim: If p_j and p_k are active throughout R, i.e. $stopped_j = stopped_k =$ `False`, then

$risk \neq \emptyset \iff risk_{i,j,k} = \texttt{True}$. This means that there exist two label pairs (ℓ_j, ℓ_k) where ℓ_k can force a cancellation to occur. Then the only way for this two labels to force $risk \neq \emptyset$ is if, throughout the execution, ℓ_k never reaches p_j.

The above claim is verified by a simple observation of the algorithm. If ℓ_k reaches p_j then lines 16, 20 and 22 guarantee a canceling and lines 18 and 19 ensure that these labels are kept canceled inside $storedLabels_j[]$. The latter is also ensured by the bounds on the labels given in Lemmas 4.3.3 and 4.3.4 that do not allow queue overflows. Thus to include these two labels to *risk*, is to keep ℓ_k hidden from p_j throughout R. We perform a case-by-case analysis to show that it is impossible for label ℓ_k to be "hidden" from p_j for an infinite number of steps in R.

The case of $(\ell_j, \ell_k) \in hName_{i,j,k}$**.** This is the case where $\ell_j = max_j[j]$ and ℓ_k is a label in $\mathcal{H}_{k,j}$ that appears to be $max_k[k]$. This may also contain such labels from the corrupt state. We note that p_j and p_k are active throughout R. The stabilizing implementation of the data-link ensures that a message cannot reside in the communication channel during an infinite number of *transmit*() – *receive*() events of the two ends. Thus ℓ_k, which may well have only a single instance in the link coming from the initial corrupt state, will either eventually reach p_j or it become lost. In the both cases (the first by the Claim for the second trivially) the two clashing labels are removed from *risk* and the result follows.

The case of $(\ell_j, \ell_k) \in hAck_{i,j,k}$**.** This is the case where $\ell_j = max_j[j]$ and ℓ_k is a label in $\mathcal{H}_{k,j}$ that appears to be $max_k[j]$. The proof line is exactly the same as the previous case.

This case follows by the same arguments to the case of $(\ell_j, \ell_k) \in ack_{i,j,k}$.

The case of $(\ell_j, \ell_k) \in max_{i,j,k}$**.** Here the label pairs ℓ_j and ℓ_k are named by p_j and p_k as their local maximal label. We note that p_j and p_k are active throughout R. By our self-stabilizing data-links and by the assumption on the communication that a message sent infinitely often is received infinitely often, then p_k transmits its $max_k[k]$ label infinitely often when executing line 12. This implies that p_j receives ℓ_k infinitely often. By the Claim the canceling takes place, and the two labels are eventually removed from the global observer's *risk* set, giving a contradiction.

The case of $(\ell_\mathbf{j}, \ell_\mathbf{k}) \in \mathbf{ack_{i,j,k}}$**.** This is the case case where the labels (ℓ_j, ℓ_k) belong to $\{(max_j[j], max_k[j])\}$. Since processor p_k continuously transmits its label pair in $max_k[j]$ (line 12) the proof is almost identical to the previous case.

The case of $(\ell_j, \ell_k) \in \mathbf{stored_{i,j,k}}$. This case's proof, follows by similar arguments to the case of $(\ell_j, \ell_k) \in max_{i,j,k}$. Namely, p_k eventually receives the label pair $\ell_j = max_j[j]$. The assumption that $risk_{i,j,k}(\ell_j, \ell_k)$ holds implies that one of the tests in lines 19 and 22 will either update $storedLabels_k[i]$, and respectively, $max_k[j]$ with canceling values. We note that for the latter case we argue that p_j eventually received the canceled label pair in $max_k[j]$, because we assume that p_j does not change the value of $max_j[j]$ throughout R.

By careful and exhaustive examination of all the cases, we have proved that there is no way to to keep ℓ_k hidden from p_j throughout R. This is a contradiction to our initial assumption, and thus eventually $risk = \emptyset$. □

These two label pairs, ℓ_j and ℓ_k, can be the ones that processors p_j and p_k name as their local maximal label, as in $max_{i,j,k} = \{(max_j[j], max_k[k])\}$, or recently received from one another, as in $ack_{i,j,k} = \{(max_j[j], max_k[j])\}$. These two cases also appear when considering the communication channel (or buffers) from p_k to p_j, as in $hName_{i,j,k} = \{(\ell_j, \ell_k) : \ell_j = max_j[j] \wedge (\exists \langle \ell_k, \bullet \rangle \in \mathcal{H}_{k,j})\}$ and $hAck_{i,j,k} = \{(\ell_j, \ell_k) : \ell_j = max_j[j] \wedge (\exists \langle \bullet, \ell_k \rangle \in \mathcal{H}_{k,j})\}$. We also note the case in which p_k stores a label pair that might disturb the one that p_j names as its (local) maximal, as in $stored_{i,j,k} = \{\{max_j[j]\} \times storedLabels_k[i]\}$.

Lemma 4.3.6. *Suppose that $risk = \emptyset$ in every configuration throughout R and that there exists at least one processor, $p_{unknown} \in P$ whose identity is unknown, that takes practically infinite number of steps in R. Then p_j* never *adopts labels (line 23), $\ell_j : (\ell_j =_{lCreator} i)$, from p_i's unknown domain ($\ell_j \notin labels_j(\ell_j)$).*

Proof. Note that the definition of *risk* considers almost every possible combination of two label pairs ℓ_j and ℓ_k from p_i's domain that are stored by processor p_j, and respectively, p_k (or in the channels to them). The only combination that is not considered is $(\ell_j, \ell_k) \in storedLabels_j[i] \times storedLabels_k[i]$. However, this combination can indeed reside in the system during a legal execution and it cannot lead to a disruption for the case of $risk = \emptyset$ in every configuration throughout R because before that could happen, either p_j or p_k would have to adopt ℓ_j, and respectively, ℓ_k, which means a contradiction with the assumption that $risk = \emptyset$.

The only way that a label in $storedLabels[]$ can cause a change of the local maximum label and be communicated to also disrupt the system, is to find its way to $max[]$. Note that p_j cannot create a label under p_i's domain (line 24) since the algorithm does not allow this, nor can it adopt a label from $storedLabels_j[i]$ (by the definition of

legitLabels(), Fig. 4.2, line 10). So there is no way for ℓ_j to be added to $max_j[j]$ and thus make $risk \neq \emptyset$ through creation or adoption.

On the other hand, we note that there is only one case where p_k extracts a label from $storedLabels_k[i] : i \neq k$ and adds it to $max_k[j]$. This is when it finds a legit label $\ell_j \in max_k[j]$ that can be canceled by some other label ℓ_k in $storedLabels_k[i]$), line 22. But this is the case of having the label pair (ℓ_j, ℓ_k) in $stored_{i,j,k}$. Our assumption that $risk = \emptyset$ implies that $stored_{i,j,k} = \emptyset$. This is a contradiction. Thus a label ℓ_k cannot reach $max_k[]$ in order for it to be communicated to p_j.

In the same way we can argue for the case of two messages in transit, $\mathcal{H}_{j,k} \times \mathcal{H}_{k,j}$ and that $risk = \emptyset$ throughout R. □

Lemma 4.3.7. *Suppose that $risk = \emptyset$ in every configuration throughout R and that there exists at least one processor, $p_{unknown} \in P$ whose identity is unknown, that takes practically infinite number of steps in R. There is a legitimate label $\ell_{\max}$, such that for any processor $p_i \in P$ (that takes a practically infinite number of steps in R), it holds that $max_i[i] = \ell_{\max}$. Moreover, for any processor $p_j \in P$ (that takes a practically infinite number of steps in R), it holds that $((max_i[j].ml \preceq_{lb} \ell_{\max}.ml) \wedge ((\forall \ell \in storedLabels_i[j] : legit(\ell)) \Rightarrow (\ell.ml \preceq_{lb} \ell_{\max}.ml)))$.*

Proof. We initially note that the two processors p_i, p_j that take an infinite number of steps in R will exchange their local maximal label $max_i[i]$ and $max_j[j]$ an infinite number of times. By the assumption that $risk = \emptyset$, there are no two label pairs in the system that can cause canceling to each other that are unknown to p_i or p_j and are still part of $max_i[i]$ or $max_i[j]$. Hence, any differences in the local maximal label of the processors must be due to the labels' *lCreator* difference.

Since $max_i[i]$ and $max_j[j]$ are continuously exchanged and received, assuming $max_i[i].ml \prec_{lb} max_j[j].ml$ where the labels are of different label creators, then p_i will be led to a *receive*() event of $\langle sentMax_j, lastSent_j \rangle$ where $max_i[i].ml \prec_{lb} sentMax_j.ml$. By line 15, $sentMax_j$ is added to $max_i[j]$ and since $risk = \emptyset$ no action from line 16 to line 22 takes place. Line 23 will then indicate that the greatest label in $max_i[\bullet]$ is that in $max_i[j]$ which is then adopted by p_i as $max_i[i]$, i.e., p_i's local maximal. The above is true for every pair of processors taking an infinite number of steps in R and so we reach to the conclusion that eventually all such processors converge to the same ℓ_{max} label, i.e., it holds that $((max_i[j].ml \preceq_{lb} \ell_{\max}.ml) \wedge ((\forall \ell \in storedLabels_i[j] : legit(\ell)) \Rightarrow (\ell.ml \preceq_{lb} \ell_{\max}.ml)))$. □

Convergence

Theorem 4.3.3 combines all the previous lemmas to demonstrate that when starting from an arbitrary starting configuration, the system eventually reaches a configuration in which there is a global maximal label.

Theorem 4.3.3. *Suppose that there exists at least one processor, $p_{unknown} \in P$ whose identity is unknown, that takes practically infinite number of steps in R. Within a bounded number of steps, there is a legitimate label pair $\ell_{\max}$, such that for any processor $p_i \in P$ (that takes a practically infinite number of steps in R), it holds that p_i has $max_i[i] = \ell_{\max}$. Moreover, for any processor $p_j \in P$ (that takes a practically infinite number of steps in R), it holds that* $((max_i[j].ml \preceq_{lb} \ell_{\max}.ml) \wedge ((\forall \ell \in storedLabels_i[j] : legit(\ell)) \Rightarrow (\ell.ml \preceq_{lb} \ell_{\max}.ml)))$.

Proof. For any processor in the system, which may take any (bounded or practically infinite) number of steps in R, we know that there is a bounded number of label pairs, $L_i = \ell_{i_0}, \ell_{i_1}, \ldots$, that processor $p_i \in P$ adds to the system configuration (the else part of line 24), where $\ell_{i_k} =_{lCreator} i$ (Lemma 4.3.4). Thus, by the pigeonhole principle we know that, within a bounded number of steps in R, there is a period during which $p_{unknown}$ takes a practically infinite number of steps in R whilst (all processors) p_i do not add any label pair, $\ell_{i_k} =_{lCreator} i$, to the system configuration (the *else* part of line 24).

During this practically infinite period (with respect to $p_{unknown}$), in which no label pairs are added to the system configuration due to the *else* part of line 24, we know that for any processor $p_j \in P$ that takes any number of (bounded or practically infinite) steps in R, and processor $p_k \in P$ that adopts labels in R (line 23), $\ell_j : (\ell_j =_{lCreator} j)$, from p_j's unknown domain ($\ell_j \notin storedLabels_k(j)$) it holds that p_k adopts such labels (line 23) only a bounded number times in R (Lemma 4.3.3). Therefore, we can again follow the pigeonhole principle and say that there is a period during which $p_{unknown}$ takes a practically infinite number of steps in R whilst neither p_i adds a label, $\ell_{i_k} =_{lCreator} i$, to the system (the else part of line 24), nor p_k adopts labels (line 23), $\ell_j : (\ell_j =_{lCreator} j)$, from p_j's unknown domain ($\ell_j \notin labels_k(\ell_j)$).

We deduce that, when the above is true, then we have reached a configuration in R where $risk = \emptyset$ (Lemma 4.3.5) and remains so throughout R (Lemma 4.3.6). Lemma 4.3.7 concludes by proving that, whilst $p_{unknown}$ takes a practically infinite number of steps, all processors (that take practically infinite number of steps in R) name the same $\preceq_{lb}$-greatest legitimate label pair which the theorem statement

Variables: A label *lbl* is extended to the triple $\langle lbl, seqn, wid \rangle$ called a *counter* where *seqn*, is the sequence number related to *lbl*, and *wid* is the identifier of the creator of this *seqn*. A counter pair $\langle mct, cct \rangle$ extends a label pair. *cct* is a canceling counter for *mct*, such that $cct.lbl \not\prec_{lb} mct.lbl$ or $cct.lbl = \perp$. We rename structures *max*[] and *storedLabels*[] of Alg. 2 to *maxC*[] and *storedCnts*[] that hold counter pairs instead of label pairs. Variable $status \in \{\mathsf{MAX\_REQUEST}, \mathsf{MAX\_WRITE}, \mathsf{COMPLETE}\}$.
Operators: *add*(*ctp*) - places a counter pair *ctp* at the front of a queue. If *ctp.mct.lbl* already exists in the queue, it only maintains the instance with the greatest counter w.r.t. $\prec_{ct}$, placing it at the front of the queue. If one counter pair is canceled then the canceled copy is retained. We consider an array field as a single sized queue and use *add*().

Figure 4.3: Variables and Operators for Counter Increment; code for p_i.

specifies. Thus no label $\ell =_{lCreator} j$ in $max_i[\bullet]$ or in $storedLabels_i[j]$ may satisfy $\ell.ml \not\preceq_{lb} \ell_{max}.ml$. □

Algorithm complexity

The required local memory of a processor comprises of a queue of size (in labels) $2(n^3cap+2n^2-2n)$ that hosts the labels with the processor as a creator (Corollary 4.3.2). The local state also includes $n-1$ queues of size $n+n^2cap$ to store labels by other processors, and a single label for the maximal label of every processor. We conclude that the *space complexity* is of order $O(n^3)$ in labels. Given the number of possible labels in the system by the same processor is $\beta = n^3cap + 2n^2 - 2n$, as shown in the proof of Lemma 4.3.4, we deduce that the size of a label in bits is $O(\beta \log \beta)$.

By Theorem 4.3.3 we can bound the *stabilization time* based on the number of label creations. Namely, in an execution with $O(n \cdot \beta)$ label creations (e.g., up to n processors can create $O(\beta)$ labels), there is a practically infinite execution suffix (of size 2^τ iterations) where the receipt of a label which starts an iteration never changes the maximal label of any processor in the system.

4.3.3 Increment Counter Algorithm

We adjust the labeling algorithm to work with counters, so that our counter increment algorithm is a stand-alone algorithm. In this subsection, we explain how we can enhance the labeling scheme presented in the previous subsection to obtain a practically (infinite) self-stabilizing counter increment algorithm.

// Where macros coincide with Algorithm 2 we do not restate them.

1 **Macros:**

2 $exhausted(ctp) = (ctp.mct.seqn \geq 2^{\tau})$

3 $cancelExh(ctp) : ctp.cct \leftarrow ctp.mct$

4 $cancelExhMaxC() :$ **foreach** $p_j \in P,\ c \in maxC[j] : exhausted(c)$ **do** $cancelExh(maxC[j])$;

5 $legit(ctp) = (ctp.cct = \perp\rangle)$

6 $staleCntrInfo() = staleInfo() \vee (\exists p_j \in P, x \in storedCnts[j] : exhausted(x) \wedge legit(x))$

7 $retCntrQ(ct)$: **return** $(storedCnts[ct.lbl.lCreator])$

8 $retMaxCnt(ct) =$ **return** $(max_{\prec_{ct}}(ct, ct'))$ **where** $ct' \in retCntrQ(ct) \wedge (ct =_{lbl} ct')$

9 $legitCnts() = \{maxC[j].mct : \exists p_j \in P \wedge legit(maxC[j])\}$

10 $useOwnCntr() =$ **if** $(\exists cp \in storedCnts[i] : legit(lp))$ **then** $maxC[i] \leftarrow cp$
else $storedCnts[i].add(maxC[i] \leftarrow \langle\langle nextLabel(), 0, i\rangle, \perp\rangle)$ // For every $cp \in storedCnts[i]$, we pass to $nextLabel()$ both $cp.mct.lbl$ and $cp.cct.lbl$.

11 $getMaxSeq()$: **return** $max_{wid}(\{max_{seqn}(\{ctp : ctp.mct \in legitCnts() \wedge maxC[i] =_{mct.lbl} ctp\})\})$

12 $initWrite = \{\langle maxC[i], responseSet, status\rangle \leftarrow \langle maxC[i](), \emptyset, \mathsf{MAX\_WRITE}\rangle;\}$

13 $increment() = \{maxC[i] \leftarrow \langle maxC[i].mct.lbl, maxC[i].mct.seqn + 1, i\rangle;\}$

14 $correctResponse(A, B) =$ **return** $((status = \mathsf{MAX\_REQUEST} \wedge (A, B \notin \{\perp\})) \vee ((status = \mathsf{MAX\_WRITE}) \wedge (\langle A, B\rangle = \langle \perp, maxC_i[i]\rangle)))$

Figure 4.4: Macros for the Increment Counter (Algorithm 3).

From labels to counters and to a counter version of Algorithm 2

Counters. To achieve this task, we now need to work with practically unbounded *counters*. A counter *cnt* is a triplet $\langle lbl, seqn, wid\rangle$, where *lbl* is an epoch label as defined in the previous subsection, *seqn* is a τ-bit integer sequence number and *wid* is the identifier of the processor that last incremented the counter's sequence number, i.e., *wid* is the counter *writer*. Then, given two counters cnt_i, cnt_j we define the relation $cnt_i \prec_{ct} cnt_j \equiv (cnt_i.lbl \prec_{lb} cnt_j.lbl) \vee ((cnt_i.lbl = cnt_j.lbl) \wedge (cnt_i.seqn < cnt_j.seqn)) \vee ((cnt_i.lbl = cnt_j.lbl) \wedge (cnt_i.seqn = cnt_j.seqn) \wedge (cnt_i.wid < cnt_j.wid))$. Observe that when the labels of the two counters are incomparable, the counters are also incomparable.

The relation $\prec_{ct}$ defines a total order (as required by practically unbounded counters) for counters with the same label, thus, only when processors share a globally maximal label. Conceptually, if the system stabilizes to use a global maximal label, then the pair of the sequence number and the processor identifier (of this sequence number) can be used as an unbounded counter, as used, for example, in MWMR register implementations [17, 34].

Structures. We convert the label structures *max*[] and *storedLabels*[] of the labeling algorithm into the structures *maxC*[] and *storedCnts*[] that hold counters rather than labels (see Figure 4.3). Each label can yield many different counters with different $\langle seqn, wid\rangle$. Therefore, in order to avoid increasing the size of the queues of *storedCnts*

(with respect to the number of elements stored), we only keep the highest sequence number observed for each label (breaking ties with the *wid*).

This is encapsulated in the definition of the *add*() operator (Figure 4.3 – Operators). In particular, we define the operator *add*(*ctp*) (Fig. 4.3) to enqueue a counter pair *ctp* to a queue of *storedCnts*[*n*], where in case a counter with the same label already exists, the following two rules apply: (1) if at least one of the two counters is canceled we keep a canceled instance, and (2) if both counters are legitimate, we keep the greatest counter with respect to $\langle seqn, wid \rangle$. The counter is placed at the front of the queue. In this way we allow for labels for which the counters have not been exhausted to be reused. We denote a counter pair by $\langle mct, cct \rangle$, with this being the extension of a label pair $\langle ml, cl \rangle$, where *cct* is a canceling counter for *mct*, such that either $cct.lbl \prec_{lb} mct.lbl$ (i.e., the counter is canceled), or $cct.lbl = \perp$.

Exhausted counters. These are the ones satisfying $seqn \geq 2^{\tau}$, and they are treated in a way similar to the canceled labels in the labeling algorithm; an exhausted counter *mct* in a counter pair $\langle mct, cct \rangle$ is canceled, by setting $mct.lbl = cct.lbl$ (i.e., the counter's own label cancels it) and hence cannot be used as a local maximal counter in $maxC_i[i]$. This cannot increase the number of labels that are created, since the initial set of corrupt counters remains the same as the one for labels, for which we have already produced a proof in Section 4.3.1.

The enhanced labeling algorithm. Figure 4.5 presents a standalone version of the labeling algorithm adjusted for counters. Each processor p_i uses the token-based communication to transmit to every other processor p_j its own maximal counter and the one it currently holds for p_j in $maxC_i[j]$ (line 1). Upon receipt of such an update from p_j, p_i first performs canceling of any exhausted counters in *storedCnts*[] (line 4), in *maxC*[] (line 6) and in the received couple of counter pairs (line 5). Having catered for exhaustion, it then calls *maintainCntrs*($\langle \bullet, \bullet \rangle$) with the received two counter pairs as arguments. This is essentially a counter version of Algorithm 2. Macros that require some minor adjustments to handle counters are seen in Figure 4.4 lines 5 to 10. We also address the need to update counters of *maxC*[] w.r.t. *seqn* and *wid* based on counters from the *storedCnts*[] structure and vice versa in lines 18 and 10.

We define the operator *add*(*ctp*) (Fig. 4.3) to enqueue a counter pair *ctp* to a queue of *storedCnts*[*n*], where in case a counter with the same label already exists the following two rules apply: (1) if at least one of the two counters is cancelled we keep a canceled instance, and (2) if both counters are legitimate, we keep the greatest

```
   // Lines 1 and 2 run in the background.
1  upon transmitReady(p_j ∈ P \ {p_i}) do transmit(⟨maxC[i], maxC[j]⟩);
2  upon receive(⟨sentMax, lastSent⟩) from p_k do processCntr(sentMax, lastSent, j)
3  procedure processCntr(counter pair sentMax, counter pair lastSent), int k) begin
4      foreach p_j ∈ P, ctp ∈ storedCnts[j] : legit(ctp) ∧ exhausted(ctp) do cancelExh(ctp);
5      if (∃ctp' ∈ ⟨sentMax, lastSent⟩ : exhausted(ctp')) then cancelExh(ctp');
6      cancelExhMaxC(); maintainCntrs(sentMax, lastSent);
7  operator maintainCntrs(counter pair sentMax, counter pair lastSent), int k)
8  begin
9      if sentMax ≠ NULL then maxC[k] ← sentMax;
10     if lastSent ≠ NULL ∧ ¬legit(lastSent) ∧ maxC[i] =_mct.lbl lastSent then maxC[i].add(lastSent);
11     if staleCntrInfo() then storedCnts.emptyAllQueues();
12     foreach p_j ∈ P : recordDoesntExist(j) do retCntrQ(maxC[j]).add(maxC[j]);
13     foreach p_j ∈ P, cp ∈ storedCnts[j] : (legit(cp) ∧ (notgeq(j, cp) ≠ ⊥)) do cp.cct ← notgeq(j, cp);
14     foreach p_j ∈ P : ((¬legit(maxC[j]) ∨ (cp <_mct.seqn maxC[j])) ∧ (maxC[j] =_ml cp) ∧ legit(cp)
       where cp ∈ retCntrQ(maxC[j]) do cp ← maxC[j];
15     foreach p_j ∈ P, cp ∈ storedCnts[j] : double(j, cp) do cp.remove();
16     foreach p_j ∈ P : (legit(maxC[j]) ∧ (canceled(maxC[j]) ≠ ⟨⊥, ⊥⟩)) do
17         maxC[j] ← canceled(maxC[j])
18     foreach p_j ∈ P, cp ∈) do maxC[j] ← getMaxCnt(maxC[j]);
19     if legitCnts() ≠ ∅ then maxC[i] ← ⟨max_≺ct(legitCnts()), ⊥⟩;
20     else useOwnCntr();
```

Figure 4.5: The maintainCntrs() operator (code for p_i).

counter with respect to $\langle seqn, wid \rangle$. The counter is placed at the front of the queue.

Counter Increment Algorithm

Algorithm 3 shows a self-stabilizing counter increment algorithm where multiple processors can increment the counter. We start with some useful definitions and proceed to describe the algorithm.

Quorums. We define a *quorum set* $\mathbb{Q}$ based on processors in P, as a set of processor subsets of P (named *quorums*), that ensure a non-empty intersection of every pair of quorums. Namely, for all quorum pairs $Q_i, Q_j \in \mathbb{Q}$ such that $Q_i, Q_j \subset P$, it must hold that $Q_i \cap Q_j \neq \emptyset$. This *intersection property* is useful to propagate information among servers and exploiting the common intersection without having to write a value v to all the servers in a system, but only to a single quorum, say Q. If one wants to retrieve this value, then a call to *any* of the quorums (not necessarily Q), is expected to return v because there is least one processor in every quorum that

Algorithm 3: Increment Counter; code for p_i

```
1  interface function incrementCounter() begin
2      let ⟨responseSet, status⟩ ← ⟨∅, MAX_REQUEST⟩;
3      repeat
4          if status = MAX_REQUEST ∧ (∃Q ∈ 𝕼 : Q ⊆ {responseSet}) then
5              initWrite(); increment()
6          else if status = MAX_WRITE ∧ (∃Q ∈ 𝕼 : Q ⊆ {responseSet}) then
7              ⟨status ← COMPLETE⟩
8          foreach p_j ∈ P do send ⟨status, maxC[i], maxC[j]⟩;
9      until status = COMPLETE;
10     return maxC[i]
11 upon receive of m = ⟨subj, sentMax, lastSent⟩ from p_j begin
12     if (m.subj = MAX_REQUEST) then send ⟨ACK, maxC_i[i], maxC_i[j]⟩ to p_j;
13     else if (m.subj = MAX_WRITE) then
14         processCntr(sentMax, lastSent, j); send ⟨ACK, ⊥, lastSent⟩ to p_j;
15     else if (m.subj = ACK ∧ correctResponse(sentMax, lastSent)) then
16         processCntr(sentMax, lastSent, j); responseSet ← j
```

also belongs to Q. In the counter algorithm we exploit the intersection property to retrieve the currently greatest counter in the system, increment it, and write it back to the system, i.e., to a quorum therein. Note that majorities form a special case of a quorum system.

Algorithm description. To increment the counter, a processor p_i enters status MAX_REQUEST (line 2) and starts sending a request to all other processors, waiting for their maximal counter (via line 8). Processors receiving this request respond with their current maximal counter and the last sent by p_i (line 12). When such a response is received (line 15), p_i adds this to the local counter structures via the counter bookkeeping algorithm of Figure 4.5. Once a quorum of responses (line 4) have been processed, $maxC[i]$ holds the maximal counter that has come to the knowledge of p_i about the system's maximal counter. This counter is then incremented locally and p_i enters status MAX_WRITE by initiating the propagation of the incremented counter (line 5), and waiting to gather acknowledgments from a quorum (the condition of line 6). When the latter condition is satisfied, the function returns the new counter. This is, in spirit, similar to the two-phase write operation of MWMR register implementations, focusing on the sequence number rather than on an associated value.

Proof of correctness

Proof outline. Initially we prove, by extending the proof of the labeling algorithm, that starting from an arbitrary configuration the system eventually reaches to a global maximal label (as given in Theorem 4.3.3), even in the presence of exhausted counters (Lemma 4.3.4). By using the intersection property of quorums we establish that a counter that was written is known by at least one processor in every quorum (Lemma 4.3.5. We then combine the two previous lemmas to prove that counters increment monotonically.

Lemma 4.3.4. *In a bounded number of steps of Algorithm 4.5 every processor p_i has counter $maxC_i[i] = ct$ with $ct.lbl = \ell_{max}$ the globally maximal non-exhausted label.*

Proof. For this lemma we refer to the enhanced labeling algorithm for counters (Figure 4.5). The lemma proof can be mapped on the arguments proving lemmas Lemma 4.3.1 to Lemma 4.3.4 of Algorithm 2. Specifically, consider a processor p_i that has performed a full execution of *processCntr*() (Fig. 4.5 line 3) at least once due to a receive event. This implies a call to *maintainCntrs* and thus to $staleCntrInfo()$ (Fig. 4.5 line 11) which will empty all queues if exhausted non-canceled counters exist. Also there is a call to *cancelExhaustedMaxC*() which cancels all counters that are exhausted in *maxC*[]. By observation of the code, after a single iteration, there is no local exhausted counter that is not canceled.

Since every counter that is received and is exhausted becomes canceled, and since the arbitrary counters in transit are bounded, we know that there is no differentiation between exhausted labels that may cause a counter's label to be canceled. Namely, the size of the queues of *storedCnts*[] remain the same while at the same time provide the guarantees provided by the proof of the labeling algorithm. It follows from the labeling algorithm correctness and by our cancellation policy on the exhausted counters, that Theorem 4.3.3 can be extended to also include the use of counters without any need to locally keep more counters than there are labels.

We proceed to deduce that, eventually, any processor taking practically infinite number of steps in R obtains a counter with globally maximal label ℓ_{max}.

□

For the rest of the proof we refer to line numbers in Algorithm 3.

Lemma 4.3.5. *In an execution where Lemma 4.3.4 holds, it also holds that* $\forall Q \in \mathbb{Q}, \exists p_j \in Q$: $maxC_j[j] = ct \wedge (ct' \prec_{ct} ct)$, *where* $ct' \in \{storedCnts_k[k'] \cup maxC_k[k'] : ct' =_{lbl} ct\}_{p_k, p_{k'} \in P} \setminus \{ct\}$, *i.e., ct' is every counter in the system with identical label but less than ct w.r.t. seqn or wid and ct is the last counter increment.*

Proof. Observe that upon a quorum write, the new incremented counter *ct* with the maximal label *lb* is propagated (lines 6 and 8) until a quorum of acknowledgments have been received. Upon receiving such a counter by p_i, a processor p_j will first add *ct* to its structures via *processCntr*() and will then acknowledge the write. If this is the maximal counter that it has received (there could be concurrent ones) then the call to *processCntr*() will also have the following effects: (i) the counter's *seqn* and *wid* will be updated in the $storedCnts_j[]$ structure in the queue of the creator of *lb*, (ii) $maxC_j[j] \leftarrow ct$.

Since p_i waits for responses by a quorum before it returns, it follows that by the intersection property of the quorums, the lemma must hold when p_i reaches status COMPLETE. □

Theorem 4.3.6. *Given an execution R of the counter increment algorithm in which at least a majority of processors take a practically infinite number of steps, the algorithm ensures that counters eventually increment monotonically.*

Proof. Consider a configuration $c \in R'$ where R' is a suffix of R in which Lemma 4.3.4 holds, and in which ct_{max} is the counter which is maximal with respect to $\prec_{ct}$. There are two cases that the counter may be incremented.

In the first case, a legal execution, the counter ct_{max} is only incremented by a call to *incrementCounter*(), By Lemma 4.3.5 any call to *incrementCounter*() will return the last written maximal counter (namely ct_{max}). When this is incremented giving ct'_{max} then $ct'_{max}.seqn = ct_{max}.seqn + 1$ which is monotonically greater than ct'_{max} and in case of concurrent writes the *wid* is unique and can break symmetry enforcing the monotonicity.

The second case arises when ct_{max} comes from the arbitrary initial state, is not known by a quorum, and resides in either a local state or is in transit. When ct_{max} eventually reaches a processor, it becomes the local maximal and it is propagated either via counter maintenance or in the first stage of a counter increment when the maximal counters are requested by the writer. In this case the use of ct_{max} is also a

monotonic increment, and from this point onwards any increment in R' proceeds monotonically from ct_{max}, as described in the previous paragraph. □

Algorithm Complexity

The local memory of a processor implementing the counter increment is not different in order to the labeling algorithm's, since converting to the counter structures only adds an integer (the sequence number). Hence the *space complexity* of the algorithm is $O(n^3)$ in counters. The upper bound on *stabilization time* in the number of counter increments that are required to reach a period of practically infinite counter increments can be deduced by Theorem 4.3.6. For some t such that $0 \leq t \leq 2^\tau$ in an execution with $O(n \cdot \beta \cdot t)$ counter increments (recall that $\beta = n^3 cap + 2n^2 - 2n$), there is a practically infinite period of (2^τ) monotonically increasing counter increments in which the label does not change.

MWMR Register Emulation

Having a practically-self-stabilizing counter increment algorithm, it is not hard to implement a *practically-self-stabilizing MWMR register emulation*. Each counter is associated with a value and the counter increment procedure essentially becomes a write operation: once the maximal counter is found, it is increased and associated with the new value to be written, which is then communicated to a majority of processors. The read operation is similar: a processor first queries all processors about the maximum counter they are aware of. It collects responses from a majority and if there is no maximal counter, it returns $\perp$ so the processor needs to attempt to read again (i.e., the system hasn't converged to a maximal label yet). If a maximal counter exists, it sends this together with the associated value to all the processors, and once it collects a majority of responses, it returns the counter with the associated value (the second phase is a required to preserve the consistency of the register (c.f. [17,32]).

4.4 Virtually Synchronous Stabilizing Replicated State Machine

Group communication systems (GCSs) that guarantee the virtual synchrony property, essentially suggest that processes that remain together in consecutive groups (called *views*) will deliver the same messages in the desired order [8]. This is particularly suited to maintain a replicated state machine service, where replicas need to remain consistent, by applying the same changes suggested by the environment's requests. A key advantage of multicast services (with virtual synchrony) is the ability to reuse the same view over many multicast rounds, which allows every automaton step to require just a single multicast round, as compared to other more expensive solutions.

GCSs provide the VS property by implementing two main services: a reliable multicast service, and a membership service to provide the membership set of the view, whilst they also assume access to unbounded counters to use as unique view identifiers. We combine existing self-stabilizing versions of the two services (with adaptations where needed), and we use the counter from the previous section to build the first (to our knowledge) practically-self-stabilizing virtually synchronous state machine replication. While the ideas appear simple, combining the services is not always intuitive, so we first proceed to a high-level description of the algorithm and the services, and then follow the algorithm with a more technical description and the correctness proof.

4.4.1 Preliminaries

The algorithm progresses in state replication by performing multicast rounds once a view is installed, where a *view* is a tuple composed of a members' *set* taken from P, and of a unique identifier (ID) that is a counter as defined in the previous section. This view must include a *primary partition* (defined formally in Definition 4.4.1), namely it must contain a majority of the processors in P, i.e., $n/2 + 1$. In our version, a processor, the *coordinator*, is responsible: (1) to progress the multicast service which we detail later, (2) to change the view when its failure detector suggests changes to the composition of the view membership. Therefore, the output of the coordinator's failure detector defines the set of view members; this helps to maintain a consistent

membership among the group members, despite inaccuracies between the various failure detectors.

On the other hand, the counter increment algorithm that runs in the background allows the coordinator to draw a counter for use as a view identifier and in this case, the counter's writer identifier (*wid*) is that of the view's coordinator. This defines a simple interface with the counter algorithm, which provides an identical output. Pairing the coordinator's member set with a counter as view identity we obtain a *view*. Of course as we will describe later, reaching to a unique coordinator may require issuing several such view proposals, of which one will prevail. We first suggest a possible implementation of a failure detector (to provide membership) and of a reliable multicast service over the self-stabilizing FIFO data link given in Chapter 3, and then proceed to an algorithm overview.

Definition 4.4.1 (Primary Partition)**.** *We say that the output of the (local) failure detectors in execution R includes a* primary partition *when it includes a supporting majority of processors* $P_{maj} : P_{maj} \subseteq P$*, that (mutually) never suspect at least one processor, i.e.,* $\exists p_\ell \in P$ *for which* $|P_{maj}| > \lfloor n/2 \rfloor$ *and* $(p_i \in (P_{maj} \cap FD_\ell)) \Longleftrightarrow (p_\ell \in (P_{maj} \cap FD_i))$ *in every* $c \in R$*, where* FD_x *returns the set of processors that according to* p_x*'s failure detector are active.*

Failure detector. The use of failure detection for our virtual synchrony algorithm is merely for liveness. Inaccurate failure detection that violates the liveness assumption of Definition 4.4.1 can lead to continuous changes in views or coordinators. Following [26], we present a possible implementation of a self-stabilizing failure detector. As stated there, the implementation works under a partial synchrony assumption; this assumption is strong enough to implement a perfect failure detector, whilst, as we discuss below, our algorithm requires a weaker failure detector than this. (Recall the discussion in Section 2.5.2.)

The failure detector is implemented as follows. Every processor p uses the token-based mechanism to implement a heartbeat (see Chapter 3) with every other processor, and maintain a heartbeat integer counter for every other processor q in the system. Whenever processor p receives the token from processor q over their data link, processor p resets the counter's value to zero and increments all the integer counters associated with the other processors by one, up to a predefined threshold value W. Once the heartbeat counter value of a processor q reaches W, the failure detector of processor p considers q as inactive. In other words, the failure detector at

processor p considers processor q to be active, if and only if the heartbeat associated with q is strictly less than W.

As an example, consider a processor p which holds an array of heartbeat counters for processors p_i, p_j, p_k such that their corresponding values are $\langle 2, 5, W-1 \rangle$. If p_j sends its heartbeat, then p's array will be changed to $\langle 3, 0, W \rangle$. In this case, p_k will be suspected as crashed, and the failure detector reading will return the set p_i, p_j as the set of processors considered correct by p.

Note that our virtual synchrony algorithm, employs the same implementation but has weaker requirements than [26] that solve consensus, and thus they resort to a failure detector at least as strong as Ω [60]. Specifically, in Definition 4.4.1 we pose the assumption that *just a majority of the processors* do not suspect at least one processor of P for sufficiently long time, in order to be able to obtain a long-lived coordinator. This is different, as we said before, to an eventually perfect failure detector that ensures that after a certain time, no active processor suspects any other active processor.

Our requirements, on the other hand, are stronger than the weakest failure detector required to implement atomic registers (when more than a majority of failures are assumed), namely the Σ failure detector [130], since virtual synchrony is a more difficult task. In particular, whilst the Σ failure detector eventually outputs a set of only correct processors to correct processors, we require that this set in at least half of the processors, will contain at least one common processor. In this perspective our failure detector seems to implement a *self-stabilizing* version of a slightly stronger failure detector than Σ. It would certainly be of interest for someone to study what is the weakest failure detector required to achieve practically-self-stabilizing virtually synchronous state replication, and whether this coincides with our suggestion.

Reliable multicast implementation. The coordinator of the view controls the exchange of messages (by multicasting) within the view. The coordinator requests, collects and combines input from the group members, and then it multicasts the updated information. Specifically, when the coordinator decides to collect inputs, it waits for the token (see Chapter 3) to arrive from each group participant. Whenever a token arrives from a participant, the coordinator uses the token to send the request for input to that participant, and waits the token to return with some input (possibly $\perp$, when the participant does not have a new input). Once the coordinator

receives an input from a certain participant with respect to this multicast invocation, the corresponding token will not carry any new requests to receive input from the same participant; of course, the tokens continue to move back and forth to sustain the heartbeat-based failure detector. When all inputs are received, the processor combines them and again uses the token to carry the updated information. Once this is done, the coordinator can proceed to the next input collection.

We provide the pseudocode for the practically-stabilizing replicate state machine implementation as Algorithm 4, and an algorithm description followed by the correctness analysis.

4.4.2 Virtual Synchrony Algorithm

Algorithm Outline. The algorithm is coordinator-led. Namely, each processor ensures that it has its coordinator. If it does not have a coordinator, it may either propose or wait until through the exchange of information, it either finds a coordinator, or finds enough support to propose. A coordinator on its side, proposes and installs the new view and builds the new state based on collected messages and states (with majorities ensuring state integrity). It then proceeds to lead multicast rounds by first collecting all the messages by the view members, and then propagating them back to the view members for them apply the required side-effects on the replica.

Detailed description

The existence of a coordinator p_ℓ is in the heart of Algorithm 4. Processors that belong to and accept p_ℓ's view proposal are called the *followers* of p_ℓ. The algorithm determines the availability of a coordinator and acts towards the election of a new one when no valid such exists (lines 2–6). The pseudocode details the coordinator-side (lines 7–10) and the follower-side (lines 11–14) actions. At the end of each iteration, Algorithm 4 defines how p_ℓ and its followers exchange messages (lines 15, 16).

Variables. The state of each processor includes its current *view*, and *status* = {Propose, Install, Multicast}, which refers to either usual message multicast operation (when in Multicast), or view establishment rounds in which the coordinator can first Propose a new view and then proceed to Install it once conditions are met. During multicast rounds, *rnd* denotes the round number, *state* stores the replica, $msg[n]$ is an array that includes the last delivered messages to the state machine,

Interfaces: $fetch()$ next multicast message, $apply(state, msg)$ applies the step msg to $state$ (while producing side effects), $synchState(replica)$ returns a replica consolidated state, $synchMsgs(replica)$ returns a consolidated array of last delivered messages, $failureDetector()$ returns a vector of processor pairs $\langle pid, crdID \rangle$, $inc()$ is a call to the $incrementCounter()$ interface function of Algorithm 3 and returns a counter.

Variables: $rep[n] = \langle view = \langle ID, set \rangle, status \in \{\text{Multicast}, \text{Propose}, \text{Install}\}$, (*multicast round number*) rnd, (*replica*) $state$, (*last delivered messages*) $msg[n]$ (*to the state machine*), (*last fetched*) $input$ (*to the state machine*), $propV = \langle ID, set \rangle$, (*no coordinator alive*) $noCrd$, (*recently live and connected component*) $FD\rangle$: an array of the state machine's replica, where $rep[i]$ refers to the one that processor p_i maintains, and $rep[j]$ refers to the last arriving message from p_j containing p_j's $rep[j]$. A local variable $FDin$ stores the $failureDetector()$ output. FD is an alias for $\{FDin.pid\}$, i.e. the set of processors that the failure detector considers as active. Let $crd(j) = \{FDin.crdID : FDin.pid = j\}$, i.e. the id of p_j's local coordinator, or $\perp$ if none.

Figure 4.6: Interfaces and Variables for Algorithm 4, code for processor p_i

which is the *input* fetched by each group member and then aggregated by the coordinator during the previous multicast round. During multicast rounds, it holds that $propV = view$. However, whenever $propV \neq view$ we consider $propV$ as the newly proposed view and $view$ as the last installed one. Each processor also uses $noCrd$ and FD to indicate whether it is aware of the absence of a recently active and connected valid coordinator, and respectively, of the set of processor present in the connected component, as indicated by its local failure detector.

Interfaces. Algorithm 4 assumes access to the application's message queue via $fetch()$, which returns the next multicast message, or $\perp$ when no such message is available. It also assumes the availability of the automaton state transition function, $apply(state, msg)$, which applies the aggregated input array, msg, to the replica's $state$ and produces the local side effects. The algorithm collects the followers' replica states and uses $synchState(replica)$ to return the new state. Function $failureDetector()$ provides access to p_i's failure detector by returning a set of responsive processors. It also returns the processor that every processor considers as its coordinator. Function $inc()$ (counter increment) is a call to the $incrementCounter()$ function (of Section 4.3.3). It fetches a new and unique counter to be used as a view identifier, ID. This can be totally ordered by $\preceq_{ct}$ and $ID.wid$ is the identity of the processor that incremented the counter. The state variables, constants and interfaces can also be seen in Figure 4.6.

We proceed with a detailed explanation of the algorithm's functionality. The descriptive macros of Algorithm 4 are rigorously defined in Figure 4.7.

Algorithm 4: Practically-self-stabilizing automaton replication using virtual synchrony, code for processor p_i

```
1  Do forever begin
2      readFD();
3      candCoords ← findCandidateCoordSet();
4      maxValCrd ← findCandidateCoordWithMaxViewID();
5      setCoordVariables();
6      if proposeRequired&Permitted() then ⟨status, propV⟩ ← ⟨Propose, ⟨inc(), FD⟩⟩;
7      else if selfCoordinator() ∧ roundProceedReady() then
8          if status = Multicast then coordinateMcastRnd();
9          else if status = Install then coordinateInstall() ;
10         else if status = Propose then coordinatePropose();
11     else if ¬selfCoordinator() ∧ roundReadyToFollow() then
12         if status = Multicast then followMcastRnd();
13         else if status = Install then followInstall();
14         else if status = Propose then followPropose();
15     sendUpdates();
16 Upon message arrival m from p_j do rep[j] ← m;
```

Replica consistency. Each participant maintains a replica of the state machine and the last processed (composite) message. We bound the memory used to store the history of the replicated state machine by keeping the (encapsulated influence of the history represented by the) current state of the replicated state machine. In addition, each participant maintains the last delivered message set to ensure common reliable multicast, as the coordinator may stop being active prior to ensuring that all members received a copy of the last multicast message.

Whenever a new view is installed, the coordinator inquires all members (forming a majority) for the most updated state and delivered message (macro *coordinateInstall*() in line 9). Since at least one of the members, say p_i, participated in the view in which the last completed state machine transition took place, p_i's information will be recognized as associated with the largest counter. This is adopted by the coordinator that assigns the most up-to-date state and available delivered messages to all the current group members (via *followInstall*() line 13), thus satisfying the virtual synchrony property.

Performing multicast rounds. As part of the replication procedure, the coordinator leads consecutive multicast rounds by first collecting inputs received from the environment before ensuring that all group members apply these inputs to the replica state machine (procedure *coordinateMcastRnd*() line 8). Note that the received mul-

1 **Macros:**
2 $readFD() = \{FDin \leftarrow failureDetector()\}$;
3 $findCandidateCoordSet() = \{p_\ell = rep[\ell].propV.ID.wid \in FD : (|rep[\ell].propV.set| > \lfloor n/2 \rfloor) \wedge (|rep[\ell].FD| > \lfloor n/2 \rfloor) \wedge (p_\ell \in rep[\ell].propV.set) \wedge (p_k \in rep[\ell].propV.set \leftrightarrow p_\ell \in rep[k].FD) \wedge ((rep[\ell].status = \mathsf{Multicast}) \rightarrow (rep[\ell].(view = propV) \wedge crd(\ell) = \ell)) \wedge ((rep[\ell].status = \mathsf{Install}) \rightarrow crd(\ell) = \ell)\}$;
4 $findCandidateCoordWithMaxViewID() = \{p_\ell \in seemCrd : (\forall p_k \in seemCrd : rep[k].propV.ID \preceq_{ct} rep[\ell].propV.ID)\}$
5 $setCoordVariables() = \{noCrd \leftarrow (|maxValCoord| \neq 1); crdID \leftarrow maxValCoord\}$;
6 $proposeRequiredAndPermitted() = ((|FD| > \lfloor n/2 \rfloor) \wedge (majNoCrd() \vee repropose())$;
7 $majNoCrd() = ((|maxValCoord| \neq 1) \wedge (|\{p_k \in FD : p_i \in rep[k].FD \wedge rep[k].noCrd\}| > \lfloor n/2 \rfloor))$;
8 $repropose() = ((maxValCoord = \{p_i\}) \wedge (FD_i \neq propV.set) \wedge (|\{p_k \in FD_i : rep[k].propV = propV\}| > \lfloor n/2 \rfloor)))$;
9 $selfCoordinator() = (maxValCoord = \{p_i\})$;
10 $roundProceedReady() = (\forall p_j \in view.set : rep[j].(view, status, rnd) = (view, status, rnd)) \vee ((status \neq \mathsf{Multicast}) \wedge (\forall p_j \in propV.set : rep[j].(propV, status) = (propV, \mathsf{Propose}))$;
11 $coordinateInstall() = \{(view, status, rnd) \leftarrow (propV, \mathsf{Multicast}, 0)\}$;
12 $coordinatePropose() = \{(state, status, msg) \leftarrow (synchState(rep), \mathsf{Install}, synchMsgs(rep))\}$;
13 $roundReadyToFollow() = (rep[\ell].rnd = 0 \vee rnd < rep[\ell].rnd \vee rep[\ell].(view \neq propV))$;
14 $followInstall() = \{rep[i] \leftarrow rep[\ell]\}$;
15 $followPropose() = \{(status, propV) \leftarrow rep[\ell].(status, propV)\}$;
16 **Procedures:**
17 *coordinateMcastRnd*() **do begin**
18 $apply(state, msg)$; $input \leftarrow fetch()$;
19 **foreach** $p_j \in P$ **do if** $p_j \in view.set$ **then** $msg[j] \leftarrow rep[j].input$ **else** $msg[j] \leftarrow \perp$;
20 $rnd \leftarrow rnd + 1$;
21 *followMcastRnd*() **do begin**
22 $rep[i] \leftarrow rep[\ell]$; $apply(state, rep[\ell].msg)$;
23 $input \leftarrow fetch()$;
24 *sendUpdates*() **do begin**
25 **let** $m = rep[i]$;
26 **let** $sendSet = (seemCrd \cup \{p_k \in propV.set : maxValCoord = \{p_i\}\} \cup \{p_k \in FD : noCrd \vee (status = \mathsf{Propose})\})$;
27 **foreach** $p_j \in sendSet$ **do** $send(m)$;

Figure 4.7: Macros and Procedures for Algorithm 4, code for processor p_i

ticast messages consist of input (possibly $\perp$) from each of the processors, thus, the processors need to apply one input at a time, and the processors may apply them in an agreed upon sequential order, say from the input of the first processor to the last. Alternatively, the coordinator may request one input at a time in a round-robin fashion and multicast it. Finally, to ensure that the system stabilizes when started in an arbitrary configuration, the coordinator assigns the state of its replica to the other members (via *sendUpdates*() line 15 and *followMcastRnd*() line 12).

Determining coordinator availability. The system may reach a configuration with no coordinator due to an arbitrary initial configuration, or due to the coordinator's crash failure. Each participant detecting the absence of a coordinator, seeks for potential candidates for the task based on the exchanged information. Processor p regards processor q as a candidate ($findCandidateCoordSet()$, line 3), if q is active according to p's failure detector, and there is a majority of processors that appear to support this based on p's knowledge, which due to asynchrony may be inaccurate. If there are more than one candidates, processor p selects the one with the view bearing the highest identifier, i.e., counter ($findCandidateCoordWithMaxViewID()$ line 4). If there is one, then p considers this to be the coordinator and waits to hear from it or learn that it is not active.

If there is no such processor, and if based on its local knowledge there is a majority of processors that also do not have a coordinator (condition *proposeRequiredAndPermitted*() line 6), then processor p proceeds to propose a new view. This has a view ID acquired from the increment counter algorithm and a view set composed of the processors that p's failure detector reports as active (line 6). As we show, if p's proposal is accepted from *all* the processors in the view (i.e., they have all executed $followPropose()$ line 14 and then propagated this via *sendUpdates*() back to p), then p proceeds to install the view, unless another processor who has obtained a higher counter does so.

We also note that GCSs providing VS often leverage on the system's ability to preserve (when possible) the coordinator during view transitions rather than venturing to install a new one, a certainly more expensive procedure. Our solution naturally follows this approach through our assumption of a supportive majority (Definition 4.4.1), where coordinators have the support of a majority of processors by never being suspected throughout a very long period. During such a period, our algorithm persists on using the same coordinator even when views change.

4.4.3 Correctness Proof of Algorithm 4

Outline. The correctness proof shows that starting from an arbitrary state in an execution R of Algorithm 4 satisfying Definition 4.4.1, we reach a configuration $c \in R$ in which some processor with supporting majority p_ℓ will propose a view including its supporting majority (Lemmas 4.4.1 and 4.4.2). This view is either accepted by all

its member processors or in the case where p_ℓ experiences a failure detection change, it can repropose a view (Lemma 4.4.3). We conclude with Theorem 4.4.4 proving that any execution suffix of R that begins from such a configuration c will preserve the virtual synchrony property and implement state machine replication. We begin with some definitions.

Definitions. For a processor p_ℓ which has a supporting majority, predicate $supMaj(\ell) = \mathsf{True}$, and its supporting majority set is denoted by $P_{maj}(\ell)$. Once the system considers p_ℓ as the view coordinator, $P_{maj}(\ell)$ can extend the support throughout R and thus p_ℓ continues to emulate the automaton with them. Furthermore, there is no clear guarantee for a view coordinator to continue to coordinate for an unbounded period when it does not meet the criteria of Definition 4.4.1 throughout R. Therefore, for the sake of presentation simplicity, the proof considers any execution R with only *definitive suspicions*, i.e., once processor p_i suspects processor p_j, it does not stop suspecting p_j throughout R. The correctness proof implies that eventually, once all of R's suspicions appear in the respective local failure detectors, the system elects a coordinator that has a supporting majority throughout R.

Consider a configuration c in an execution R of Algorithm 4 and a processor $p_i \in P$. We define the *local (view) coordinator* of p_i, say p_ℓ, to be the only processor that, based on p_i's local information, has a proposed view satisfying the conditions of lines 3 and 4 in Fig. 4.7 such that $maxValCoord = \{p_\ell\}$. p_ℓ is also considered the *global (view) coordinator* if for all p_k in p_ℓ's proposed view ($propV_\ell$), it holds that $maxValCoord_k = \{p_\ell\}$. We write $(global(\ell) = \mathsf{True}) \Leftrightarrow (propV_\ell.set \subseteq \{p_k \in P : maxValCoord_k = \{p_\ell\}\})$. When p_i has a (local) coordinator then p_i's local variable $noCrd = \mathsf{False}$, whilst when it has no local coordinator, $noCrd = \mathsf{True}$. Moving to the proof, we consider the following useful remark on Definition 4.4.1 of page 66.

Remark: Definition 4.4.1 suggests that we can have more than one processor that has supporting majority. In this case, it is not necessary to have *the same* supporting majority for all such processors. Thus for two such processors p_i, p_j with respective supporting majorities $P_{maj}(i)$ and $P_{maj}(j)$ we do not require that $P_{maj}(i) = P_{maj}(j)$, but $P_{maj}(i) \cap P_{maj}(j) \neq \emptyset$ trivially holds.

From this point onward we refer to an execution R of Algorithm 4 as an execution of infinite number of steps starting in an arbitrary system configuration, and in which Definition 4.4.1 holds.

Lemma 4.4.1. *Consider a processor p_i which has a local coordinator p_k, such that p_k is either inactive or $\neg supMaj(k)$ throughout R. There is a configuration $c \in R$, after which p_i does not consider p_k to be its local coordinator.*

Proof. We prove that eventually $maxValCrd_i = p_k$ stop holding. There are the two possibilities regarding processor p_k.

Case 1: Processor p_k is inactive throughout R. By our responsiveness failure detector, eventually $p_k \notin FD_i$. The threshold W that we set for our failure detector determines how soon p_k is suspected. Then $p_k \notin FD_i \Rightarrow p_k \notin candCoords \Rightarrow p_k \notin maxValCoord_i$, i.e., p_i stops considering p_k as its local coordinator. By definitive suspicions, p_i does not stop suspecting p_k throughout$R$.

We study the case where p_k is active, but $\neg supMaj(k)$. Two subcases exist:

Case 2(a): p_k is the coordinator of itself and considers itself to have support from a majority of others' failure detectors. Namely, $(p_k \in maxValCrd_k) \Rightarrow (p_k \in candCoords_k) \Rightarrow (|FD_k| \geq \lfloor n/2 \rfloor)$. By *sendUpdates*() (line 15), p_k propagates $rep_k[k]$ to p_i in every iteration. By the assumption that $\nexists P_{maj}(k)$ then eventually $|FD_k| < \lceil n/2 \rceil$ thus $(p_k \in maxValCrd_k)$ becomes false. Then,

(*i*) If p_k does not find a new coordinator, $noCrd_k = \mathsf{True}$, and p_k propagates this to p_i. By the detailed description of $findCandidateCoordSet()$ (Fig. 4.7 line 3), since $|rep_i[k].FD| \not\geq \lfloor n/2 \rfloor$ then $(p_k \notin candCoords_i) \Rightarrow (p_k \notin maxValCrd_i)$.

(*ii*) Alternatively, p_k may find a new coordinator before propagating $rep_k[k]$. If p_k has a coordinator other than itself, then it propagates $rep_k[k]$ only to its coordinator, thus p_i does not receive this information. We refer to the next case:

Case 2(b): p_k has a different local coordinator than itself, say p_ℓ, namely, $maxValCrd_k = \ell$. We remind that the failure detector token returns both the set of responsive processors, and the processor they consider as their coordinator (Fig. 4.6). As per the algorithm's notation, the coordinator of processor p_k known by p_i is given by $crd_i(k)$. Since p_i's failure detector regards p_k as active, then $crd_i(k)$ is indeed updated (remember that p_i receives the token with p_k's $crd(k)$ infinitely often from p_k), otherwise p_k is removed from FD and is not a valid coordinator for p_i. Thus eventually, $crd_i(k) = \ell \neq k$. We conclude that p_k stops being p_i's coordinator. □

We now define the notion of "propose" to be used in the sequel.

Definition 4.4.2. *Processor $p_\ell \in P$ with status $= \mathsf{Propose}$, is said to* propose *a view $propV_\ell$, if in a complete iteration of Algorithm 4, p_ℓ either satisfies $maxValCoord_\ell = \{p_\ell\}$ or*

satisfies all the conditions of line 6 to create $propV_\ell$. A proposal is completed when $propV_\ell$ is propagated through sendUpdates() (line 15) to all the members of FD_ℓ.

The above definition does not imply that p_ℓ will continue proposing the view $propV_\ell$, since the replicas received from other processors may force p_ℓ to either exclude itself from $maxValCoord_\ell$ or create a new view (see Lemma 4.4.3). Also note that the origins of such a proposed view are not defined, i.e., it may be the result of the arbitrary initial state

Lemma 4.4.2. *Consider an execution R starting with no global coordinator, the system reaches a configuration in which at least one processor with a supporting majority will propose a view.*

Proof. Let $p_\ell \in P$ such that $supMaj(\ell) = \mathsf{True}$. Assume for contradiction that throughout R, no processor satisfying $supMaj()$ proposes a view. Then p_ℓ either does not have a coordinator or has a non-global local coordinator.

Case 1: $\mathbf{p}_\ell$ does not have a coordinator ($noCrd_\ell = \mathsf{True}$). Since p_ℓ is assumed to not propose, it must be that the conditions for *proposeRequired&Permitted*() (line 6) do not hold. We note that:

(1) $|FD_\ell| \geq \lfloor n/2 \rfloor$ holds since we assumed $supMaj(\ell)$.

(2) In the second condition of *proposeRequiredAndPermitted*(), both (i) $majNoCrd_\ell() = ((|maxValCoord_\ell| \neq 1) \wedge (|\{p_i \in FD_\ell : p_\ell \in rep_\ell[i].FD_\ell \wedge rep_\ell[i].noCrd\}| > \lfloor n/2 \rfloor))$ and (ii) $repropose_\ell() = ((maxValCoord_\ell = \{p_\ell\}) \wedge (FD_\ell \neq propV_\ell.set) \wedge (|\{p_i \in FD : rep[i].propV = propV\}| > \lfloor n/2 \rfloor))$ must fail due to our assumption that p_ℓ never proposes. Indeed (ii) fails since $noCrd_\ell = \mathsf{True} \Rightarrow maxValCoord_\ell \neq \{p_\ell\}$.

(3) Note that, $(noCrd_\ell = \mathsf{True}) \Rightarrow (|maxValCrd| \neq 1)$ (see definition of *setCoordVariables*() in Fig. 4.7).

From (1), (2) and (3) we deduce it is sufficient for condition $(|\{p_i \in FD_\ell : p_\ell \in rep_\ell[i].FD_\ell \wedge rep_\ell[i].noCrd\}| > \lfloor n/2 \rfloor))$ to be satisfied for p_ℓ to propose.

Let's assume that only one processor $p_j \in P_{maj}(\ell) \subseteq FD_\ell$ is required to switch from $noCrd_j = \mathsf{False}$ to True in order for p_ℓ to gain a majority of processors without a coordinator. But if $noCrd_j = \mathsf{False}$ then p_j must already have a coordinator, say p_k. If $supMaj(k) = \mathsf{True}$ then the lemma trivially holds. It must be that $supMaj(k) = \mathsf{False}$. Lemma 4.4.1 guarantees that p_j eventually stops considering p_k as its coordinator, and so $noCrd = \mathsf{True}$ and by the propagation of its replica, $majNoCrd_\ell()$ holds, allowing p_ℓ to propose. But this implies a contradiction, so the following case must hold.

Case 2: p_ℓ has a coordinator, say $p_{k'}$. Using the same arguments of the previous paragraph for p_ℓ rather than for p_j leads to contradiction and thus the lemma follows. □

Lemma 4.4.2 establishes that at least one processor with supporting majority will propose a view in the absence of a valid coordinator. We now move to prove that such a processor will only propose one view, unless it experiences changes in its *FD* that render the view proposal's membership obsolete. The lemma also proves that any two processors with supporting majority will not create views in order to compete for the coordinatorship.

Lemma 4.4.3 (Closure and Convergence)**.** *In an execution R, the system reaches a configuration where some p_ℓ satisfying supMaj(ℓ) proposes a view $propV_\ell$, and only proposes a new view if repropose$_\ell$() = True (Fig. 4.7, line 8). Moreover, global(ℓ) holds throughout R.*

Proof. We distinguish the following cases:

Case 1: Only one processor p_ℓ satisfying *supMaj*() exists. By Lemma 4.4.2, p_ℓ eventually proposes $propV_\ell$, based on its current FD_ℓ reading. Therefore, $P_{maj}(\ell) \subseteq propV_\ell.set$. By Lemma 4.4.1, processors without a supporting majority stop proposing and being local coordinators of any $p_j \in propV_\ell.set$. By propagation of information (line 15), it eventually it holds that $\forall p_j \in propV_\ell.set$, $rep_j[j].propV = rep_j[\ell].propV = propV_\ell$ and $rep_\ell[j] = propV_\ell$.

The only condition that may prevent some $p_j \in propV_\ell.set$ from adopting $propV_\ell$ is if for some $p_{j'} \in rep_j[\ell].propV_\ell.set$ it holds that $p_\ell \notin rep_j[j'].FD$ (Fig. 4.7, line 3). Plainly put, p_j believes that $p_{j'}$ suspects p_ℓ. Then, p_ℓ will either suspect $p_{j'}$ and propose a view (via *repropose*()) or p_j has obsolete information and as soon as its information is updated by propagation from $p_{j'}$, p_j adopts $propV_\ell$. By $supMaj(\ell)$, p_ℓ has at least a majority of $propV_\ell.set$ to accept $propV_\ell$. This allows p_ℓ and only p_ℓ to repropose a view. The processors that accepted the previous proposed view will accept this one as well. If $propV_\ell$ is accepted by all in $propV_\ell.set$ then p_ℓ proceeds to install the view. Thus p_ℓ eventually satisfies $global(\ell)$ if it is the single majority-supported processor.

Case 2: More than one processor with supporting majority. Consider two processors $p_\ell, p_{\ell'}$ that satisfy *supMaj*() and each creates a view. By the correctness of our counter algorithm, *inc*() returns two distinct and ordered counters to use as view identifiers. Without loss of generality, we assume that $propV_\ell$ proposed by p_ℓ has

the greatest identifier of all the counters created by calls to *inc*(). We identify the following four subcases:

Case 2(a): $p_\ell \in FD_{\ell'} \wedge p_{\ell'} \in FD_\ell$. Then $p_{\ell'}$ proposes $propV_{\ell'}$ and p_ℓ receives $propV_{\ell'}$, but $propV_{\ell'}.ID \preceq_{ct} propV_\ell.ID$ (line 3) so $maxValCrd = \{p_\ell\} \neq \{p_{\ell'}\}$ Proposal $propV_\ell$ is also propagated and since it is maximal in $propV_\ell.ID$, it is adopted by all $p_i \in propV_\ell.set$ including $p_{\ell'}$. Thus any processor satisfying *supMaj*() like $p_{\ell'}$ will propose at most once, and p_ℓ will become the sole coordinator. Note that in case where the failure detection reading changes for p_ℓ the reasoning is the same as Case 1 of this lemma, by noticing that if p_ℓ manages to get a majority of processors of *propV.set* then p_ℓ will change its proposed view without losing this majority.

Case 2(b): $p_\ell \notin FD_{\ell'} \wedge p_{\ell'} \notin FD_\ell$. By this condition, none of the two processors are informed of the other's proposal directly, and they exclude the other by the first condition of $findCandidateCoordSet()$. Eventually by propagation, $\forall p_i \in propV_\ell.set \cap propV_{\ell'}.set$ it holds that $maxValCrd_i = \{p_\ell\}$. It is possible that briefly $global(\ell') = \mathsf{True}$ but eventually $global(\ell)$ holds making the latter false. What is more crucial, is that $p_{\ell'}$ cannot make another proposal, since it will not have a majority of processors that do not have a coordinator. This is deduced from the intersection property of the two majorities ($propV_\ell.set$ and $propV_{\ell'}.set$) and reasoning of previous cases.

Case 2(c): $p_\ell \in FD_{\ell'} \wedge p_{\ell'} \notin FD_\ell$. Here we note that since p_ℓ has the greatest counter but has not included $p_{\ell'}$ to its $propV_\ell.set$, it should eventually be able to get all the processors in $propV_\ell.set$ to follow some $propV_\ell$ by using the arguments of Case 2(a). In the mean time $p_{\ell'}$ will, in vain, be waiting for a response from p_ℓ accepting $propV_{\ell'}$. By reasoning of previous cases, $p_{\ell'}$ does not repropose.

Case 2(d): $p_\ell \notin FD_{\ell'} \wedge p_{\ell'} \in FD_\ell$. This case is not symmetric to the above due to our assumption that p_ℓ is the one that has drawn the greatest view identifier from *inc*(). Here $propV_\ell.set$ includes $p_{\ell'}$ so p_ℓ waits for a response from $p_{\ell'}$ to proceed to the installation of $propV_\ell$. On the other hand, $p_{\ell'}$ will be waiting for responses from the processors in $propV_{\ell'}.set$. Any $p_i \in propV_\ell.set \cap propV_{\ell'}.set$ cannot keep $propV_\ell$ (even if initially it has accepted it, since it does not satisfy condition $p_{\ell'} \in rep[\ell].propV.set \Leftrightarrow p_\ell \in rep[\ell'].FD$ of $findCandidateCoorSet()$ (Fig. 4.7, line 3). Thus p_i accept $propV_{\ell'}$ instead of $propV_\ell$, and p_ℓ cannot repropose due to lack of a majority.

By the above exhaustive examination of cases, we reach to the result. The above proof guarantees both convergence to a legal execution and closure, since p_ℓ remains the coordinator as long as it has a supporting majority. □

Theorem 4.4.4. *An execution R of Algorithm 4 simulates an automaton replication preserving the virtual synchrony (VS) property.*

Proof. Consider a finite prefix R' of R. Assume that in this prefix Lemma 4.4.3 holds, i.e., we reach a configuration in which $\exists p_\ell : global(\ell) \wedge supMaj(\ell)$, and p_ℓ coordinates view v. We define a *multicast round* to be a sequence of ordered events: (i) $fetch()$ input and propagate to coordinator, (ii) coordinator propagates the collected messages of this round, (iii) messages are delivered and (iv) all view members $apply()$ side effects. The VS property is preserved between two consecutive rounds r, r' that may belong to different views v, v' (with possibly identical coordinators $p_\ell, p_{\ell'}$) respectively, if and only if $\forall p_i \in v.set \cap v'.set$ it holds that every $rep_i[i].input_i$ at round r is in $rep_{[}\ell'].msg[i]$ of round r'. Our proof is progressive: Claim 4.4.5 proves that VS is preserved between any two consecutive multicast rounds, Claim 4.4.6 that VS is preserved in two consecutive views with the same coordinator and Claim 4.4.7 preservation in two consecutive view installations where the coordinator changes.

Claim 4.4.5. *VS is preserved between r and r′ where $v = v'$.*

Proof. Remark: In any multicast round, p_ℓ executes *coordinateMcastRnd*() (line 8) only once and a follower executes $followMcastRnd()$ (line 12) only once, since the conditions in lines 7 and 8 are only satisfied in the first iteration after p_ℓ's change of round number is received.

Suppose that there exists an input in $p_i \in v.set$ at round r that is not applied in r' by all processors. By the Remark we note that $fetch()$ is called only once per round to collect input from the environment. This is the only line modifying the *input* field. For followers $rep[i] \leftarrow rep[\ell]$ is in $followMcastRnd()$ (Fig. 4.7, line 22) and for the coordinator it is in the definition of *coordinateMcastRnd*(). At this point followers have already produced side effects for the previous round (using $apply()$) based on the messages and state of the previous round. Similar arguments apply for the coordinator. So line 19 of this procedure populates the *msg* array with messages and *including m*. Then p_ℓ continuously propagates its current replica but cannot change it by the Remark and until condition ($\forall\ p_i \in v.set : rep_\ell[i].(view, status, rnd) = (view_\ell, status_\ell, rnd_\ell)$) (of $roundProceedReay()$) holds again. This ensures that the coordinator will change its *msg* array only when every follower has executed $followMcastRnd()$ and responded back.

Any follower that keeps a previous round number does not allow the coordinator to move to the next round. If the coordinator moves to a new round, it is implied that $rep[i] \leftarrow rep[\ell]$ and thus message m was received by any follower p_i, by our assumptions that the replica is propagated infinitely often and the data links are stabilizing. Thus, by the assumptions, any message m is certainly delivered within the view and round it was sent by the coordinator, thus preserving the VS property, and achieving state replication. □

Claim 4.4.6. *VS is preserved between r and r′ where $v \neq v'$ and $p_\ell = p_{\ell'}$.*

Proof. This implies $repropose_\ell()$ holds and so the coordinator p_ℓ (currently in view v) creates a new proposed view $propV'$. The last condition of *roundProceedReady*() (Fig. 4.7, 10) guarantees that p_ℓ will not execute *coordinateMcastRnd*() and thus will not change its $rep.(state, input, msg)$ fields until all processors of $propV_\ell.set$ have sent their replicas. Followers that accept $propV_\ell$ enter status Propose leading to the installation of the view. What is important is that VS is preserved since no follower is changing $rep.(state, input, msg)$ during this procedure, since they do not execute $followMcastRnd()$ and moreover each sends its replica to p_ℓ by line 15. Once the replicas of all the followers have been collected, the coordinator creates a consolidated *state* and *msg* array of all messages that were either delivered or pending. Then p_ℓ's new replica is communicated to the followers who adopt this state as their own as part of Install (Fig. 4.7, line 14). Thus VS is preserved and once all the processors have replicated the state of the coordinator, a new series of multicast rounds can begin by producing the side effects required by the input collected before the view change. □

Claim 4.4.7. *VS is preserved between r and r′ where $v \neq v'$ and $p_\ell \neq p_{\ell'}$.*

Proof. Practically we would like to see that VS is also preserved when coordinators change. We assume that p_ℓ had a supporting majority throughout R'. Define a matching suffix R'' to prefix R', such that R'' initiates with a loss of supporting majority for p_ℓ. Notice that since Definition 4.4.1 is required to hold, then some other processor with supporting majority in R'', $p_{\ell'}$, will by Lemma 4.4.2 propose the view v' with the highest view ID. By the intersection property of majorities and the fact that a view set can only be formed by a majority set, $\exists p_i \in v.set \cap v'.set$. Thus, the "knowledge" of the system, $(state, input, msg)$ is retained by at least one processor.

As detailed in Claim 4.4.6, if a processor p_i has $noCrd$ = True or is in status Propose it does not incur any changes to its replica. If it entered the Install phase, then this implies that the proposing processor has created a consolidated state that p_i has replicated. What is noteworthy is that whether in status Propose or Install, if the proposer collapses (becomes inactive or suspected), the VS property is preserved. It follows that, once status Multicast is reached by all followers, the system can start a practically infinite number of multicast rounds. □

Thus, by the self-stabilization property of all the components of the system (counter increment algorithm, the data links, the failure detector and multicast) a legal execution is reached in which the VS property is guaranteed and common state replication is preserved. □

Algorithm Complexity

The local memory for this algorithm consists of n copies of two labels, of the encapsulated state (say of size $|S|$ bits) and of other lesser size variables. These give a *space complexity* of order $O(n\beta \log \beta + n|S|)$; recall that $\beta = n^3 cap + 2n^2 - 2n$. *Stabilization time* can be provided by a bound on view creations. It is, therefore, implicit that stabilization is dependent upon the stabilization of the counter algorithm, i.e., $O(n \cdot \beta \cdot t)$, before processors can issue views with identifiers that can be totally ordered. When this is satisfied, then Lemma 4.4.3 suggests that $O(n)$ view creations are required to acquire a coordinator, namely, in the worse case where every processor is a proposer. Once a coordinator is established then Theorem 4.4.4 guarantees that there can be practically infinite multicast rounds (0 to 2^τ).

4.5 Chapter Summary

State-machine replication (SMR) is a service that simulates finite automata by letting the participating processors to periodically exchange messages about their current state as well as the last input that has led to this shared state. Thus, the processors can verify that they are in sync with each other. A well-known way to emulate SMR is to use reliable multicast algorithms that guarantee *virtual synchrony* [71, 133]. To this respect, we have presented the first practically-self-stabilizing algorithm that guarantees virtual synchrony (equipped with a failure detector as the one of [26]),

and used it to obtain a practically-self-stabilizing SMR emulation; within this emulation, the system progresses in more extreme asynchronous executions in contrast to consensus-based SMRs, like the one in [26]. One of the key components of the virtual synchrony algorithm is a practically-self-stabilizing counter algorithm, that establishes an efficient practically unbounded counter, which in turn can be directly used to implement a practically-self-stabilizing MWMR register emulation; this counter scheme extends the one of Alon et al. [27] that implements SWMR registers. It can be considered as a more modular extension of the counter scheme of [27], in comparison to the one of [26], since it is application-independent, and it also requires smaller messages sizes, as it sends label pairs rather than n-sized label vectors.

Chapter 5

Self-Stabilizing Reconfiguration

We now present the first self-stabilizing crash-tolerant reconfiguration service that can recover from an arbitrary system state resulting from transient faults. We have already exposed the novelty of the approach in Chapter 1. We start with the necessary specific system settings and assumptions.

5.1 Specific System Settings and Definitions

We present the required definitions specific to this setting.

5.1.1 Distributed Setting

As already given in Chapter 3, we consider a *dynamic* system where the number of live and connected processors at any time of the computation is bounded by some N such that $N \ll |\mathcal{I}|$. We assume that the processors are crash-prone and that they have knowledge of the upper bound N, but not of the actual number of active processors. Intentional processor departures are modeled as crashes (i.e., a processor leaves the computation without any warning procedure, and the system handles the departure as a crash). New processors may join the system using a joining procedure at any point in time with an identifier drawn from $\mathcal{I}$, such that this identifier is only used by this processor forever. While running the joining procedure they are referred to a *joiners*. A *participant* is an active processor that has joined the computation. Note that N accounts for all active processors, both participants and those that are still joining.

5.1.2 Communication

Due to the possibility of arbitrary faults and because of the dynamic nature of the network, we may not assume that a processor has knowledge of the identifier of another processor with which it is communicating. We, thus, employ two anti-parallel data-links, where every packet of one data-link is identified by the identifiers of the sender and receiver of the data link it participates in. For example, if the communication link connects p_i and p_j, packets of the data link in which p_i acts as the sender that traverse from p_i to p_j are identified by the label p_i, while the label of packets traversing from p_j are extended by adding p_j to the label to form the label p_i, p_j. Any packet with label p_x, p_y arriving to p_i where $x \neq i$ is ignored since this implies that p_i was not the intended recipient. Thus, eventually the data link in which p_i is the sender is implemented by packets with label p_i traversing from p_i to p_j. The analogous holds for the packets implementing the data link in which p_j serves as the sender. Thus, both parties will eventually know the identifier of the other party. They can then regard the token of the data link in which the sender has the greater identifier among them, to be the used token.

Using the underlying packet exchange protocol described, a processor p_i that has received a packet from some processor p_j which did not belong to p_i's failure detector, engages in a two phase protocol with p_j in order to "clean" their intermediate link. This is done before any messages are delivered to the algorithms that handle reconfiguration, joining and applications. We follow the snap-stabilizing data link protocol detailed in [134]. A *snap-stabilizing* protocol is one which allows the system (after faults cease) to behave according to its specification upon its first invocation. We require that every data-link established between two processors is initialized and cleaned straight after it is established. In contrast to [134] where the protocol is run on a tree and initiated from the root, our case requires that each pair of processors takes the responsibility of cleaning their intermediate link. Snap-stabilizing data links do not ignore signals indicating the existence of new connections, possibly some physical carrier signal from the port. In fact, when such a connection signal is received by the newly connected parties, they start a communication procedure that considers the bound on the packet in transit and possibly in buffers too, to clean all unknown packets in transit. They repeatedly send the same packet until more than the round trip capacity acknowledgments arrive.

5.1.3 The (N, Θ)-failure detector

We consider the (N, Θ)-failure detector that uses the token exchange and heartbeat detailed above. This is an extension of the Θ-failure detector used in [26] and discussed in Section 2.5.2. It allows each processor p_i to order other processors according to how recently they have communicated. Each processor p_i maintains an ordered heartbeat (integer) counter vector *nonCrashed*, with an entry corresponding to each processor p_k that exchanges the token (i.e., sends a heartbeat) with p_i. Specifically, whenever p_i receives the token from p_j, it sets the counter corresponding to p_j to 0 and increments the counter of every other processor by one. In this way, p_i manages to rank every processor p_k according to their mutual token exchanges and in relation to the token exchanges that it has performed with some other processor p_j. So the processor that has most recently contacted p_i is the first in p_i's vector.

The technique enables p_i to obtain an estimate on the number of processors n_i that are active in the system; $n_i \leq N$. Assuming that p_c is the most recently crashed processor, then every processor remaining active processor will eventually exchange the token with p_i many times, and their heartbeat counter will be set to zero, while p_c's will be increasing continuously. Eventually, every other active processor's counter will become lower than p_c's and p_c will be ranked last in *nonCrashed*. Moreover, while difference between heartbeat counters of non-crashed processors does not become large, the difference of these counters and that of p_c increases to form a significant ever-expanding "gap". The last processor before the gap is the n_i^{th} processor and this provides an estimate on the number of active processors. These n_i processors are the ones *trusted* by processor p_i to be active. Since there are at most N processors in the computation at any given time, we can ignore any processors that rank below the N^{th} vector entry. If, for example, the first 30 processors in the vector have corresponding counters of up to 30, then the 31^{st} will have a counter much greater than that, say 100; so n_i will be estimated at 30. This estimation mechanism is suggested in [44] and in [45].

5.1.4 The System Reconfiguration Task.

We refer to a *(quorum) configuration* as a bounded size set of processors, which we refer to as config[1]. We say that the system has a valid configuration, called *conflict-free*, when no two processors that are active in the system store different values in their config variables. Note that the system does not accept a config to be empty, i.e., to take the value of the empty set. The system assigns the symbol $\perp$ to config whenever it detects a configuration conflict and is thus in the process of *configuration reset*. By the end of the reset process, the system shall store in all config variables identical and valid configuration values. Note that the reset configuration process is the recovery strategy from transient faults; a process that ends when the system state returns to be conflict-free.

Once the system is conflict-free, only *participants* can call for the establishment of new configurations, proposing a non-empty trusted participant set to replace the current configuration. Newly arrived processors can become participants, as long as no reconfiguration occurs. While reconfiguration is in progress, the system may block changes to the participation set and stop considering any additional reconfiguration requests. Thus, when there are no configuration conflicts, and all participating processors have the same view on the participation set, the system's ability to replace the existing configuration with a proposed one depends on the participant's crash rate. This recovery strategy implies that the system is always able to converge to a valid configuration even when the churn rate had been too high with respect to crashing participants. For this purpose, the (N, Θ)-failure detector needs to be *eventually* and *temporarily* reliable, in order to allow the recovery strategy to attain a conflict-free configuration; but after that, it can be unreliable, as long as the crash rate of participating processors is such that the system can replace the current configuration on timewith a new one. A violation of the latter assumption is a transient fault from which the system recovers via the configuration reset process.

Complexity metric. We define an *asynchronous round* of a fair execution R as the shortest prefix of R in which every correct processor p_i completed an iteration I_i, and all messages p_i sent during I_i were received.

[1]In the context of self-stabilization, the term (quorum) configuration must not be confused with the term (system) configuration [19]. We consistently, use *state* to mean a *system state*, and we use *configuration* to mean *quorum configuration*.

5.2 Solution Outline

Our scheme comprises of two layers that appear as a single "black-box" module to an application that uses the reconfiguration service. The objective is to provide the application with a *conflict-free* configuration, such that no two active processors consider different configurations.

The first layer, called *Reconfiguration Stability Assurance* or *recSA* for short (detailed in Section 5.3), is mainly responsible for detecting configuration conflicts (that could be a result of transient faults). It deploys a *brute-force* technique for converging to a conflict-free new configuration. It also employs another technique for *delicate* configuration replacement when a processor notifies that it wishes to replace the current configuration with a new set of participants. For both techniques, processors use the (N, Θ)-failure detector (detailed in Section 2.5.2) to obtain membership information, and configuration convergence is reached when failure detectors have temporal reliability. Once a uniform configuration is installed, the failure detectors' reliability is no longer needed and from then on our liveness conditions consider unreliable failure detectors.

The decision for requesting a delicate reconfiguration is controlled by the other layer, called *Reconfiguration Management* or *recMA* for short (detailed in Section 5.4). Specifically, if a processor suspects that the dependability of the current configuration is under jeopardy, it seeks to obtain a majority approval from the active *members* of the current configuration, and request a (delicate) reconfiguration from *recSA*. Moreover, in the absence of such a majority (e.g., configuration replacement was not activated "on time" or the churn assumptions were violated), the *recMA* can aim to control the recovery via a *recSA* reconfiguration request. Note that the current participant set can, over time, become different than the configuration member set. As new members arrive and others leave, changing the configuration based on system membership would imply a high frequency of (delicate) reconfigurations, especially in the presence of high churn. We avoid unnecessary reconfiguration requests by requiring a weak liveness condition: if a majority of the configuration set has not collapsed, then there exists at least one processor in the failure detector of each active processor that is known to trust this majority. Such active configuration members can aim to replace the current configuration with a newer one (that would provide an approving majority for prospective reconfigurations) without the use of

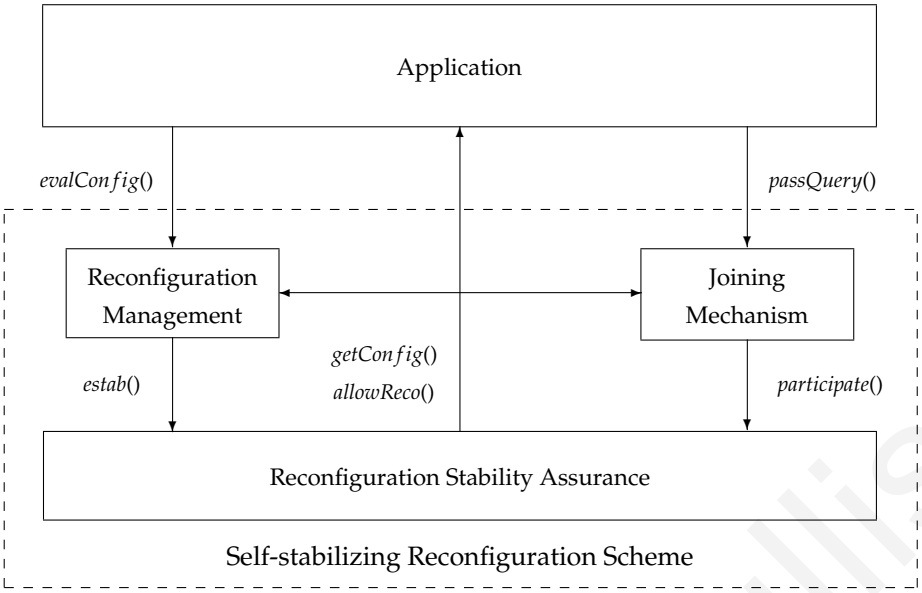

Figure 5.1: The reconfiguration scheme modules internal interaction and the interaction with the application. The Reconfiguration Stability Assurance (*recSA*) layer provides information on the current configuration and on whether a reconfiguration is not taking place using the *getConfig*() and *allowReco*() interfaces. This is based of local information. The Reconfiguration Management (*recMA*) layer uses the prediction mechanism *evalConfig*() which is application based to evaluate whether a reconfiguration is required. If a reconfiguration is required, *recMA* initiates it with *estab*(). Joining only proceeds if a configuration is in place and if no reconfiguration is taking place. When the joining mechanism has received a permission to access the application (using *passQuery*()) it can then join via *participate*(). The direction of an arrow from a module A to a module B illustrates the transfer of the specific information from A to B.

the brute-force stabilization technique.

Joining mechanism: We complement our reconfiguration scheme with a self-stabilizing joining mechanism (detailed in Section 5.5) that manages and controls the inclusion of new processors into the system. Caution is required here so that newly joining processors do not "contaminate" the system state with stale information (due to arbitrary faults). For this, together with other techniques, we follow a snap-stabilizing data link protocol (cf. Section 2.5.2). We have designed our joining mechanism so that the decision of whether new members should be included in the system or not is *application-controlled*. In this way, the churn (regarding new arrivals) can be "fine-tuned" based on the application requirements; we have modeled this by having joining processors obtaining approval from a majority of the members of the current configuration (if no reconfiguration is taking place). These, in turn, provide such approval if the application's (among other) criteria are met. We note that in the event of transient faults, such as an unavailable approving majority, *recSA* ensures

recovery via brute-force stabilization that includes all active processors.

Figure 5.1 depicts the interaction between the modules and the application. We continue to detail the three modules: *recSA* in Section 5.3, *recMA* in Section 5.4, and the Joining Mechanism in Section 5.5. We conclude by suggesting how the services of Chapter 4 could take advantage of the reconfiguration scheme to run in a dynamic environment.

5.3 Reconfiguration Stability Assurance

We present the Reconfiguration Stability Assurance layer (*recSA*), which is a self-stabilizing algorithm for assuring a correct and consistent configuration while allowing the updates from the Reconfiguration Management layer (Section 5.4). We first describe the algorithm (Section 5.3.1) and then we prove its correctness (Section 5.3.2).

5.3.1 Algorithm Description

We first present an overview of the algorithm and then proceed to a line-by-line description.

Overview

The *recSA* layer uses a self-stabilizing algorithm, Algorithm 5, for assuring correct configuration while allowing the updates from the reconfiguration management layer. Algorithm 5 guarantees that (1) all active processors have eventually identical copies of a single configuration, (2) when participants notify the system that they wish to replace the current configuration with another, the algorithm selects one proposal and replaces the current configuration with it, and (3) joining processors can become participants eventually.

The algorithm combines two techniques: one for *brute force stabilization* that recovers from stale information and a complementary technique for *delicate (configuration) replacement*, where participants jointly select a single new configuration that replaces the current one. As long as a given processor is not aware of ongoing configuration replacements, Algorithm 5 merely monitors the system for stale information, e.g., it makes sure that all participants have a single (non-empty) config-

Variables: The following arrays consider both p_i's own value (entry i) and p_j's most recently received value (entry j).

config[]: an array in which config[i] is p_i's view on the current configuration. Note that p_i assigns the *empty (configuration)* value $\perp$ after receiving a conflicting (different) non-empty configuration value.

FD[]: an array in which FD[i] represents p_i's failure detector. Note that we consider only the trusted processors rather than the suspected ones. Namely, crashed processors are eventually suspected. **FD**[].**part** is the participant set, where FD[i].*part* is an alias for $\{p_j \in \text{FD}[i] : \text{config}[j] \neq \sharp\}$ and FD[j].*part* refers to the last value received from p_j. Namely, the FD field of every message encodes also this participation information.

prp[] is an array of pairs $\langle phase \in \{0, 1, 2\}, set \subseteq \mathcal{I} \rangle$, where prp[$i$] refers to p_i's configuration replacement notifications. In the pair prp[i], the field *set* can either be $\perp$ (indicating 'no value') or the proposed processor set.

all[] is an array of Booleans, where all[i] refers to the case in which p_i observes that all trusted processors had noticed its current (maximal) notification and they hold the same notification.

allSeen: a list of processors p_k for which p_i received the all[k] indication.

echo[] is an array in which echo[i] is (FD[i].*part*, prp[i], all[i])'s alias and echo[j] refers to the most recent value that p_i received from p_j after p_j had responded to p_i with the most recent values it received from p_i.

Figure 5.2: Variables and Operators for Self-stabilizing Reconfiguration Stability Assurance; code for p_i.

uration. During these periods the algorithm allows the invocation of configuration replacement processes (via the *estab*(set) interface, triggered by the Reconfiguration Management layer) as well as the acceptance of joining processors as participants (via the *participate*() interface, triggered by the Joining layer). During the process of configuration replacement, the algorithm selects a single configuration proposal and replaces the current one with that proposal before returning to monitor for configuration disagreements (Figure 5.4).

While the system reconfigures, there is no immediate need to allow joining processors to become participants. By temporarily disabling this functionality, the algorithm can focus on completing the configuration replacement using the current participant set. To that end, only participants broadcast their states at the end of the do forever loop (line 14), and only their messages arrive to the other active processors (line 15). Joining processors receive such messages, but cannot broadcast before their safe entry to the participant set via the function *participate*() (Fig. 5.3, line 6), which enables p_i's broadcasting. Note that non-participants monitor the intersec-

1 **Interface functions**:
2 **function** $chsConfig()$ = **return(choose**($\{\text{config}[k]\}_{p_k \in \text{FD}[i]} \setminus \{\sharp\}$)), where **choose**($\emptyset$) = $\perp$ **else** **choose**(set) $\in$ set;
3 **function** $getConfig()$ = {**if** $allowReco()$ **then return**($chsConfig()$) **else return(**config[i])};
4 **function** $allowReco()$ = ($allSeen() \wedge \bigwedge_{p_k \in \text{FD}[i].part} echo(k)) \wedge (\neg((p_i \notin (\cap_{p_j \in \text{FD}[i] \setminus \{p_i\}} \text{FD}[j])) \vee (|\{\text{config}[k]\}_{p_k \in \text{FD}[i]} \setminus \{\sharp\}| > 1) \vee (\{\text{FD}[i].part\} \neq \{\text{FD}[k].part, \text{echo}[k].FD.part\}_{p_k \in \text{FD}[i]}) \vee (\exists p_k \in \text{FD}[i] : (config[k] = \perp) \vee ((prp[k], \text{all}[k]) \neq (\text{dfltNtf}, true))))$;
5 **function** $estab$(set) = {**if** $(allowReco() \wedge (set \notin \{\text{config}[i], \emptyset\}))$ **then** $(\text{prp}[i], \text{all}[i], \text{allSeen}) \leftarrow (\langle 1, \text{set}\rangle, false, \emptyset)$};
6 **function** $participate()$ = {**if** $(allowReco())$ **then** config[i] $\leftarrow$ $chsConfig()$};
7 **Constants, macros and functions**: dfltNtf = $\langle 0, \perp\rangle$ /* the default notification tuple */
8 **macro** $myAll(k)$ = **return** (all[k] $\vee$ ($\exists p_\ell \in$ allSeen : $i = k \wedge \text{prp}[i].phase + 1 \bmod 3 = \text{prp}[\ell].phase$));
9 **macro** $degree(k)$ = **return** ($2 \cdot \text{prp}[k].phase + |\{1 : \text{myAll}(k)\}|$); /* p_k's most recently received prp*/
10 **macro** $corrDeg(k, k')$ = **return** ($\{degree(k), degree(k')\} \in \{\{x, x+1\}, \{x, x\} : x \in \{0, \ldots, 4\}\} \cup \{\{0, 5\}, \{5, 5\}\}$);
11 **macro** $echoNoAll(k)$ = **return** ($\{(\text{FD}[i].part, \text{prp}[i])\} = \{(\text{echo}[k].part, \text{echo}[k].\text{prp})\}$);
12 **macro** $echo(k)$ = **return** ($\{(\text{FD}[i].part, \text{prp}[i], \text{myAll}(i))\} = \{\text{echo}[k]\} \wedge degree(k) - degree(i) (\bmod 6) \in \{0, 1\}$);
13 **macro** $configSet(val)$ = {**foreach** $p_k \in \mathcal{I}$ **do** $(\text{config}[k], \text{prp}[k]) \leftarrow (val, \text{dfltNtf})$}; /* access to p_i's config */
14 **macro** $increment(prp)$ = {**case** $(prp.phase)$ **of** 1 : **return**($(\langle 2, prp.set\rangle), false$); 2 : **return**((dfltNtf, $false$)); **else return** ((prp[i], all[i]));}
15 **macro** $allSeen()$ = ($\text{all}[i] \wedge \text{FD}[i].part \subseteq (\text{allSeen} \cup \{p_i\})$);
16 **macro** $modMax()$ ={**if** $(1 \in Phs \wedge 2 \notin Phs \wedge \text{prp}[i].phase \neq \max Phs)$ **then** {allSeen $\leftarrow$ $\emptyset$; **return** $\max Phs$} **else return** prp[i].$phase$, **where** $Phs = \{\text{prp}[k].phase\}_{p_k \in \text{FD}[i].part}$}
17 **macro** $maxNtf()$ = {**if** $\{(degree(k) - degree(i)) (\bmod 6)\}_{p_k \in \text{FD}[i].part} \not\subseteq \{0, 1\}$ **then return** prp[i] **else return** $\langle modMax(), \max_{\text{lex}}\{\text{prp}[k].set\}_{p_k \in FD[i].part}\rangle$}

Figure 5.3: Macros and Interface Functions for the Reconfiguration Stability Assurance module (Algorithm 5).

tion between the current configuration and the set of active participants (line 6). In case it is empty, the processors (participants or not) essentially begin a brute-force stabilization (outlined below) where joining processors are no longer blocked from becoming participants.

Brute-force stabilization. The algorithm detects the presence of stale information and recovers from these transient faults. *Configuration conflicts* are one of several kinds of such stale information and they refer to differences in the field config, which stores the configuration values. Processor p_i can signal to all processors that it had detected stale information by assigning $\perp$ to config$_i$ and by that start a *reset process* that nullifies all config fields in the system (lines 6 and 8). Algorithm 5 uses the brute-force technique for letting processor p_i to assign its set of trusted processors to config$_i$ (line 9). This set is provided by p_i's failure detector FD$_i$. Note that by the

Algorithm 5: Self-stabilizing Reconfiguration Stability Assurance; code for processor p_i

```
1  do forever begin
2      foreach p_k ∉ FD[i].part do (config[k], prp[k]) ← (♯, dfltNtf);        /* clean after crashes */
3      prp[i] ← maxNtf();
4      all[i] ← ⋀_{p_k∈FD[i].part}(echoNoAll(k));
5      foreach p_k ∈ FD[i].part : all[k] do allSeen ← allSeen ∪ {p_k};
6      if ((∃p_k:((prp[k] = ⟨0, s⟩) ∧ (s ≠ ⊥)) ∨ (config[k] ∈ {⊥, ∅}))) ∨ (∃p_k ∈ FD[i].part:¬corrDeg(i, k))∨
       ({p_k ∈ FD[i].part : prp[i].phase + 1 mod 3 = prp[k].phase} ⊈ allSeen) ∨ (∃p_ℓ ∈ FD[i].part :
       prp[ℓ] = ⟨2, •⟩ ∧ |{prp[k].set ≠ ⊥}_{p_k∈FD[i].part}| > 1) ∨ (({(FD[i], FD[i].part)}
       = {(FD[k], FD[k].part)}_{p_k∈FD[i].part}) ∧ ((config[i] ≠ ⊥) ∧ ((config[i] ∩ FD[i].part) = ∅))) then
       configSet(⊥)
7      if ({prp[k].phase}_{p_k∈FD[i].part} = {0}) then                        /* when no notification arrived */
8          if |{config[k]}_{p_k∈FD[i]} \ {⊥, ♯}| > 1 then configSet(⊥); /* nullify the configuration upon conflict */
9          if (config[i] = ⊥ ∧ |{FD[j] : p_j ∈ FD[i]}| = 1) then configSet(FD[i]); /*reset during admissible
           runs */
10     else
11         if (allSeen() ∧ ⋀_{p_k∈FD[i].part} echo(k)) then ((prp[i], all[i]), allSeen) ← (increment(prp[i]), ∅);
12         if prp[i].phase = 2 then config[i] ← prp[i].set;
13     if config[i] ≠ ♯ then
14         foreach p_j ∈ FD[i] do send(⟨FD[i], config[i], prp[i], myAll(i), (FD[j].part, prp[j], all[j])⟩)
15 upon receive m = ⟨FD, config, prp, all, echo⟩ from p_j do
   (FD[j], config[j], prp[j], all[j], echo[j]) ← m ;
16 upon interrupt p_i's booting do foreach p_k do (config[k], prp[k], all[k]) ← (♯, dfltNtf, false);
   echo[k] ← (FD[k].part, prp[k], all[k]);
```

end of the brute-force process, all active processors (joining or participant) become participants. We show that eventually all active processors share identical (non-$\perp$) config values by the end of this process.

Delicate (configuration) replacement. Participants can propose to replace the current configuration with a new one, set, via the *estab*(set) interface. This replacement uses the *configuration replacement* process, which for the purposes of the overview, we abstract as the automaton depicted in Figure 5.4. When the system is free from stale information, the configuration uniformity invariant (of the config field values) holds. Then, any number of calls to the *estab*(set) interface starts the configuration replacement process, which controls the configuration replacement using the following three phases: (1) selecting (deterministically and uniformly) a single proposal (while verifying the eventual absence of "unselected" proposals), (2) replacing (deterministically and uniformly) all config fields with the jointly selected proposal, and (3) bringing back the system to a state in which it merely tests for stale information.

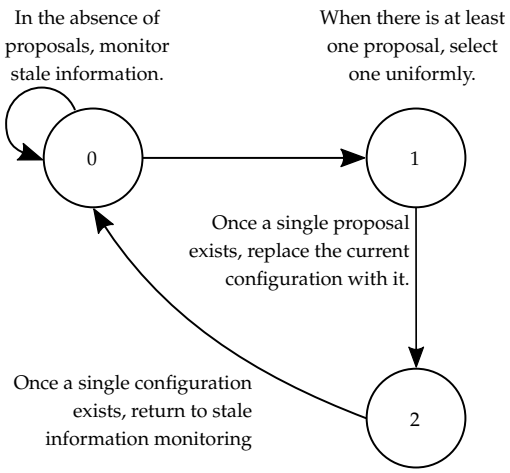

Figure 5.4: The configuration replacement automaton.

The configuration replacement process requires coordinated phase transition. Algorithm 5 lets processor p_i to represent proposals as $\text{prp}_i[j] = (phase, set)$, where p_j is the processor from which p_i received the proposal, $phase \in \{0, 1, 2\}$ and *set* is a processor set or the null value, $\perp$. The *default proposal*, $\langle 0, \perp \rangle$, refers to the case in which prp encodes "no proposal". When p_i calls the function *estab*(set), it changes prp to $\langle 1, set \rangle$ (Fig. 5.3, line 5) as long as p_i is not aware of an ongoing configuration replacement process, i.e., *allowReco*() returns *true*. Upon this change, the algorithm disseminates $\text{prp}_i[i]$ and by that guarantees eventually that *allowReco*() returns false for any processor that calls it. Once that happens, no call to *estab*(set) adds a new proposal for configuration replacement and no call to *participate*() lets a joining processor to become a participant (Fig. 5.3, line 6). The algorithm can then use the lexical value of the $\text{prp}_i[]$'s tuples to select one of them deterministically. To that end, each participant ensures that all other participants report the same tuples by waiting until they "echo" back the same values as the ones it had sent to them. Once that happens, participant p_i makes sure that the communication channels do not include other "unselected" proposals by raising a flag ($\text{all}_i = true$) and waiting for the echoed values of these three fields, i.e., participant set, $\text{prp}_i[i]$ and all_i. This waiting lasts until the echoed values match the values of any other active participant in the system (while monitoring their well-being). Before this participant proceeds, it makes sure that all active participants have noticed its phase completion. Each processor p maintains the allSeen variable; a set of participants that have noticed p's phase completion (line 5) and are thus added to p's allSeen set.

The above mechanism for phase transition coordination allows progression in a unison fashion. Namely, no processor starts a new phase before it has seen that all

other active participants have completed the current phase and have noticed that all other have done so (because they have identical participant set, prp and all[] values). This is the basis for emulating every step of the configuration replacement process (line 12) and making sure that the phase 2 replacement occurs correctly before returning to the monitoring phase 0, in which the system simply tests for stale information. We show that since the failure detectors monitor the participants' well-being, this process terminates. This procedure is illustrated with the automaton of Figure 5.4. The variables for Algorithm 5 can be found in Figure 5.2, and the macros and interface functions in Figure 5.3.

Detailed description

We now proceed to a detailed, line-by-line description of Algorithm 5.

Variables. The algorithm uses a number of fields that each active participant broadcasts to all other system processors. The processor stores the values that they receive in arrays. Namely, we consider both p_i's own value (the i-th entry) and p_j's most recently received value (the j-th entry).

- The field config[] is an array in which config[i] is p_i's view on the current configuration. Note that p_i assigns the *empty (configuration)* value $\perp$ after receiving a conflicting non-empty configuration value, i.e., the received configuration is different than p_i's configuration. We use the symbol $\sharp$ for denoting that processor p_i is not a participant, i.e., $\text{config}_i[i] = \sharp$.

- The field FD[] is an array in which FD[i] represents p_i's failure detector of trusted processors (without the list of processors that are suspected to be crashed).

- FD[].*part* is the participant set, where FD[i].*part* is an alias for $\{p_j \in \text{FD}[i] : \text{config}[j] \neq \sharp\}$ and FD[j].*part* refers to the last value received from p_j. Namely, the FD field of every message encodes also this participation information.

- The field prp[] is an array of pairs $\langle phase \in \{0, 1, 2\}, set \subseteq \mathcal{I} \rangle$, where prp[$i$] refers to p_i's configuration replacement notifications. In the pair prp[i], the field *set* can either be $\perp$ ('no value') or the proposed processor set.

- The field all[] is an array of Booleans, where all[i] refers to the case in which p_i observes that all trusted processors have noticed its current (maximal) notification and they hold the same notification.

- The field echo[] is an array in which echo[i] is an alias of (FD[i].*part*, prp[i], all[i]) and echo[j] refers to the most recent value that p_i has received from p_j after p_j had responded to p_i with the most recent values it got from p_i.

- The variable allSeen is a set that includes the processors p_k for which p_i received the all[k] indication.

Constants, functions and macros. The constant dfltNtf (Fig. 5.3, line 7) denotes the default notification (of a configuration replacement proposal) tuple $\langle 0, \perp \rangle$. The following functions define the interface between Algorithms 5 and 6 (Reconfiguration Management layer) and the Joining Mechanism (Algorithm 7). Note that the behavior which we specify below considers legal executions. All the line references below can be found in Figure 5.3.

- The function $chsConfig()$ (line 2) returns config whenever there is a single such non-$\sharp$ value. Otherwise, $\perp$ is returned.

- The function $getConfig()$ (line 3) allows Algorithms 6 and 7 to retrieve the value of the current quorum configuration, i.e., $\text{config}_i[i]$. We note that during legal executions, this value is a set of processors whenever p_i is a participant. However, this value can be $\sharp$ whenever p_i is not a participant and $\perp$ during the process of configuration reset.

- The function *allowReco*() (line 4) returns $false$ whenever (1) p_i was not recognized as a trusted processor by a processor that p_i trusts, (2) there are configuration conflicts, (3) the participant sets have yet to stabilize, (4) there is an on-going process of brute force stabilization, or (5) there is a delicate (configuration) replacement in progress. This part of the interface allows Algorithms 6 and 7 to test for the presence of local evidence according to which Algorithm 5 shall disable delicate (configuration) replacement and joining to the participant set.

- The function *estab*(set) (line 5) provides an interface that allows the *recMA* layer to request from Algorithm 5 to replace the current quorum configuration

with the proposed set, which is a non-empty group of participants. Note that Algorithm 5 disables this functionality whenever *allowReco*() = *false* or *set* = config[*i*].

- The function *participate*() (line 6) provides an interface that allows the Joining Mechanism to request from Algorithm 5 to let p_i join the participant set, which is the group that can participate in the configuration and request the replacement of the current configuration with another (via the *estab*(set) function). Note that Algorithm 5 disables this functionality whenever *allowReco*() = *false* and thus there exists a single configuration in the system, i.e., the call to *chsConfig*() (line 2) returns the single configuration that all active participants store as their current quorum configuration. This is except for the case in which $\{\text{config}_i[k]\}_{p_k \in \text{FD}_i[i]} \setminus \{\sharp\} = \emptyset$. Here, *chsConfig*() returns $\perp$, which starts a reset process in order to deal with a complete collapse where the quorum system includes no active participants.

Algorithm 5 uses the following macros.

- The macro *myAll*(*k*) (line 8) returns the disjunction of the value stored in all[*k*] and the value of $p_i \in$ allSeen.
- The macro *degree*(*k*) (line 9) calculates the degree of p_k's most-recently-received notification degree, which is twice the notification phase plus one whenever all participants are using the same notification (and zero otherwise), where each notification is a configuration replacement proposal.
- The macro *corrDeg*(*k*, *k'*) (line 10) tests whether p_k and $p_{k'}$ have degrees that differ by at most one when considering operations in (mod 6).
- The macros *echoNoAll*(*k*) and *echo*(*k*) (lines 11, and 12 respectively) test whether p_i was acknowledged by all participants for the values it has sent. The former function considers just the fields that are related to its own participant set and notification, whereas the latter considers also the field all[].
- The macro *modMax*() (line 16) assumes that no two processors in FD[*i*].*part* have two notifications that p_i stores for which the degree differs by more than one. The function returns that maximum phase value when considering an order relation that is based on modulo 3 arithmetics.

- The macro $maxNtf()$ (line 17) selects a notification with the maximal lexicographical value or returns $\perp$ in the absence of notification that is not the default (phase 0) notification. We define our lexicographical order between prp_1 and prp_2, as $\text{prp}_1 \leq_{lex} \text{prp}_2 \iff ((\text{prp}_1.phase < \text{prp}_2.phase) \vee ((\text{prp}_1.phase = \text{prp}_2.phase) \wedge (\text{prp}_1.set \leq_{lex} \text{prp}_2.set)))$, where $\text{prp}_1.set \leq_{lex} \text{prp}_2.set$ can be defined using a common lexical ordering and by considering sets of processors as ordered tuples that list processors in, say, an ascending order with respect to their identifiers.

- The macro $configSet(val)$ (line 13) acts as a wrapper function for accessing p_i's local copies of the field config. This macro also makes sure that there are no (local) active notifications.

- The macro $increment(phs)$ (line 14) performs the transition between the phases of the delicate configuration replacement technique.

- The macro $allSeen()$ (line 15) tests whether all active participants have noticed that all other participants have finished the current phase.

The do forever loop. A line-by-line walkthrough of the pseudocode of Algorithm 5 follows.

Cleaning up, removal of stale information and invariant testing. The do forever loop starts by making sure that non-participant nodes cannot have an impact on p_i's computation (line 2) before testing that p_i's state does not include any stale information (line 6). Algorithm 5 tests for the following four types of stale information.

Type (1) – All the notifications (of configuration replacement proposals) are valid,

Type (2) – There are no configuration conflicts or an active reset process,

Type (3) – The phase information (including the set *allSeen*) are not out of synch, and

Type (4) – There are active participants in config.

These are formalized in Definition 5.3.1. In case any of these tests succeeds, the algorithm starts a configuration *reset process* by calling $configSet(\perp)$ also known as brute-force stabilization.

The brute-force stabilization technique. As long as no active notifications are present locally (line 7), every processor performs this technique for transient fault recovery. In the presence of configuration conflicts, the algorithm starts the configuration reset process (line 8). Moreover, during the configuration reset process, the

algorithm waits until all locally trusted processors report that they trust the same set of processors (line 9).

The delicate replacement technique — phase synchronization. This technique synchronizes the system phase transitions by making sure that all active participants work with the same notification.

Each active participant tests whether all other trusted participants have echoed their current participant set and notifications and also that they have the same values with respect to these two fields (line 3). The success of this test assigns *true* to the field $\text{all}_i[i]$. The algorithm then extends this test to include also the field all[] (line 5), where $\text{all}_i[j]$ refers to the case in which node p_j has reported to p_i that it has passed the previous test (line 3). Upon the success of this test with respect to participant p_k, the algorithm adds p_k to the set allSeen_i. Once processor p_i receives reports from all participants that the current phase is over, it moves to the next phase (line 11).

The delicate replacement technique — finite-state-machine emulation. Each of the three phases represent an automaton state (line 11); recall Figure 5.4 and the configuration replacement process discussed in the Algorithm's description overview. During phase 1, the system converges to a single notification. During phase 2, the system replaces the current configuration with the proposed one. Next, the system returns to its ideal state, i.e., phase 0, which allows new participants to join, as well as further reconfigurations (line 12).

Message exchange and the control of newly arrived processors. When a participant finishes executing the do forever loop, it broadcasts its entire state (line 14). Once these messages arrive, processor p_i stores them (line 15). The only way for a newly arrived processor to start executing Algorithm 5 is by responding to an interrupt call (line 16). This procedure notifies the processor state and makes sure that it cannot broadcast messages (line 14). The safe entry of this newly arrived processor to the participant set, is via the function *participate*() (Fig. 5.3, line 6), which enables p_i's broadcasting. Thus, a non-participant merely follows the system messages until the function *allowReco*() returns *true* and allows its join to the participant set by a call (of the Joining Mechanism) to the function *participate*() (Fig. 5.3, line 6).

5.3.2 Correctness

We begin by defining three type of executions, that play a key role in the algorithm's correctness.

An execution R is *fair* when every active processor p_i that has a step a_i applicable infinitely often, executes a_i infinitely often. A fair execution R is *admissible* when throughout R the failure detector values of active processors are identical, do not change, and consist of only themselves (the set of active processors). I.e., $\forall c \in R$, $p_i, p_j \in \mathcal{I}$ that are active in R, it holds $\text{FD}_i[i] = \text{FD}_j[j]$ and $p_k \in \text{FD}_i[i] \iff p_k$ is active. We also consider a "weaker" form of admissible executions: A fair execution R is *admissible w.r.t. participants* when $\text{FD}_i[i].part = \text{FD}_j[j].part$ and $p_k \in \text{FD}_i[i] \iff p_k$ is active in R, where $\text{FD}_i[i].part$ is p_i's view on the participation set. Note that the set of all admissible executions contains the set of all admissible w.r.t. participants executions, because the former considers joining processors whereas the latter does not.

To guarantee the success of a reset process we assume that the system eventually reaches an admissible execution until the reset process terminates. In some sense, the above assumption implies that the algorithm completes the reset process by having a temporal access to reliable failure detectors. However, once Algorithm 5 completes this process, safety holds forever thereafter because, as shown in the proof, the system cannot include stale information (or start another reset process) after the reset process termination. In other words, once the reset process establishes safety, the failure detector reliability is no longer needed, because the success of Algorithm 5 to achieve its task does not require that the system reaches admissible executions, and liveness is conditioned by the failure detector's unreliable signals. In the case of delicate (configuration) replacement, it suffices for the system to reach an admissible w.r.t. participants execution (since new processor joining is blocked), unless it ends up running the reset process (in which case all active processors, including joining ones become participants). In practice, one expects admissible w.r.t. participants executions to be realized much faster than admissible executions; furthermore, one expects the system to undergo delicate replacements more frequently than reset replacements. Hence the weaker form of admissibility seems to be leading to more efficient implementations of the proposed reconfiguration scheme.

For bounding the convergence time of our reconfiguration scheme we will be

using the notion of *asynchronous rounds*, defined below. We regard this as a natural complexity measure for an asynchronous setting, such as the one we consider in this work.

An *asynchronous round* of a fair execution R is the shortest prefix of R in which every active processor p_i completes an iteration (of the do forever loop) I_i and all messages p_i sent during I_i were received. When R is an admissible execution w.r.t. participants, then an asynchronous round is defined over only the active processors that are participants in R.

Next, we provide an outline of the proof and then proceed in steps to establish the correctness of the algorithm.

Proof Outline

The correctness proof of Algorithm 5 shows that eventually all stale information is cleaned (line 2) or is detected and causes a configuration reset with a call to $configSet(\perp)$ (line 6). This is proved to result to a common configuration being installed. In particular, Definition 5.3.1 distinguishes the four types of configuration conflicts and stale information that can exist in the system. Following the definition, Lemma 5.3.1 proves that all notifications are valid and so stale information of **type-1** stop existing in the system. Lemma 5.3.2 proves that configuration conflicts seize, thus **type-2** stale information stops. Lemmas 5.3.8 and 5.3.15 show that all processors the follow the phase information in lock step or the system is led to a reset. Both cases imply that there is no **type-3** stale information eventually. Finally, Lemma 5.3.7 establishes that there is no non-$\perp$ configuration for which the local failure detector of a processor sees no active members, and thus there is no **type-4** staleness. The above result in Theorem 5.3.16, which proves convergence within $O(N)$ asynchronous rounds. The correctness proof is completed with the closure proof (Theorem 5.3.17), which shows that since there are no configuration conflicts after convergence, the only way to change a view is via a call to $estab(set)$ by the upper layer *recMA*, and this results to a view being replaced.

Configuration conflicts and stale information

We begin by classifying the stale information into four types.

Definition 5.3.1. *We say that processor p_i in system state c has a stale information in c of:*

- **type-1:** *when* $(\exists p_k : ((\text{prp}_i[k] = \langle 0, s\rangle) \wedge (s \neq \perp)))$ *(cf. line 6).*

- **type-2:** *when* $(\exists p_k : (\text{config}_i[k] \in \{\perp, \emptyset\}))$ *(cf. line 6) or c encodes a* (configuration) conflict*, i.e., there are two active processors* p_i *and* p_j *for which* $\text{config}_i[i] \neq \text{config}_j[j]$*,* $\text{config}_i[i] \neq \text{config}_i[j]$*, or* $m_{j,i}.\text{config} \neq \text{config}_i[i]$ *in any message in the communication channel from* p_i *to* p_j*.*

- **type-3:** *when* $(\exists p_k \in \text{FD}_i[i].part{:}\neg corrDeg(i,k)) \vee (\{p_k \in \text{FD}_i[i].part : \text{prp}_i[i].phase + 1 \bmod 3 = \text{prp}_i[k].phase\} \not\subseteq \text{allSeen}_i) \vee (\exists p_\ell \in \text{FD}_i[i].part : \text{prp}_i[\ell] = \langle 2, \bullet\rangle \wedge |\{\text{prp}_i[k].set \neq \perp\}_{p_k \in \text{FD}_i[i].part}| > 1)$

- **type-4:** *when* $((\{(\text{FD}_i[i], \text{FD}_i[i].part)\} = \{(\text{FD}_i[k], \text{FD}_i[k].part)\}_{p_k \in \text{FD}_i[i].part}) \wedge ((\text{config}_i[i] \neq \perp) \wedge ((\text{config}_i[i] \cap \text{FD}_i[i].part) = \emptyset))$.

Lemma 5.3.1 (No type-1 stale information)**.** *Let R be a fair execution. Within* $O(1)$ *asynchronous rounds, the system reaches a state* $c \in R$ *in which the invariant of no type-1 stale information holds thereafter.*

Proof. Let $c \in R$ be a system state in which processor p_i has an applicable step a_i that includes the execution of the do forever loop (line 1 to 14). Every processor p_i takes this step within $O(1)$ asynchronous rounds. We note that immediately after a_i, processor p_i has no type-1 stale information (line 6 removes it). Moreover, that removal always occurs before a_i sends any message m (line 14). Therefore, within $O(1)$ asynchronous rounds, for every active processor p_j that receives message m from p_i (line 15), it holds that $(m.phase = 0) \iff (m.\text{prp} = (0, \perp))$ as well as for every item $\text{prp}_i[k] : p_k \in \mathcal{I}$. Once this invariant holds for every pair of active processors p_i and p_j, the system reaches state c. We conclude the proof by noting that Algorithm 5 never assigns to $\text{prp}_i[j]$ a values that violates this claim invariant. □

Dealing with explicit and spontaneous replacements

We say that a processor p_i's state encodes a *(delicate) replacement* when $\text{prp}_i[j] \neq \langle 0, \perp\rangle$ and we say that a message $m_{i,j}$ in the channel from p_i to p_j encodes a *(delicate) replacement* when $m_{i,j}.\text{prp} \neq \langle 0, \perp\rangle$. Given a system execution R, we say that R *does not include an explicit (delicate) replacement* when throughout R no node p_i calls *estab*(). Suppose that execution R does not include an explicit (delicate) replacement and yet there is a system state $c \in R$ in which a processor state or a message in the

communication channels encodes a (delicate) replacement. In this case, we say that R includes a *spontaneous (delicate) replacement*.

Lemma 5.3.2 (No type-2 stale information)**.** *Let R be an admissible execution that does not include explicit (delicate) replacements nor spontaneous ones. Within $O(1)$ asynchronous rounds, the system reaches a state $c \in R$ in which the invariant of no type-2 stale information holds thereafter.*

Proof. Note that any of R's steps that includes the do forever loop (line 1 to 14) does not run lines 5 to 12 (since R does not include an explicit nor spontaneous replacement). If R's starting system state does not include any configuration conflicts, we are done. Suppose that R's starting system state does include a conflict, i.e., $\exists p_i, p_j \in \mathcal{I} : (\text{config}_i[i] = \perp) \vee (\text{config}_i[i] \neq \text{config}_i[j]) \vee (\text{config}_i[i] \neq \text{config}_j[j])$ or there is a message, $m_{i,j}$, in the communication channel from p_i to p_j, such that the field $(m_{i,j}.\text{config}[k] = \perp) : p_k \in \text{FD}_i[i] \vee (m_{i,j}.\text{config} \neq \text{config}_i[i])$, where both p_i and p_j are active processors. In Claims 5.3.3, 5.3.4 and 5.3.5 we show that in all of these cases, within $O(1)$ asynchronous rounds, $\forall p_i \in \mathcal{I} : \text{config}_i[i] \in \{\perp, \text{FD}_i[i]\}$ holds before showing in Claim 5.3.6 that, within $O(1)$ asynchronous rounds, there are no configuration conflicts.

Claims 5.3.3, 5.3.4 and 5.3.5 consider the values in the field config that are either held by an active processor $p_i \in \mathcal{I}$ or in its outgoing communication channel to another active processor $p_j \in \mathcal{I}$. We define the set $S = \{S_i \cup S\_out_i\}_{p_i \in \mathcal{I}}$ to be the set of all these values, where $S_i = \{\text{config}_i[j]\}_{p_j \in \text{FD}_i[i]}$ and $S\_out_i = \{m_{i,j}.\text{config}\}_{p_j \in \text{FD}_i[i]}$.

Claim 5.3.3. *Suppose that in R's starting system state, there are processors $p_i, p_j \in \mathcal{I}$ that are active in R, for which $|S \setminus \{\perp, \sharp\}| > 1$, where $\exists S' \subseteq S : S' \in \{\{\text{config}_i[i], \text{config}_i[j]\}, \{\text{config}_i[i], m_{i,j}.\text{config}\}\}$. Within $O(1)$ asynchronous rounds, the system reaches a state in which $\text{config}_i[i] \in \{\perp, \text{FD}_i[i]\}$ holds.*

Proof. Suppose that $S' = \{\text{config}_i[i], \text{config}_i[j]\}$ holds. Immediately after R's starting state, processor p_i has an applicable step that includes the execution of the do forever loop (line 1 to 14). In that step, which occurs within $O(1)$ asynchronous rounds, the if-statement condition ($|\{\text{config}_i[k] : p_k \in \text{FD}_i[i]\} \setminus \{\perp, \sharp\}| > 1$) (line 8's if-statement) holds, p_i assigns $\perp$ to $\text{config}_i[i]$ and the proof is done. Suppose that $S' = \{\text{config}_i[i], m_{i,j}.\text{config}\}$ holds. Upon $m_{i,j}$'s arrival, which occurs within $O(1)$ asynchronous rounds, processor p_i assigns $m_{i,j}.\text{config}$ to $\text{config}_i[j]$ (line 15) and the case of $S' = \{\text{config}_i[i], \text{config}_i[j]\}$ holds. □

Claim 5.3.4. *Suppose that in R's starting system state, there are processors $p_i, p_j \in \mathcal{I}$ that are active in R, for which $|S \setminus \{\perp, \sharp\}| > 1$, where $\exists S' \subseteq S : S' \in \{\{\text{config}_i[i], \text{config}_j[j]\}\}$. Within $O(1)$ asynchronous rounds, the system reaches a state in which $\text{config}_i[i] \in \{\perp, \text{FD}_i[i]\}$ or $\text{config}_j[j] \in \{\perp, \text{FD}_i[i]\}$ holds.*

Proof. Suppose, towards a contradiction, for any system state $c \in R$ that neither $\text{config}_i[i] \in \{\perp, \text{FD}_i[i]\}$ nor $\text{config}_j[j] \in \{\perp, \text{FD}_i[i]\}$. Note that p_i and p_j exchange messages within $O(1)$ asynchronous rounds, because whenever processor p_i repeatedly sends the same message to processor p_j, it holds that p_j receives that message within $O(1)$ asynchronous rounds and vice versa. Such a message exchange implies that the case of $|S \setminus \{\perp, \sharp\}| > 1$ (Claim 5.3.3) holds within $O(1)$ asynchronous rounds, where $\exists S' \subseteq S : S' \in \{\{\text{config}_i[i], m_{i,j}.\text{config}\}, \{\text{config}_j[j], m_{i,j}.\text{config}\}\}$. Thus, we reach a contradiction and therefore, within $O(1)$ asynchronous rounds, the system reaches a state in which $\text{config}_i[i] \in \{\perp, \text{FD}_i[i]\}$ or $\text{config}_j[j] \in \{\perp, \text{FD}_i[i]\}$ hold. □

Claim 5.3.5. *Suppose that in R's starting system state, there is a processor $p_i \in \mathcal{I}$ that is active in R, for which $\text{config}_i[i] \in \{\perp, \text{FD}_i[i]\}$. Within $O(1)$ asynchronous rounds, (1) for any system state $c \in R$, it holds that $\text{config}_i[i] \in \{\perp, \text{FD}_i[i]\}$. Moreover, (2) $R = R' \circ R''$ has a suffix, R'', for which $\forall c'' \in R'' : \forall p_i, p_j : (\{m_{i,j}.\text{config}, \text{config}_j[i], \text{config}_j[j]\} \setminus \{\perp, \text{FD}_i[i]\}) = \emptyset$.*

Proof. We prove each part of the statement separately.

Part (1). We start the proof by noting that $\forall p_i, p_j \in \mathcal{I}$, it holds that, throughout R, we have that $\text{FD}_i[i]$'s value does not change and that $\text{FD}_i[i] = \text{FD}_j[j]$, because this lemma assumes that R is admissible. To show that $\text{config}_i[i] \in \{\perp, \text{FD}_i[i]\}$ holds in any $c \in R$, we argue that any step $a_i \in R$ in which p_i changes $\text{config}_i[i]$'s value includes the execution of line 8 or line 9 (see the remark at the beginning of this lemma's proof about $a_i \in R$ not including the execution of lines 5 to 12), which assign to $\text{config}_i[i]$ the values $\perp$, and respectively, $\text{FD}_i[i]$.

Part (2). In this part of the proof, we first consider the values in $m_{i,j}.\text{config}$ and $\text{config}_j[i]$ before considering the one in $\text{config}_j[j]$.

Part (2.1). To show that in $c'' \in R''$ it holds that $\forall p_i, p_j : \{m_{i,j}.\text{config}, \text{config}_j[i]\} \setminus \{\perp, \text{FD}_i[i]\} = \emptyset$, we note that when p_i loads a message $m_{i,j}$ (line 14) before sending to processor p_j, it uses $\text{config}_i[i]$'s value for the field config. Thus, within $O(1)$ asynchronous rounds, $m_{i,j}.\text{config} \in \{\perp, \text{FD}_i[i]\}$ and therefore $\text{config}_j[i] \in \{\perp, \text{FD}_i[i]\}$

records correctly in $c'' \in R''$ the most recent $m_{i,j}$'s value that p_j receives from p_i (line 15).

Part (2.2). To show that, within $O(1)$ asynchronous rounds, $\text{config}_j[j] \in \{\perp, \text{FD}_i[i]\}$, we note that once p_j changes the value of $\text{config}_j[j]$, it holds that $\text{config}_j[j] \in \{\perp, \text{FD}_i[i]\}$ thereafter (due to the remark in the beginning of this lemma, which implies that only lines 8 and 9 can change $\text{config}_j[j]$, and by the part (1) of this claim's proof while replacing the index i with j). Suppose, towards a contradiction, that p_j does not change that value of $\text{config}_j[j]$ throughout R and yet $\text{config}_j[j] \notin \{\perp, \text{FD}_i[i]\}$. Note that $(|\{\text{FD}[j] : p_j \in \text{FD}[i]\}| = 1)$ (see the second clause of the if-statement condition in line 9) holds throughout R, because R is admissible. Therefore, whenever p_i takes a step that includes the execution of the do forever loop (line 1 to 14), processor p_i assigns $\text{FD}_i[i]$ to $\text{config}_i[i]$ (line 9) and sends to p_j the message $m_{i,j}$, such that $m_{i,j}.\text{config} = \text{config}_i[i]$ (because it executes line 14) and $\text{config}_i[i] = \text{FD}_i[i]$ (see part (2.1) of this proof). Since p_i sends $m_{i,j}$ repeatedly, processor p_j receives $m_{i,j}$ within $O(1)$ asynchronous rounds and stores in $\text{config}_j[i] = m_{i,j}.\text{config} = \text{config}_i[i] = \text{FD}_i[i] \neq \perp$. Immediately after that step, the system state allows p_j to take a step in which the condition $(|\{\text{config}_j[k] : p_j \in \text{FD}_j[k]\} \setminus \{\perp, \sharp\}| > 1)$ (line 8's if-statement) holds and p_j changes $\text{config}_j[j]$'s value to $\perp$. Thus, this part of the proof ends with a contradiction, which implies that the system reaches a state in which $\text{config}_j[j] \in \{\perp, \text{FD}_i[i]\}$. □

Claim 5.3.6. *Suppose that in R's starting system state, it holds that for every two processors $p_i, p_j \in \mathcal{I}$ that are active in R, we have that* $(\{\text{config}_i[i], \text{config}_j[i], m_{i,j}.\text{config}\} \setminus \{\perp, \text{FD}_i[i]\}) = \emptyset$, *where $m_{i,j}$ is a message in the channel from p_i to p_j. Within $O(1)$ asynchronous rounds, the system reaches a state in which* $\text{config}_i[i] = \text{FD}_i[i]$.

Proof. By this claim assumptions, we have that in R's starting system state, the if-statement condition $(|\{\text{config}_i[k] : p_k \in \text{FD}_i[k]\} \setminus \{\perp, \sharp\}| > 1)$ (line 8) does not hold. Moreover, $|\{\text{FD}_i[j] : p_j \in \text{FD}_i[i]\}| = 1$ (line 9) holds throughout R, because R is admissible. Therefore, this claim's assumptions with respect to R's starting states actually hold for any system state $c \in R$, because only lines 8 and 9 can change the value of $\text{config}_i[i] \in \{\perp, \text{FD}_i[i]\}$, which later p_i uses for sending the message $m_{i,j}$ (line 14), and thus also $m_{i,j}.\text{config} \in \{\perp, \text{FD}_i[i]\}$ as well as $\text{config}_j[i] \in \{\perp, \text{FD}_i[i]\}$ records correctly the most recent $m_{i,j}$'s that p_j receives from p_i (line 15).

To conclude this proof, we note that immediately after any system state $c \in R$, processor p_i has an applicable step $a_i \in R$ that includes the execution of line 9 (by

similar arguments to the ones used by the first part of this proof). Moreover, a_i does not include the execution of line 8, because, by the first part of this proof, the condition of the if-statement of line 8 does not hold. In the system state that immediately follows a_i, the invariant $\text{config}_i[i] = \text{FD}_i[i]$ holds. Note that a_i is taken within $O(1)$ asynchronous rounds. □

By this lemma's assumption, there is no configuration $c \in R$ replacement state nor replacement message. Claim 5.3.6 implies that the system reaches a state $c_{noConf} \in R$ that has no configuration conflict within $O(1)$ asynchronous rounds. Thus, c_{noConf} is safe. This completes the proof of Lemma 5.3.2. □

Lemma 5.3.7 (No type-4 stale information)**.** *Let R be an admissible execution of Algorithm 5. Within $O(1)$ asynchronous rounds, the system reaches a state $c \in R$ in which the invariant of no type-4 stale information holds thereafter.*

Proof. Without the loss of generality, suppose that there is no system state in R that encodes a configuration conflict. (We can make this assumption without losing generality because Lemma 5.3.2 implies that this claim is true whenever this assumption is false.) Moreover, let $c \in R$ be a system state in which processor p_i has, within $O(1)$ asynchronous rounds, an applicable step a_i that includes the execution of the do forever loop (line 1 to 14).

Since R is admissible, $(\{(\text{FD}_i[i], \text{FD}_i[i].part)\} = \{(\text{FD}_i[k], \text{FD}_i[k].part)\}_{p_k \in \text{FD}_i[i].part})$ holds in c. Therefore, the case in which $((\text{config}_i[i] \cap \text{FD}_i[i].part) = \emptyset)$ in c implies a call to the function $configSet(\perp)$ (line 6) in the step that immediately follows. By using Lemma 5.3.2, we have that this lemma is true within $O(1)$ asynchronous rounds. □

Phase and degree progressions

Let R be an execution of Algorithm 5 that is admissible with respect to the participant sets. Suppose that p_i is a processor that is active in R, and that $p_i \in \text{FD}[i].part$. We say that processor p_i is an *active participant* in R.

Notation. Let $p_i, p_j, p_k \in \mathcal{I}$ be processors that are active participants in R and $c \in R$ a system state. We denote by:

- $\text{NA}(c) = \{(\text{prp}_j[k], \text{myAll}_j(k))\}_{p_j, p_k \in \text{FD}_i[i].part} \cup \{(\text{prp}_j[k], \text{myAll}_j(k)) : m_{j,k} = \langle \bullet, \text{prp}_j[k], \text{myAll}_j(k), (\bullet) \rangle \in channel_{k,j}\}_{p_j, p_k \in \text{FD}_i[i].part} \cup \{(\text{echo}_j[k].\text{prp}, \text{echo}_j[k].\text{all}) : m_{j,k} = \langle \bullet, (\bullet, \text{echo}_j[k].\text{prp}, \text{echo}_j[k].\text{all}) \rangle \in channel_{k,j}\}_{p_j, p_k \in \text{FD}_i[i].part} \setminus \{(\langle 0, \perp \rangle, \bullet)\}$ the

set of all pairs that includes the notification (of a configuration replacement proposal) and all fields that appear in c (after excluding the default notification, $\langle 0, \perp \rangle$, while including all the information in the processors' states and communication channels as well as their replications, e.g., the echo field).

- $N(c) = \{\mathrm{n} : (\mathrm{n}, \mathrm{a}) \in \mathrm{NA}(c)\}$ the set of notifications that appear in c.
- $D(c, \mathrm{n}) = \{2 \cdot \mathrm{n}.\mathit{phase} + |\{1 : \mathrm{a}\}| : (\mathrm{n}, \mathrm{a}) \in \mathrm{NA}(c)\}$, the *degree set* of notification $\mathrm{n} \in N(c)$.
- $S(c, \mathrm{n}) = \{\mathrm{n}' : ((\mathrm{n}' \in N(c)) \wedge (\mathrm{n}.\mathit{set} = \mathrm{n}'.\mathit{set}))\}$ the set of all notifications $\mathrm{n}' \in N(c)$ that have the same *set* field as the one of a given notification $\mathrm{n} \in N(c)$ that appears in a given system state $c \in R$, and $S(c) = \{\mathrm{n}.\mathit{set} : \mathrm{n} \in N(c)\}$ is the set of all notification sets in c.

We proceed to define the notion a *proposal conformity*, which intuitively suggests that the phase of p_i conforms to p_j if their *prp* are the same, and p_k's phase precedes that of p_j by one modulo 3. At the same time, p_k remains in the $\mathit{allSeen}_j$ set which allows p_j to catch up with p_k's phase.

Definition 5.3.2 (Proposal conformity)**.** *We say that* $\mathrm{prp}_j[j]$ *conforms to* $\mathrm{prp}_k[k]$*, and use* $\mathrm{prp}_j[j] \rightsquigarrow_{prp} \mathrm{prp}_k[k]$ *to denote this, when* $(\mathrm{prp}_j[j].\mathit{set} = \mathrm{prp}_k[k].\mathit{set}) \wedge (((\mathrm{prp}_j[j].\mathit{phase} = \mathrm{prp}_k[k].\mathit{phase}) \vee (\mathrm{prp}_k[k].\mathit{phase} = (\mathrm{prp}_j[j].\mathit{phase} + 1) (\mathrm{mod}\ 3))) \implies (p_k \in \mathrm{allSeen}_j))$.

Lemma 5.3.8. *Let R be an execution of Algorithm 5 that is admissible with respect to the participant sets and that does not include an explicit (delicate) replacement. Suppose that in R's starting system state, c, there are notifications (of configuration replacement proposals), i.e., $N(c) \neq \emptyset$. Moreover, suppose that for any active participant $p_i \in \mathcal{I}$ and any $c^* \in R$ it holds that $(\mathrm{prp}_i[i], \mathrm{myAll}_i(i)) = (\mathrm{n}_i^*, \mathrm{a}_i^*) \in \mathrm{NA}(c^*)$. Within $O(1)$ asynchronous rounds, the system reaches a state $c^{\circledast} \in R$ in which one of the following is true:*
(i) $(\mathrm{n}_i^*, \mathrm{a}_i^*) \notin \mathrm{NA}(c^{\circledast})$,
(ii) the system takes a step (immediately after $c^{\circledast}$) in which there is a call to the function $configSet(\perp)$ (line 6), or
(iii) the following invariants (1) to (7) hold.

(1) *Within $O(1)$ asynchronous rounds, the system reaches a state $c' \in R$ in which for any $p_j \in \mathrm{FD}_i.\mathit{part}$ that is an active participant in R, it holds that $(\mathrm{prp}_j[i], \mathrm{all}_j[i]) = (\mathrm{n}_i^*, \mathrm{a}_i^*)$ and $\mathrm{FD}_j[i].\mathit{part} = \mathrm{FD}_i.\mathit{part}$ and $\mathrm{FD}_j[j].\mathit{part} = \mathrm{FD}_i.\mathit{part}$. Moreover, $\mathrm{prp}_i[i] \rightsquigarrow_{prp} \mathrm{prp}_j[j]$ or $\mathrm{prp}_j[j] \rightsquigarrow_{prp} \mathrm{prp}_i[i]$ in c'.*

(2) *Suppose that invariant (1) holds in every system state of R. Within $O(1)$ asynchronous rounds, the system reaches a state $c' \in R$ in which it holds that* $\text{echo}_i[j].\text{prp} = \text{n}_i^*$, $\text{echo}_i[j].\text{part} = \text{FD}_i[i].\textit{part}$ *in c'.*

(3) *Suppose that invariants (1) and (2) hold in every system state of R. Within $O(1)$ asynchronous rounds, the system reaches a state $c' \in R$, such that* $\text{myAll}_i(i) = \textit{true}$ *holds in c'.*

(4) *Suppose that invariants (1), (2) and (3) hold in every system state of R. Within $O(1)$ asynchronous rounds, the system reaches a state $c' \in R$, in which it holds that* $\text{all}_j[i] = \textit{true}$ *in c', where $p_j \in \text{FD}_i[i].\textit{part}$.*

(5) *Suppose that invariants (1) to (4) hold in every system state of R. Within $O(1)$ asynchronous rounds, the system reaches a state $c' \in R$, in which it holds that* $\text{echo}_i[j] = (\text{FD}_i[i].\textit{part}, \text{prp}_i[i], \text{myAll}_i(i))$ *in c', where $p_j \in \text{FD}_i[i].\textit{part}$.*

(6) *Suppose that invariants (1) to (5) hold in every system state of R. Within $O(1)$ asynchronous rounds, the system reaches a state $c' \in R$, in which it holds that $p_i \in \textit{allSeen}_j$ in c', where $p_j \in \text{FD}_i[i].\textit{part}$.*

(7) *Suppose that invariants (1) to (6) hold in every system state $c' \in R$. Within $O(1)$ asynchronous rounds, the system reaches a state $c'' \in R$, such that there exists an active participant $p_k \in P$ for which the if-statement condition of line 11 holds in c''. Specifically, $\forall p_k \in \text{FD}_j[j].\textit{part} : \textit{degree}_j(k) - \textit{degree}_j(j) \pmod 6 \in \{-1, 0, 1\}$ holds in c' and either* **(7.1)** *$\forall p_k \in \text{FD}_j[j].\textit{part} : \textit{degree}_j(k) - \textit{degree}_j(j) \pmod 6 \in \{0\}$ and this part holds for any such p_k, or* **(7.2)** *the if-statement condition of line 11 holds in c'' for any $p_k \in \text{FD}_j[j].\textit{part} : \textit{degree}_j(k) - \textit{degree}_j(j) \pmod 6 \in \{-1\}$ but not for $p'_k \in \text{FD}_j[j].\textit{part} : \textit{degree}_j(k) - \textit{degree}_j(j) \pmod 6 \in \{0, 1\}$.*

Proof. Suppose that cases (i) and (ii) do not occur during R. We prove invariants (1) to (7) hold within $O(1)$ asynchronous rounds (or show a contradiction with the lemma assumptions, such that, at some system state during R, the value of $(\text{prp}_i[i], \text{myAll}_i(i))$ is not $(\text{n}_i^*, \text{a}_i^*)$).

(1) Since p_i repeatedly sends message $m_{i,j}$ to every active processor $p_j \in \text{FD}_i[j].\textit{part}$ (line 14), where $m_{i,j} = \langle \bullet, \text{n}_i^*, \text{a}_i^* \rangle$, message $m_{i,j}$ arrives within $O(1)$ asynchronous rounds to p_j (line 15 and the definition of fair execution, Section 5.3.2). This causes p_j to store $(\text{n}_i^*, \text{a}_i^*)$ in $(\text{prp}_j[i], \text{all}_j[i])$ (because of our assumption that $\text{prp}_i[i] = \text{n}_i^*$

and $\text{myAll}_i(i) = \text{a}_i^*$ hold in every $c^* \in R$). By this lemma assumption that R is admissible with respect to the participant sets and similar arguments to the above, we get that $\text{FD}_j[i].part = \text{FD}_i.part$ in c''. The same assumption also implies that $\text{FD}_j[j].part = \text{FD}_i.part$ in c''.

Note that the proof is done when assuming that, throughout $O(1)$ asynchronous rounds, $\text{prp}_i[i] \rightsquigarrow_{prp} \text{prp}_j[j]$ and therefore we assume, towards a proof by contradiction, that $\text{prp}_j[j] = \text{n}_j^* \not\rightsquigarrow_{prp} \text{n}_i^* = \text{prp}_i[i]$ in c'. Suppose, without the loss of generality, that n_j^* is lexicographically greater than n_i^* (generality is not lost due to the symmetry between p_i and p_j with respect to $\text{prp}_j[j] = \text{prp}_i[j] = \text{n}_j^*$ and $\text{prp}_i[i] = \text{prp}_j[i] = \text{n}_i^*$). For each of the following four cases, we show a contradiction, and therefore, $\text{prp}_i[i] \rightsquigarrow_{prp} \text{prp}_j[j]$ in c'.

- Let $A = \{degree_i(k) - degree_i(i)\}_{p_k \in \text{FD}_i[i].part}$. Suppose that neither $A \subseteq \{-1, 0\}$ nor $A \subseteq \{0, 1\}$ in c'. Line 6 and the definition of *corrDeg*() (Fig. 5.3, line 10) imply that, within $O(1)$ asynchronous rounds from c', the system takes a step that includes a call to $configSet(\perp)$. This is a contradiction with this lemma assumption (that there is no call to $configSet(\perp)$), see assumption (ii). Moreover, $(\{p_k \in \text{FD}_i[i].part : \text{prp}_i[i].phase + 1 \bmod 3 = \text{prp}_i[k].phase\} \not\subseteq \text{allSeen}_i)$ must hold in c'. The reason is that this lemma assumes that there is no call to $configSet(\perp)$ and therefore the if-statement condition of line 6 must not hold.

- The case of $(degree_i(j) - degree_i(i)) = 0$ in c' leads to a contradiction. The proof of this statement follows:

  - Suppose that $(\text{prp}_i[i].set = \text{prp}_i[j].set)$. This implies that $\text{prp}_i[i] \rightsquigarrow_{prp} \text{prp}_i[j]$ in c' and then $\text{prp}_j[j] \rightsquigarrow_{prp} \text{prp}_i[i]$ in c' since $\text{prp}_j[j] = \text{prp}_i[j]$.
  - Suppose that $(\text{prp}_i[i].set \neq \text{prp}_i[j].set)$. By the assumption that $\text{prp}_i[j] = \text{n}_j^*$ is lexicographically greater than $\text{prp}_i[i] = \text{n}_i^*$ and line 3, we have that p_i assigns $\text{prp}_i[j].set$ to $\text{prp}_i[i].set$ in its first step after c' that includes the execution of the do-forever loop (lines 1 to 14). This violates our assumption that p_i does not change the value of $\text{prp}_i[i]$. Thus, a contradiction.

- Assuming that $(degree_i(j) - degree_i(i)) = 1$ in c' leads to a contradiction due to the same arguments as the latter instance of the previous case.

- Suppose that $(degree_i(j) - degree_i(i)) = (-1)$ in c'. Earlier in this proof, we showed that p_i stores $(\text{n}_j^*, \text{a}_j^*)$ in $(\text{prp}_i[j], \text{all}_i[j])$ and p_j stores $(\text{n}_i^*, \text{a}_i^*)$ in $(\text{prp}_j[i], \text{all}_j[i])$.

Therefore, $(degree_j(i) - degree_j(j)) = 1$ in c'. Hence, the proof follows from the previous item.

(2) By similar arguments to the ones that show the arrival of $m_{i,j}$ in part (1) of this proof, processor $p_j \in \text{FD}_i[i].part$ repeatedly sends the message $m_{j,i} = \langle \bullet, \text{prp} = \text{n}_j^*, \bullet, \text{echo} = (\bullet, \text{n}_i^*, \bullet) \rangle$ to p_i, which indefinitely stores n_j^* in $\text{prp}_i[j]$ and n_i^* in $\text{echo}_i[j]$ (line 15) within $O(1)$ asynchronous rounds. Using arguments that are similar to the above, we have that $\text{echo}_i[j].\text{part} = \text{FD}_i[i].part$ in c'.

(3) We show that, within $O(1)$ asynchronous rounds, p_i takes a step that includes the execution of the do forever loop (lines 1 to 14), such that immediately after that step, the system reaches a state in which $\text{myAll}_i(i) = true$ holds (due to line 3) in c'. We prove that, within $O(1)$ asynchronous rounds, the system reaches a state in which $\forall p_k \in \text{FD}_i[i].part : echoNoAll_i(k)$. Specifically, we show that $(\text{FD}_i[i].part, \text{prp}_i[i]) = (\text{echo}_i[k].part, \text{echo}_i[k].\text{prp})$. Note that by this part assumption, we know that invariant (2) holds and therefore $\text{FD}_i[i].part = \text{echo}_i[k].part$ in every system state in R. Moreover, $\text{echo}_i[j].\text{prp} = \text{n}_i^*$ and $\text{prp}_i[j] = \text{n}_j^*$ in every system state in R.

Note that the proof is done if $\text{myAll}_i(i) = true$ or $\text{n}_i^* = \text{n}_j^*$ in c'. Suppose, towards a proof by contradiction, that neither $\text{myAll}_i(i) = true$ nor $\text{n}_i^* = \text{n}_j^*$ in c'. By part (1) of this proof, we know that $\text{n}_i^* \rightsquigarrow_{prp} \text{n}_j^*$ in c', which implies that $\text{n}_i^*.set = \text{n}_j^*.set$ (Definition 5.3.2). Thus, it can only be that $\text{n}_i^*.phase \neq \text{n}_j^*.phase$. However, this means that $(\exists_{p_k \in \text{FD}_i[i].part} \neg corrDeg_i(i,k))$ in c', which contradicts this lemma assumption that p_i does not call $configSet(\perp)$ (line 6).

(4) By similar arguments that appear in parts (1) and (2) of this proof, processor p_i sends repeatedly the message $m_{i,j} = \langle \bullet, \text{all} = true, \bullet \rangle$ to processor p_j. The message $m_{i,j}$ arrives, within $O(1)$ asynchronous rounds, to p_j (line 15 and the assumption that a message sent infinitely often is received infinitely often, Chapter 3). Once that happens, processor p_j stores $true$ in $\text{all}_j[i]$ (as well as the notification n in $\text{prp}_j[i]$). This holds for every system state c' that follows $m_{i,j}$'s arrival to p_j.

(5) By similar arguments that appear in parts (1) and (2) of this proof, processor p_j sends repeatedly the message $m_{j,i} = \langle \bullet, \text{prp} = \text{n}_j^*, \text{all} = true, \text{echo} = (\text{FD}_i[i].part, \text{n}_i^*, true) \rangle$ to processor p_i. The message $m_{j,i}$ arrives, within $O(1)$ asynchronous rounds, to p_i (line 15 and the assumption that a message sent infinitely often is received infinitely often, Section 3). Once $m_{j,i}$ arrives, p_i stores $(\text{n}_i^*, true)$ in

$\text{echo}_i[j]$. This holds for every system state c' that follows $m_{j,i}$'s arrival to p_i.

(6) We show that, once the invariants of parts (1) to (5) of this proof hold, the for-each condition of line 5 holds as well. The for-each condition of line 5 requires that for any $p_j \in \text{FD}_i[i].part$ to have all[k], where the index k is substituted here with j. By part (4) of this lemma, we have that all[k] holds.

(7) We need to show that, for $p_j \in \text{FD}_j[j].part$ (or another $p_k \in \text{FD}_j[j].part$), it holds that $(allSeen_j() \wedge \bigwedge_{p_k \in \text{FD}_j[j].part} echo(k))$ (line 11), where $(\{(\text{FD}_j[j].part, \text{prp}_j[j], \text{myAll}_j(j))\} = \{\text{echo}_j[k]\} \wedge degree_j(k) - degree_j(j) (\text{mod } 6) \in \{0, 1\})$ (line 12, Fig. 5.3).

We note that part (6) of this proof implies $allSeen_j()$ and part (5) implies $\{(\text{FD}_j[j].part, \text{prp}_j[j], \text{myAll}_j(j))\} = \{\text{echo}_j[k]\}$. By the assumption of this proof that no reset occurs during R, we have that $\forall p_k \in \text{FD}_j[j].part : degree_j(k) - degree_j(j) (\text{mod } 6) \in \{-1, 0, 1\}$ (because otherwise the if-statement condition in line 6 would hold and a reset would occur). We note that the proof is done in the case of $\forall p_k \in \text{FD}_j[j].part : degree_j(k) - degree_j(j) (\text{mod } 6) \in \{0, 1\}$. Suppose that $\exists p_k \in \text{FD}_j[j].part : degree_j(k) - degree_j(j) (\text{mod } 6) \in \{-1\}$ in system state c'. By part (1) of this proof, it holds that $degree_j(k) - degree_j(j) (\text{mod } 6) \in \{1\}$ in c' (note that j and k are swapped) and therefore the if-statement condition of line 11 holds for p_k. Thus, the system has reached the state c''. □

We proceed to prove that the phase of the configuration mechanisms stabilizes within a bounded period.

Theorem 5.3.9. *Let R be an admissible execution w.r.t. participants of Algorithm 5 (which may include explicit delicate or spontaneous replacements). Within $O(N)$ asynchronous rounds, R has a suffix that does not include a system state and $p_i, p_j \in \mathcal{I}$ for which Equation 5.1 holds.*

$$\begin{aligned} &(\exists p_k \in \text{FD}[i].part : \neg corrDeg(i,k)) \vee \\ &(\exists p_k \in \text{FD}_i[i].part : (\text{prp}_j[k].phase = (\text{prp}_j[j].phase + 1) (\text{mod } 3)) \\ &\wedge (p_k \notin allSeen_j)) \end{aligned} \tag{5.1}$$

Proof. Let R' be a suffix of R that the system reaches within $O(1)$ asynchronous rounds during which stale information of type-1,-2 and-4 is removed from the system, cf. Lemmas 5.3.1, 5.3.2 and 5.3.7 as well as the statement of Claim 5.3.11. We use lemmas 5.3.10, 5.3.12 and 5.3.13 to show that, within $O(N)$ asynchronous rounds, R' has a suffix that does not include a system state in which Equation 5.1 holds.

Lemma 5.3.10. *Suppose that during R' there is no processor $p_j \in \mathrm{FD}_i[i].part$ that changes $(\mathrm{prp}_j[j], \mathrm{all}_j[j])$ more than thirteen times. In this case, within $O(N)$ asynchronous rounds during R, either:*
(i) the system reaches a state $c^{\circledast} \in R$ in which $\mathrm{NA}(c^{\circledast}) = \emptyset$,
(ii) the system takes a step in which there is a call to the function $configSet(\perp)$ (line 6), or
(iii) the invariants (1) to (7) hold.

Proof. The lemma proof is treated incrementally in three steps.
Step 1 – *Suppose that during the first $O(1)$ asynchronous rounds of R', no processor $p_j \in \mathrm{FD}_i[i].part$ changes $(\mathrm{prp}_j[j], \mathrm{all}_j[j])$. Conditions (i) to (iii) hold within these $O(1)$ asynchronous rounds.*

Proof. The proof of this step is implied by Lemma 5.3.8.

Step 2 – *Suppose that during R' there is at most one processor $p_j \in \mathrm{FD}_i[i].part$ that changes $(\mathrm{prp}_j[j], \mathrm{all}_j[j])$ at most once (while all the other processors $p_k \in \mathrm{FD}_i[i].part$ do not change $(\mathrm{prp}_k[j], \mathrm{all}_k[j])$). Conditions (i) to (iii) hold within $O(1)$ asynchronous rounds.*

Proof. Suppose that, within $O(1)$ asynchronous rounds, processor p_j takes a step $a_j \in R'$ that changes $(\mathrm{prp}_j[j], \mathrm{all}_j[j])$. By Step 1, this step holds within $O(1)$ asynchronous rounds from a_k.

Suppose, towards a contradiction, that processor p_k takes a step $a_k \in R'$ that changes $(\mathrm{prp}_k[j], \mathrm{all}_k[j])$ only after more than $O(1)$ asynchronous rounds from the starting system state of R'. By Step 1, within $O(1)$ asynchronous rounds from the starting state of R', one of the conditions (i) to (iii) holds. For the cases in which conditions (i) or (ii) hold, the proof is done. Suppose that invariants (1) to (7) (Lemma 5.3.8) hold. By line 11, a_k occurs within $O(1)$ asynchronous rounds from the starting system state of R'. However, we assumed (towards a contradiction) that a_k is the only step in R' that changes $(\mathrm{prp}_k[j], \mathrm{all}_k[j])$ and it does not occur within $O(1)$ asynchronous rounds from the starting system state of R'. Thus, a contradiction and the claim is true.

Step 3 – *Suppose that during R' all processors $p_j \in \mathrm{FD}_i[i].part$ change $(\mathrm{prp}_j[j], \mathrm{all}_j[j])$ at most x times. Conditions (i) to (iii) hold within $O(xN)$ asynchronous rounds.*

Proof. We first consider the case of $x = 1$ and then the case of $x > 1$.
The case of x = 1. Let $p_{k_1} \in \mathrm{FD}_i[i].part$ be the first processor that changes the value

of (prp, all) (if such change exists in R'), say, in step $a_{k_1} \in R_1$, where $R_1 = R'$. The step a_{k_1} occurs within $O(1)$ asynchronous rounds in R_1 (by similar arguments to the ones for Step 2). Let us consider the suffix R_2 of R_1 that starts immediately after a_{k_1}. Let $p_{k_2} \in \text{FD}_i[i].part$ be the second processor that changes the value of (prp, all) (if such change exists in R_2), say, in step $a_{k_2} \in R_2$. By the same arguments as above, a_{k_2} occurs within $O(1)$ asynchronous rounds from the start of R_2. In the same manner, we can construct R_ℓ to include at most one step in which processor p_ℓ changes $(\text{prp}_\ell[\ell], \text{all}_\ell[\ell])$. The proof of case $x = 1$ holds when considering R_ℓ for any $\ell \leq N$ and observing that a_ℓ occurs within $O(N)$ asynchronous rounds from the start of R'.

The case of x > 1. Let $a_{k_1}, a_{k_2}, \ldots, a_{k_\ell}$, where $\ell \leq xN$, be the sequence of steps in which any processor in $\text{FD}_i[i].part$ changes the value of (prp, all). Let us write $R' = R_1 \circ \ldots \circ R_\ell$ such that each sub-execution, say, R_ℓ, includes at most one step from the sequence $a_{k_1}, a_{k_2}, \ldots, a_{k_\ell}$, which is a_{k_ℓ} for the case of R_ℓ. By similar arguments to this case of $x = 1$, for any $\ell' < \ell$, it holds that R'_ℓ includes at most $O(1)$ asynchronous rounds. Moreover, a_{k_ℓ} occurs within $O(1)$ asynchronous rounds in R_ℓ. This proves that conditions (i) to (iii) hold within $O(xN)$ asynchronous rounds.

This completes the proof of Lemma 5.3.10. □

Claim 5.3.11 details the way in which the order of updates to the pair $(\text{prp}, myAll(\bullet))$ appear is the system.

Claim 5.3.11. *Let $p_j, p_k \in \text{FD}_i[i].part$ and $(\text{prp}^{j,k,0}, \text{all}^{j,k,0})$ be the value of $(\text{prp}_j[k], \text{all}_j[k])$ in R's starting system state. Moreover, let $\text{pa}^{j,k,0} = (\text{prp}^{j,k,0}, \text{all}^{j,k,0}), (\text{prp}^{j,k,1}, \text{all}^{j,k,1}), \ldots$ be the sequence of all values that p_j stores in $(\text{prp}_j[k], \text{all}_j[k])$ during R.*

(1) *Within $O(1)$ asynchronous rounds, there exists $y \in \mathbb{Z}^+$, such that the sequence $\text{pa}^{j,k,y} = (\text{prp}^{j,k,y}, \text{all}^{j,k,y}), (\text{prp}^{j,k,y+1}, \text{all}^{j,k,y+1}), \ldots$ is a sub-sequence of the $\text{pa}^{k,k,0} = (\text{prp}^{k,k,0}, \text{all}^{k,k,0}), (\text{prp}^{k,k,1}, \text{all}^{k,k,1}), \ldots$ sequence. That is, $\text{pa}^{j,k,y}$ of R' includes a subset of element in $\text{pa}^{k,k,0}$ while keeping their order of appearance in R'. Moreover,*

(2) *suppose that message m' is in transit from p_k to p_j in system state $c \in R'$ and that in c the value of $(\text{prp}_k[k], \text{all}_k[k])$ is $(\text{prp}^{j,k,x}, \text{all}^{j,k,x})$. Then, $m' = \langle \text{prp}^{j,k,x'}, \text{all}^{j,k,x'} \rangle$, where $x' \in \{0, \ldots, x\}$. Furthermore,*

(3) *suppose that message $m'' = \langle \text{prp}^{j,k,x''}, \text{all}^{j,k,x''} \rangle$ is in transit from p_k to p_j after m' while $(\text{prp}_k[k], \text{all}_k[k]) = (\text{prp}^{j,k,x'''}, \text{all}^{j,k,x'''})$. Then $x'' \in \{x', \ldots, x'''\}$.*

Proof. The proof of parts **(1)** to **(3)** is by the communication fairness assumption, the

correctness of the self-stabilizing end-to-end data link algorithm (Chapter 3), which uses a token circulation mechanism for providing reliable FIFO communications of messages sent in line 14 and received in line 15. □

Lemma 5.3.12. *Suppose that there is a processor* $p_j \in \text{FD}_i[i].part$ *that changes* $(\text{prp}_j[j], \text{all}_j[j])$ *more than thirteen times. In this case, within* $O(N)$ *asynchronous rounds during* R'*, the system reaches to a state* $c^* \in R'$ *in which Equation 5.2 holds.*

$$\forall_{p_k,p_\ell \in \text{FD}_j[j].part} : ((\text{prp}_k[\ell] = \text{prp}_j[j]) \wedge ((\text{prp}_j[k], \text{all}_j[k]) = (\text{prp}_j[j], \text{all}_j[j] = true))) \wedge \quad (5.2)$$
$$(\exists_{m \in channel_{j,k} \cup channel_{k,j} : j \in \{k,\ell\}} \implies m = (\bullet, \text{prp}_j[j], \text{all}_j[j] = true, (\bullet, \text{prp}_j[j], \text{all}_j[j] = true))) \wedge$$
$$(\exists_{m \in channel_{k,\ell} : j \notin \{k,\ell\}} \implies m = (\bullet, \text{prp}_j[j], \bullet, (\bullet, \text{prp}_j[j], \bullet))) \wedge$$
$$(p_j \in \text{allSeen}_k) \wedge (\text{all}_j[j] \wedge \text{FD}_j[j].part \subseteq (\text{allSeen}_j \cup \{p_j\}))$$

Proof. Let R'' be a prefix of R' that includes $O(xN)$ asynchronous rounds during which at least one processor $p_j \in \text{FD}_i[i].part$ takes at least $x \geq 13$ steps that change $(\text{prp}_j[j], myAll_j(j))$ without having conditions (i) to (iii) of Lemma 5.3.10 hold at any state of R''. Denote by $c_{j,y} \in R''$ the system state that immediately precedes the step $a_{j,y}$ in which p_j changes $(\text{prp}_j[j], myAll_j(j))$ for the y-th time during R''. We show that there exists $z \leq 13$ so that it is possible to choose $p_j \in \text{FD}_i[i].part$, such that Equation 5.2 holds in $c_{j,z}$.

Note that whenever p_j changes $(\text{prp}_j[j], myAll_j(j))$ during R' (in a way that does not assign dfltNtf to prp), it does so by taking a step that includes either a call to *estab*(), or an execution of line 3, or line 11 (that is, either a call to $maxNtf()$ or to *increment*()).

Showing that changes to $(\text{prp}_j[j], \text{myAll}_j(j))$'s degree are modulo six. We use the following two fact for observing that each individual processor in $p_j \in \text{FD}_i[i].part$ changes the value of $(\text{prp}_j[j], myAll_j(j))$ only in the modulo six manner.

(a) The changes from an even degree value to an odd one, i.e., the assignment of *true* to all, is according to the process carried out by lines 3 to 11.

(b) The changes from an odd degree value to an even one is according to *estab*() and *increment*(). Specifically, *estab*() changes the processor state only from degree 1 to 2 (while *increment*() takes no effect). The definition of *increment*() implies that a degree of 3 changes to 4 as well as 5 changes to 0.

Therefore, each $p_j \in \text{FD}_i[i]$ changes $(\text{prp}_j[j], myAll_j(j))$'s degree modulo six.

The values of (prp, myAll) change concurrently and in a unison manner. We can see that different processors in $\text{FD}_j[j].part$ are not more than one value away (modulo six) from each other in the system state $c_{j,y}$, which immediately precedes the step in which p_j changes $(\text{prp}_j[j], myAll_j(j))$. This is because the steps in which these changes occur, include the execution of line 6, which means that $\forall p_k \in \text{FD}[j].part : corrDeg(j,k)$ holds in $c_{j,y}$ since we assume that conditions (i) to (iii) of Lemma 5.3.10 do not hold during R'' and that the statement of Claim 5.3.11 does hold. Therefore, during R' changes to $(\text{prp}_j[j], myAll_j(j))$ are concurrent and in a unison manner.

Showing that $(\text{FD}_\mathbf{j}[\mathbf{j}].\mathbf{part} \subseteq (\mathbf{allSeen_j} \cup \{\mathbf{p_j}\}) \wedge (\mathbf{all_j}[\mathbf{j}] = \mathbf{true}))$**.** Recall the value of $z \leq 13 = 2 \cdot 6 + 1$, which can associate with increments that causes two modulo six wrap arounds and then one more increment. Due to the two arguments above, we can claim that p_j changes the value of $(\text{prp}_j[j], \text{all}_j[j])$ to $(\langle 2, \bullet\rangle, false)$ at least twice and then changes the value of $(\text{prp}_j[j], \text{all}_j[j])$ to $(\langle 0, \bullet\rangle, true)$ once more.

Let $c_{j,y} \in R'' : y \leq z - 2$ be the system state that immediately precedes the step $a_{j,y}$ in which p_j changes $(\text{prp}_j[j], \text{all}_j[j])$ to $(\langle 2, \bullet\rangle, false)$ for the second time during R'' and $c_{j,y'} \in R'' : y' = y + 2$ be the system state that immediately precedes the step $a_{j,y'}$ in which p_j changes $(\text{prp}_j[j], \text{all}_j[j])$ from $(\langle 2, \bullet\rangle, true)$ to $(\langle 0, \perp\rangle, false)$. In both times that p_j changes the value of $(\text{prp}_j[j], \text{all}_j[j])$ to $(\langle 2, \bullet\rangle, false)$, the if-statement condition of line 11 must be true. That is, in both times, $(\text{FD}_j[j].part \subseteq (\text{allSeen}_j \cup \{p_j\}) \wedge (\text{all}_j[j] = true))$ is true in the system states that immediately precede these two steps that change $(\text{prp}_j[j], \text{all}_j[j])$ to $(\langle 2, \bullet\rangle, false)$ and then p_j empties allSeen_j (lines11 and Fig. 5.3, line 14). This implies that between the first time and the second time that p_j changed $(\text{prp}_j[j], \text{all}_j[j])$ to $(\langle 2, \bullet\rangle, false)$, node p_j had emptied allSeen_j and then p_j changes $(\text{prp}_j[j], \text{all}_j[j])$ to $(\langle 1, \bullet\rangle, true)$ as well as adds all the elements in $(\text{FD}_j[j].part \setminus \{p_j\})$ to allSeen_j (line 5).

Showing that $\forall_{\mathbf{p_k}, \mathbf{p_\ell} \in \mathbf{FD_j[j].part}} \mathbf{prp_k}[\ell] = \mathbf{prp_j}[\mathbf{j}] \wedge (\mathbf{prp_j}[\mathbf{k}], \mathbf{all_j}[\mathbf{k}]) = (\mathbf{prp_j}[\mathbf{j}], \mathbf{all_j}[\mathbf{j}])$**.** Due to Claim 5.3.11 as well as line 3 and line 5, the only way that the above scenario could happen is by having the following.

(A) For each $p_k \in \text{FD}_j[j].part$, there are system states $c_{k,y_k}, c'_{k,y_k} \in R''$, such that c_{k,y_k} appears before c'_{k,y_k}, and $\bigwedge_{p_\ell \in \text{FD}_j[j].part}(echoNoAll_k(\ell))$ holds in c_{k,y_k} (imme-

diately before p_k takes a step that includes the execution of line 3) as well as ($\text{all}_j[k]$) holds in c'_{k,y_k} (immediately before p_j takes a step that includes the execution of line 5). We note that since $a_{j,y}$ refers to the time in which p_j changes $(\text{prp}_j[j], \text{all}_j[j])$ to $(\langle 1, \bullet\rangle, \mathit{false})$ for the second time in R'', then by Claim 5.3.11 and lines 3 and 5 we know that c_{k,y_k}, c'_{k,y_k} in R'', rather than just in R.

(B) By using similar arguments to the ones that appear above, for each $p_k \in \text{FD}_j[j].\mathit{part}$ there are system states $c_{k,y'_k}, c'_{k,y'_k} \in R''$, such that c'_{k,y_k} appears before c_{k,y'_k} (because of the way that *increment*() changes p_j's degree), c_{k,y'_k} appears before c'_{k,y'_k}, and $\bigwedge_{p_\ell \in \text{FD}_j[j].\mathit{part}}(\mathit{echoNoAll}_k(\ell))$ holds in c_{k,y'_k} as well as $\text{all}_j[k]$ holds in c'_{k,y'_k}.

Using the definitions of *echoNoAll*() (Fig. 5.3, line 11) we can write this as follows:

(A) $\forall p_k \in \text{FD}_j[j].\mathit{part} : \forall p_\ell \in \text{FD}_j[j].\mathit{part} : \text{prp}_k[\ell] = \text{prp}_j[j] = \langle 1, \bullet\rangle$ in c_{k,y_k} and $\forall p_k \in \text{FD}_j[j].\mathit{part} : (\text{prp}_j[k], \text{all}_j[k]) = (\text{prp}_j[j], \text{all}_j[j]) = (\langle 1, \bullet\rangle, \mathit{true})$ and in c'_{k,y_k}.

(B) $\forall p_k \in \text{FD}_j[j].\mathit{part} : \forall p_\ell \in \text{FD}_j[j].\mathit{part} : \text{prp}_k[\ell] = \text{prp}_j[j] = \langle 2, \bullet\rangle$ in c_{k,y'_k} and $\forall p_k \in \text{FD}_j[j].\mathit{part} : (\text{prp}_j[k], \text{all}_j[k]) = (\text{prp}_j[j], \text{all}_j[j]) = (\langle 2, \bullet\rangle, \mathit{true})$ in c'_{k,y'_k}.

Arguing about the first processor to change (prp, all). Suppose, without the loss of generality, that p_j is the first among all the processors $p_k \in \text{FD}_j[j].\mathit{part}$ that changes the value of $(\text{prp}_k[k], \text{all}_k[k])$ from $(\langle 2, \bullet\rangle, \mathit{true})$ to $(\langle 0, \bullet\rangle, \mathit{false})$ for the second time during R'' at step $a_{j,y'}$, which immediately follows $c_{j,y'}$. Since p_j is the first to change and any processor $p_k \in \text{FD}_j[j].\mathit{part}$ (including $p_j = p_k$) needs invariant (B) to hold in order to change the value of $(\text{prp}_k[k], \text{all}_k[k])$, we have that in $c_{j,y'}$ invariant (B) holds both for the case of c'_{k,y_k} and c'_{k,y'_k}. Therefore, $\forall_{p_k,p_\ell \in \text{FD}_j[j].\mathit{part}} : ((\text{prp}_k[\ell] = \text{prp}_j[j]) \wedge ((\text{prp}_j[k], \text{all}_j[k]) = (\text{prp}_j[j], \text{all}_j[j] = \mathit{false})))$ holds in $c_{j,y'}$. Using Claim 5.3.11 as well as lines 11, 14 and 15, we also get $\forall_{p_k,p_\ell \in \text{FD}_j[j].\mathit{part}} : (m \in \mathit{channel}_{j,k} \cup \mathit{channel}_{k,j} \implies m = (\bullet, \text{prp}_j[j], \text{all} = \mathit{false}, (\bullet, \text{prp}_j[j], \text{all} = \mathit{false})))$ as well as $p_j \in \text{allSeen}_k$ holds in $c_{j,y'}$. Since we have already showed $\text{all}_j[j] \wedge (\text{FD}_j[j].\mathit{part} \subseteq (\text{allSeen}_j \cup \{p_j\}))$, we have that Equation 5.2 holds in $c_{j,y'}$. Since $y' = z$ we get the result as required by the lemma. □

Lemma 5.3.13. *Suppose that R' starts from system state c, such that c is either $c^{\circledast}$ (Lemma 5.3.10) or c^* (Lemma 5.3.12). Execution R' does not include a system state in which Equation 5.1 holds.*

Proof. The lemma holds immediately for the case of $c^{\circledast}$ (Lemma 5.3.10). The rest of the proof considers the case of c^{*} (Lemma 5.3.12). We show that when $c = c^{*}$ is the starting system state of R', no system state in R' satisfies Equation 5.1.

The proof of Lemma 5.3.12 shows that there is a processor $p_j \in \mathrm{FD}_i[i].part$ for which Equation 5.2 holds in a system state (which does not satisfy Equation 5.1) and immediately precedes a step in which p_j changes $(\mathrm{prp}_j[j], \mathrm{all}_j[j] = true)$ to $((\mathrm{prp}_j[j].phase + 1) \pmod 3, \mathrm{all}_j[j] = false)$. This brings the system to a state $c' \in R'$ that does not satisfy Equation 5.1.

Note that in c' and in the system states that follow, processor p_j cannot change that value of $((\mathrm{prp}_j[j].phase + 1) \pmod 3, \mathrm{all}_j[j] = false)$ before every processor $p_k \in \mathrm{FD}_i[i].part \cup \{p_j\}$ changes the value of $(\mathrm{prp}_k[k], \mathrm{all}_k[k] = true)$. This is due to the process that line 3, 5 and 11 control. Moreover, the fact that Equation 5.1 does not hold in c', does not prevent from p_k to change the value of $(\mathrm{prp}_k[k], \mathrm{all}_k[k])$ via a call to the function *increment*() (line 11), because in c, and therefore in c', we have that $p_j \in \mathrm{allSeen}_k$ holds (cf. Equation 5.2). Therefore, by similar arguments to the ones that appear in the first part of the proof of Step 3 of Lemma 5.3.10, the system reaches a state, within $O(N)$ asynchronous rounds, in which Equation 5.3 holds (while not supporting Equation 5.1).

$$\forall_{p_k, p_\ell \in \mathrm{FD}_j[j].part} : ((\mathrm{prp}_k[\ell] = \mathrm{prp}_j[j]) \wedge ((\mathrm{prp}_j[k], \mathrm{all}_j[k]) = (\mathrm{prp}_j[j], \mathrm{all}_j[j] = false))) \wedge \tag{5.3}$$

$$(m \in channel_{j,k} \cup channel_{k,j} \implies m = (\bullet, \mathrm{prp}_j[j], \mathrm{all} = false, (\bullet, \mathrm{prp}_j[j], \mathrm{all} = false)))$$

Moreover, by using again similar arguments to the ones that appear in the first part of the proof of Step 3 of Lemma 5.3.10, the system reaches a state in which Equation 5.2 holds once more (while not supporting Equation 5.1). Furthermore, once Equation 5.2 holds, the same arguments can be repeated, as much as necessary, throughout R'. □

This concludes the proof of Theorem 5.3.9. □

Corollary 5.3.14 is implied by Equation 5.1 of Theorem 5.3.9.

Corollary 5.3.14. *Let R be an admissible execution w.r.t. participants of Algorithm 5 (which may include explicit delicate or spontaneous replacements). Denote R's starting system state*

by c and suppose that $\langle 1, \text{false}\rangle \in \text{NA}(c) \wedge (\langle 0, \text{false}\rangle \in \text{NA}(c) \vee \langle 2, \bullet\rangle \in \text{NA}(c))$. *Within* $O(N)$ *asynchronous rounds, either (i) the system reaches a state* $c' \in R$ *that encodes no notifications, or (ii) there is a step that includes a call to the function* $configSet(\perp)$ *(line 6).*

Algorithm 5 is self-stabilizing

Let $a_i \in R$ be a step in which processor p_i calls the function *estab*(set) (Fig. 5.3, line 5), and in which the if-statement condition (*allowReco*() $\wedge$ ($set \notin \{\text{config}[i], \emptyset\}$)) does hold in the system state that immediately precedes a_i. We say that a_i is an *effective (configuration establishment)* step in R. Similarly, we consider $a_i \in R$ to be a step in which processor p_i calls the function *participate*() (line 6), and in which the if-statement condition *allowReco*() does hold in the system state that immediately precedes a_i. Let $R = R' \circ R_{\text{VNER}} \circ R'''$ be an execution that does include explicit (delicate) replacements, where R' and R''' are a prefix, and respectively, a suffix of R. Let us consider R_{VNER}, which is a part of execution R. We say that R_{VNER} *virtually does not include explicit (delicate) replacements* (VNER) when for any step $a \in R_{\text{VNER}}$ that includes a call the function *estab*(set) (Fig. 5.3, line 5) or *participate*() (Fig. 5.3, line 6) is ineffective. Given a system state $c \in R$, we say that c includes no notification if none of its active processors stores a notification and there are no notifications in transit between any two active processors.

Lemma 5.3.15 (Eventually there is an VNER part)**.** *Let R be an admissible execution w.r.t. participants of Algorithm 5 (which may include explicit delicate or spontaneous replacements). (1) Within* $O(N)$ *asynchronous rounds, the system reaches a state* $c \in R$ *after which an VNER part,* R_{VNER}*, starts. (2) After c, the system reaches, within* $O(N)$ *asynchronous rounds, a state* $c_{\text{goodNtf}} \in R_{\text{VNER}}$ *that has either (2.1) no notifications, i.e.,* $\text{NA}(c_{\text{goodNtf}}) = \emptyset$ *or (2.2) exactly one notification in the system, i.e.,* $\text{NA}(c_{\text{goodNtf}}) = \{n\}$*, which becomes within* $O(1)$ *asynchronous rounds the system configuration after reaching a state in which invariants (1) to (7) of Lemma 5.3.8 hold in* c_{goodNtf}.

Proof. The proof looks into two cases. In the first case, we assume that R has a prefix R' of $O(N)$ asynchronous rounds in which there is at least one step a_x that includes a call to the function *estab*(set) (Fig. 5.3, line 5). The proof of this case shows that within $O(1)$ asynchronous rounds from a_x, the system reaches VNER execution part that allows the satisfaction of the conditions in parts (2) of this lemma within $O(N)$ asynchronous rounds. The second case assumes that no such step a_x effectively

calls the function *estab*(set) (Fig. 5.3, line 5) during R'. This assumption basically means that R' is an VNER execution and the rest of the proof of this case follows the arguments that we give the first case. Therefore, the reminder of the proof focuses only on the first case and assume that R has a prefix R' of $O(N)$ asynchronous rounds in which there is at least one step a_x in which a processor calls the function *estab*(set) (Fig. 5.3, line 5).

Let $c'' \in R$ be a system state that occur $O(N)$ asynchronous rounds after the first effective step $a_x \in R'$ that includes a call to the function *estab*(set). We denote by R'' the prefix of R that ends at c''. Let R''' be the prefix of R that extends R'' by $O(1)$ asynchronous rounds. Recall that the proof is done if during R''' the system either (i) reaches a state that encodes no notifications, i.e., $\text{NA}(c_{\text{goodNtf}}) = \emptyset$, (ii) includes a step in which there is a call to the function $configSet(\perp)$, or (iii) includes an VNER part. We show that R''' includes one of these cases.

Suppose that $\exists \mathbf{c}^{\circledast} \in \mathbf{R}'' : \mathbf{c}^{\circledast} \neq \mathbf{c}'' \wedge \langle \mathbf{1}, \text{false} \rangle \in \text{NA}(\mathbf{c}^{\circledast}) \wedge (\langle \mathbf{0}, \text{false} \rangle \in \text{NA}(\mathbf{c}^{\circledast}) \vee \langle \mathbf{2}, \bullet \rangle \in \text{NA}(\mathbf{c}^{\circledast}))$. It is implied by Corollary 5.3.14 that either (i) in the last system state c'' of R'', it does not hold that $\langle 1, \text{false} \rangle \in \text{NA}(c'') \wedge (\langle 0, \text{false} \rangle \in \text{NA}(c'') \vee \langle 2, \bullet \rangle \in \text{NA}(c''))$, or (ii) R''' includes a step in which there is a call to the function $configSet(\perp)$ (line 6). In both cases the proof is done, because it satisfies part (2.1).

Suppose that $\nexists \mathbf{c}^{\circledast} \in \mathbf{R}'' : \langle \mathbf{1}, \text{false} \rangle \in \text{NA}(\mathbf{c}^{\circledast})$**.** In this case, R'' is an VNER part of execution R, because it cannot be that R'' includes an effective step that includes a call to the function *estab*(set) (Fig. 5.3, line 5).

Suppose that $\nexists \mathbf{c}^{\circledast} \in \mathbf{R}'' : (\langle \mathbf{0}, \bullet \rangle \in \text{NA}(\mathbf{c}^{\circledast}) \vee \langle \mathbf{2}, \bullet \rangle \in \text{NA}(\mathbf{c}^{\circledast}))$**.** This case implies that $\forall c^{\circledast} \in R'' : (\langle 1, \bullet \rangle \in \text{NA}(c^{\circledast}))$. This means that $\text{prp}_i[j] = \langle 1, \bullet \rangle$, where $p_i, p_j \in \mathcal{I}$ are active participants in R. Therefore, R'' is an VNER part of execution R (by definition of VNER).

In the latter two cases, we showed that R'' is a VNER part of R (which means R''' includes a VNER part). For these cases, the proof of part (2) follows from Lemma 5.3.8 and Theorem 5.3.9. □

Theorem 5.3.16 demonstrates the eventual absence of stale information, which implies that Algorithm 5 convergences eventually, i.e., it is self-stabilizing.

Theorem 5.3.16 (Convergence)**.** *Let R be an admissible execution of Algorithm 5. Within $O(N)$ asynchronous rounds the system reaches a state $c \in R$ in which none of the invariants*

of type-1, type-2, type-3 and type-4 of stale information hold thereafter.

Proof. Lemma 5.3.1 shows that there is no type-1 stale information eventually ($O(1)$ asynchronous rounds). Lemma 5.3.2 shows that there is no type-2 stale information eventually ($O(1)$ asynchronous rounds). Lemmas 5.3.8 and 5.3.15 say that the system either reaches a state in which there are no notifications or there is at most one notification (for which invariants (1) to (7) of Lemma 5.3.8 hold) that later becomes the system configuration. Note that both cases imply that there is no type-3 stale information eventually ($O(N)$ asynchronous rounds). Lemma 5.3.7 shows that eventually there is no type-4 stale information ($O(1)$ asynchronous rounds). □

Theorem 5.3.17 (Closure)**.** *Let R be an execution of Algorithm 5. Suppose that execution R starts from a system state, c, that includes no stale information. (1) For any system state $c \in R$, it holds that c includes no stale information. Suppose that the step that immediately follows c includes a call to estab*(). *(2) The only way that* set *becomes a notification is via a call to estab*(set) *(Fig. 5.3, line 5) and the only way that a processor becomes a participant is via a call to participate*() *(Fig. 5.3, line 6). (3) If notifications exist, the configuration is replaced within $O(N)$ asynchronous rounds.*

Proof. The proof essentially follows from established results above.

Part (1). Since there is no stale information in the system state that immediately proceeds *c*, there is no stale information in *c*. This follows from a close investigation of the lines that can change the system state in a way that might introduce stale information; the most relevant lines are the ones that deal with notifications (line 3 and lines 5 to 12) and new participants (line 16). Thus, the proof is completed via Lemma 5.3.15 as well as parts (2) and (3) of this proof.

Part (2). This is immediate from lines 5 and 6 (of Fig. 5.3).

Part (3). It is not difficult to see that *R* includes an R_{VNER} part (Lemma 5.3.15, Part (1)). Then, the proof completes by applying Lemma 5.3.8 and Theorem 5.3.9 as well as 5.3.15 twice: Once for showing the selection of a single notification during phase 1 (before moving to phase 2), and the second time for showing that the selected notification replaces the quorum configuration (before returning to phase 0). Specifically, Lemma 5.3.15 says that the conditions for applying Lemma 5.3.8 holds until returning to phase 0, and Theorem 5.3.9 bounds the time that it takes ($O(N)$ asynchronous rounds). □

5.4 Reconfiguration Management

The Reconfiguration Management *recMA* layer shown in Algorithm 6 bears the weight of initiating or *triggering* a reconfiguration when (i) the configuration majority has been lost, or (ii) when the prediction function $evalConf()$ indicates to a majority of members that a reconfiguration is needed to preserve the configuration. To trigger a reconfiguration, Algorithm 6 uses the *estab*(*set*) interface with the *recSA* layer. In this perspective, the two algorithms display their modularity as to their workings. Namely, *recMA* controls *when* a reconfiguration should take place, but the reconfiguration replacement process is left to *recSA*, which will install a new configuration also trying to satisfy *recMA*'s proposal of the new configuration's set. Several processors may trigger reconfiguration simultaneously, but, by the correctness of Algorithm 5, this does not affect the delicate reconfiguration, and by the correctness of Algorithm 6, each processor can only trigger once when this is needed.

In spite of using majorities, the algorithm is generalizable to other (more complex) quorum systems, while the prediction function $evalConf()$ can be either very simple, e.g., asking for reconfiguration once $1/4^{th}$ of the members are not trusted, or more complex, based on application criteria or network considerations. More elaborate methods may also be used to define the set of processors that Algorithm 6 proposes as the new configuration. Our current implementation, aiming at simplicity of presentation, defines the set of trusted participants of the proposer as the proposed set for the new configuration.

5.4.1 Algorithm Description

Preserving a majority. The algorithm strives to ensure that a majority of the configuration is active. Although majority is a special case of a quorum, the solution in extensible to host other quorum systems that can be built on top of the $config$ set, in which case, the algorithm aims at keeping a robust quorum system where robustness criteria are subject to the system's dynamics and application requirements. In this vein, the presented algorithm employs a configuration evaluation function $evalConf()$ used as a black box, which predicts the quality of the current $config$ and advises any participant whether a reconfiguration of $config$ needs to take

Algorithm 6: Self-stabilizing Reconfiguration Management; code for proc. p_i

1 **Interfaces:** $evalConf()$ returns True/False on whether a reconfiguration is required or not by using some (possibly application-based) prediction function. The rest of the interfaces are specified in Algorithm 5. *allowReco*() returns True if a reconfiguration is not taking place, or False otherwise. *estab*(*set*) initiates the creation of a new configuration based on the *set*. $getConfig()$ returns the current local configuration.

2 **Variables:** $needReconf[\,]$ is an array of size at most N, composed of booleans {True, False}, where $needReconf_i[j]$ holds the last value of $needReconf_j[j]$ that p_i received from p_j as a result of exchange (lines 19 and 20) and $needReconf$ is an alias to $needReconf_i[i]$, i.e., of p_i's last reading of $evalConf()$. Similarly, $noMaj_i[\,]$ is an array of booleans of size at most N on whether some trusted processor of p_i detects a majority of members that are active per the reading of line 12. $noMaj_i[j]$ (for $i \neq j$) holds the last value of $noMaj_j[j]$ that p_i received from p_j. Finally, $prevConfig$ holds p_i's believed previous $config$.

3 **macro** $core() = \bigcap_{p_j \in \mathrm{FD}_i[i].part} FD[j].part$;

4 **macro** $flushFlags() = \{$**foreach** $p_j \in FD[i]$ **do** $needReconf[j] \leftarrow (noMaj[j] \leftarrow \mathsf{False})\}$;

5 **Do forever begin**

6 **if** $p_i \in FD[i].part$ **then**

7 $curConf = getConfig()$;

8 $needReconf[i] \leftarrow (noMaj[i] \leftarrow \mathsf{False})$;

9 **if** $prevConfig \notin \{curConf, \perp\}$ **then** $flushFlags()$;

10 **if** *allowReco*() = True **then**

11 $prevConfig \leftarrow curConf$;

12 **if** $(|\{p_j \in curConf \cap FD[i]\}| < (\frac{|curConf|}{2} + 1))$ **then** $noMaj[i] \leftarrow \mathsf{True}$;

13 **if** $(noMaj[i] = \mathsf{True}) \wedge (|core()| > 1) \wedge (\forall p_k \in core() : noMaj[k] = \mathsf{True})$ **then**

14 *estab*($FD[i].part$);

15 $flushFlags()$;

16 **else if** $(needReconf[i] \leftarrow evalConf(curConf)) \wedge$
$(|\{p_j \in curConf \cap FD[i] : needReconf[j] = \mathsf{True}\}| > \frac{|curConf|}{2})$ **then**

17 *estab*($FD[i].part$);

18 $flushFlags()$;

19 **foreach** $p_j \in FD[i].part$ **do** $send(\langle noMaj[i], needReconf[i]\rangle)$;

20 **Upon receive** m **from** p_j **do if** $p_i \in FD[i].part$ **then** $\langle noMaj[j], needReconf[j]\rangle \leftarrow m$;

place. Given that local information is possibly inaccurate, we prevent unilateral reconfiguration requests –that may be the result of inaccurate failure detection– by demanding that a processor must first be informed of a majority of processors in the current $config$ that also require a reconfiguration (lines 16–18).

Majority failure. On the other hand, we ensure liveness by handling the case where either the prediction function does not manage to prevent the collapse of a majority, or an initial arbitrary state lacks a majority but there are no $config$ inconsistencies that can trigger a delicate reconfiguration (via the *estab*() interface). Lines 13–15 tackle this case by defining the *core* of a processor p_i to be the intersection of the failure detector

readings that p_i has for the processors in its own failure detector, i.e., $\cap_{p_j \in \text{FD}_i[i]} \text{FD}_i[j]$. If this local core agrees that there is no majority, i.e. that $noMaj$ = True, then p_i can request a new $config$. As a liveness condition to avoid triggering a new $config$ due to FD inconsistencies when there actually exists a majority of active configuration members, we place the *majority-supportive core* assumption on the failure detectors, as seen in Definition 5.4.1 below. Simply put, the assumption requires that if a majority of the current configuration is active, then the core of every processor p_t that is a participant, contains at least one processor p_s with a failure detector supporting that a majority of $config$ is trusted. Furthermore, p_t has knowledge that p_s can detect a majority of trusted members.

Detailed description. The algorithm is essentially executed only by participants as the condition of line 6 suggests. Line 7 reads the current configuration, while line 8 initiates the local $noMaj_i[i]$ and $needReconf_i[i]$ variables to False. If a change from the previous configuration has taken place, the arrays $noMaj[\,]$ and $needReconf[\,]$ are reset to False (line 9). The algorithm proceeds to evaluate whether a reconfiguration is required by first checking whether a reconfiguration is already taking place (line 10) through the *allowReco*() interface of *recSA*. If this is not the case, then it checks whether it can see a trusted majority of configuration members, and updates the local $noMaj_i[i]$ boolean accordingly (line 12). If $noMaj_i[i]$ = True, i.e., no majority of members is active, and line 13 finds that all the processors in its core also fail to find a majority of members, then p_i can trigger a reconfiguration using *estab*(*set*) with the current local set of participants as the proposed new configuration *set* (lines 14–15). The $needReconf_i[\,]$ and $noMaj_i[\,]$ arrays are again reset to False to prevent other processors that will receive these to trigger. Line 16 checks whether the prediction function *evalConfig*() suggests a reconfiguration, and if a majority of members appears to agree on this, then the triggering proceeds as above. Participants continuously exchange their $noMaj$ and $needReconf$ variables (lines 19–20).

5.4.2 Correctness

The Reconfiguration Management algorithm is responsible for triggering a reconfiguration when either a majority of the members crash or whenever the (application-based) $config$ evaluation mechanism *evalConfig*() suggests to a members' majority

that a reconfiguration is required. The correctness proof ensures that, given the assumption of majority-supportive core holds, Algorithm 6 can converge from a transient initial state to a safe state, namely, that after *recMA* has triggered a reconfiguration, it will never trigger a new one before the previous one is completed and only if a new event makes it necessary.

Terminology. We use the term *steady config state* to indicate a system state in an execution where a *config* has been installed by Algorithm 5 at least once, and the system state is conflict-free. A *legal execution R* for Algorithm 6, refers to an execution that converges to a steady *config* state. Moreover, a reconfiguration in *R* takes place only when a majority of the configuration members fails, or when a majority of the members requires a reconfiguration. The system remains conflict-free and moves to a new steady *config* state with a new configuration. To guarantee progress we suggest the following liveness assumption.

Definition 5.4.1 (Majority-supportive core)**.** *Consider a steady config state in an execution R where the majority of members of the established config never crashes. The* majority-supportive core *assumption requires that every participant p_i with a local core $\cap_{p_j \in \mathrm{FD}_i[i]} \mathrm{FD}_i[j]$ containing more than one processor, must have a core with at least one active participant p_r whose failure detector trusts a majority of the config, and for such a processor $noMaj_i[r] =$ False throughout R.*

Remark: We say that Algorithm 5 is *triggered* when a reconfiguration is initialized. By Algorithm 6, the only way that Algorithm 6 can cause a triggering of Algorithm 5 is through a call to the *estab*() interface with Algorithm 5 on lines 14 and 17.

Proof Overview. The correctness of *recMA* first considers local stale information and then stale information in the communication are removed (Lemmas 5.4.1 and 5.4.2). It then proceeds to prove that given the majority supportive core assumption, if there is a configuration majority, this is not abruptly overthrown, and if the majority has collapsed, or the evaluation function requires a view change, this eventually takes place (Lemmas 5.4.3 and 5.4.4). Lemma 5.4.5 is vital since it guarantees that this layer does not force reconfiguration requests while a reconfiguration takes place at the lower layer. Theorem 5.4.6 concludes the proof.

Lemma 5.4.1. *Starting from an arbitrary initial state of an execution R, where stale information exists, then, within $O(N)$ asynchronous rounds, Algorithm 6 converges to a steady $config$ state where local stale information and stale information in the communication links is removed.*

Proof. Consider a processor p_i with an arbitrary initial local state where stale information exists (1) in the program counter, (2) in $noMaj_i[\bullet]$ and $needReconf_i[\bullet]$ and (3) in $prevConf$.

Case 1 – Stale information may initiate the algorithm in a line other than the first of the pseudocode. If p_i's program counter starts after line 10 and if a reconfiguration is taking place, then Algorithm 6 may force a second reconfiguration while Algorithm 5 is already reconfiguring. The counter for example could start on lines 14 and 17. This would force a brute reconfiguration. This triggering cannot be prevented in such a transient state, but we note that any subsequent iteration of the algorithm is prevented from accessing *estab*() lines (as in Remark 5.4.2) before the reconfiguration is finished.

Case 2 – We note that after a triggering as the one described above, the fields of arrays $noMaj_i[\bullet]$ and $needReconf_i[\bullet]$ are set to False. Moreover, in every iteration $noMaj_i[i]$ and $needReconf_i[i]$ are set to False. During a reconfiguration these values are propagated to other processors and p_i receives their corresponding values. Within $O(1)$ asynchronous rounds, p_i receives $noMaj_j[j]$ ($needReconf_j[j]$) from some participant p_j, and overwrites any transient values. Lemma 5.4.2 bounds the number of reconfigurations that may be triggered by corruption that is not local, i.e., that emerges from corrupt $noMaj$ ($needReconf$) values that arrive from the communication links.

Case 3 – We anticipate that any reconfiguration returns a different configuration than the previous one. In a transient state though, the previous configuration ($prevConfig$) and the current configuration $curConf$ may coincide. This ignores the check of line 9 that sets $noMaj_i[\bullet]$ and $needReconf_i[\bullet]$ to False upon the detection of a reconfiguration change. This forms a source of a potential unneeded reconfiguration. Nevertheless, $prepConfig$ receives the most recent configuration value on every iteration of line 11 and thus the above may only take place once throughout R per processor.

Eliminating these sources of corruption, we reach a steady $config$ state without local stale information. Note that the result holds within just $O(1)$, but since the reconfiguration service *recSA* layer requires $O(N)$, the worse case analysis is $O(N)$. □

Lemma 5.4.2. *Consider a steady $config$ state c in an execution R where the majority-supportive core assumption holds throughout, the majority of $config$ processors never crash and there is never a majority of members supporting $evalConf()$ = True. There is a bounded number of triggerings of Algorithm 5 that are a result of stale information, namely $O(N^2 cap)$. These are removed within $O(N)$ asynchronous rounds.*

Proof. By Remark 5.4.2, the only way that the algorithm may interrupt a steady $config$ state, is by reaching lines 14 and 17 that have a call to *estab*(). We assume that some member $p_t \in config$ has triggered Algorithm 5 at some system state $c_t \in R$, and we examine whether and when this state is attainable from c. We note that *in a complete iteration* of Algorithm 6, p_t must have no reconfiguration taking place while triggering, since this is a condition to reach the above mentioned lines imposed by line 10. We first prove that if there is a triggering it must be due to initial corrupt information and then argue that this can take place a bounded number of times.

Case 1 – The reconfiguration was initiated by line 14. This implies that the condition of line 13 is satisfied, i.e., at some system state $c_t \in R$, p_t has local information that satisfies $(noMaj_t[t] = \text{True}) \wedge (|core_t()| > 1) \wedge (\forall p_k \in core_t() : noMaj_t[k] = \text{True})$. Condition $(noMaj_t[t] = \text{True})$ may be true locally for p_t, only due to failure detector inaccuracy, because, by the claim, the majority of processors in the $config$ never fails throughout R. Condition $|core_t()| > 1$ suggests that p_t has at least two participant processors in its core (without requiring $p_t \in core_t()$). By the majority-supportive core assumption and the above, we are guaranteed that $\exists p_s \in core_t() : |\text{FD}_s[s] \cap config_s[s]| > \frac{|config_s[s]|}{2}$ throughout R and $noMaj_t[s] = \text{True} \iff noMaj_s[s] = \text{True}$. But in this case, $noMaj_s[s] = \text{False}$ and $noMaj_t[s] = \text{True}$ which contradicts the majority supportive assumption. We thus reach to the result.

Note that $noMaj_t[s] = \text{True}$ can reside in p_t's local state or in the communication links that may carry stale information. Because of the boundedness of our system, we can have one instance of corrupt $noMaj_t[s] = \text{True}$ in p_t's local state, and cap instances in the communication link. I.e., such information may cause a maximum of $1 + cap \cdot N$ triggerings per processor. Any processor that enters the system cannot introduce corrupt information to the system due to the data-links protocols and the joining mechanism. Thus majority supportive assumption is also attainable even when starting from arbitrary states.

Case 2 – The reconfiguration procedure was triggered by line 16. This implies that

for p_t, both conditions were true, i.e., (a) ($needReconf_t \leftarrow evalConf_t(config_t)$) and (b) $|\{p_j \in config_t \cap \text{FD}_t : needReconf_t[j] = \text{True}\}| > \frac{|config_t|}{2}$. We note that the $needReconf_t[t]$ variable is always set to False upon the beginning of every iteration. Thus the local function $evalConf_t()$ due to p_t's failure detector and other application criteria explicitly suggested a reconfiguration *in the specific iteration* in which p_t triggered the reconfiguration. From the claim, there is no majority of processors in the $config$ that supports a reconfiguration, even at the time when p_t triggered the reconfiguration.

Thus $needReconf_t[s] = \text{True}$ must reside in p_t's local state and in the communication links. We can have one instance of corrupt $needReconf_t[s] = \text{True}$ in p_t's local state, and cap instances in the communication links. I.e. such information may cause a maximum of $1 + cap \cdot N$ triggerings per processor. Note that after every such triggering, the source of triggering is eliminated by reseting $needReconfig[\,]$ to False (lines 9, 15 and 18). From this point onwards any processor that enters the system cannot by the data-links and the joining mechanism introduce corrupt information to the system.

So the possible triggerings in the system attributed to stale information are confined to $O(N^2 cap)$ and by Algorithm 5 guarantees we always reach a steady $config$ state within $O(N)$ asynchronous rounds. These are eliminated within $O(1)$ asynchronous rounds by the definition of an asynchronous round and the reliable FIFO data links. □

Let c_{safe} denote a *safe system state* where all possible sources of triggerings attributed to the arbitrary initial state have been eliminated. We denote an execution starting from c_{safe} as R_{safe}.

Lemma 5.4.3. *Consider an execution R_{safe} where the majority-supportive core assumption holds throughout, the majority of $config$ processors never crash and there is never a majority of the $config$ with local $evalConf()$ = True. This execution is composed of only steady $config$ states.*

Proof. By Lemma 5.4.2 there is a bounded number of triggerings due to initial arbitrary information. Given that we have reached a safe system state, these triggerings do not occur. The last $config$ change, has by line 9 reset all the fields in $noMaj[\,]$ and $needReconf[\,]$ to False and this holds for all participants (even if they are not members of the $config$). By our assumption a majority of processors does not crash.

The majority-supportive core assumption states that throughout R_{safe} there exists at least one processor p_i in the core of p_t that always has $noMaj_i[i] = \mathsf{False}$ and p_t has $noMaj_t[i] = \mathsf{False}$. Thus the condition of line 13 can never be true, and thus there is no iteration of the algorithm that can reach line 14. Similarly, since no majority of processors in the $config$ change to $needReconf = \mathsf{True}$ in this execution, and the local states are exchanged continuously over the token-based data-link, line 17 cannot be true. Thus any system state in R is a steady $config$ state. □

Lemma 5.4.4. *Starting from an R_{safe} execution, Algorithm 6 guarantees that if (1) a majority of $config$ members collapse or if (2) a majority of members require a reconfiguration as per the prediction function, within $O(1)$ asynchronous rounds a reconfiguration takes place.*

Proof. We consider the two cases separately.

Case 1 – If a majority of the members collapses, then based on the failure detector's correctness, a non-crashed participant p_t will eventually stop including a majority of $config$ members in its failure detector and participants (*FD.part*) set. We remind that rejoins are not permitted. Since the majority-supporting core assumption does not apply in this case, any processor in p_t's core must eventually reach to the same conclusion as p_t. Every such participating processor $p_s \in core_t()$ propagates $noMaj_s[s] = \mathsf{True}$ in every iteration. By the assumption that a packet sent infinitely often arrives infinitely often, any processor such as p_t collects a $noMaj = \mathsf{True}$ from every member like p_s core and thus within $O(1)$ asynchronous rounds enables a reconfiguration.

Case 2 – The arguments are similar to Case 1. The difference lies in that the processor p_t must within $O(1)$ asynchronous rounds receive $needReconfig = \mathsf{True}$ from a majority of $config$ members (rather than the local core processors) before it moves to trigger a reconfiguration. □

Lemma 5.4.5. *Starting from an R_{safe} execution, any triggering of Algorithm 5 (lines 14 and 17) related to a specific event (majority collapse or agreement of majority to change $config$), can only cause a one per participant concurrent trigger. After the $config$ has been established, no triggerings that relate to this event take place.*

Proof. We consider the two cases that can trigger a reconfiguration (Remark 5.4.2), and assume that p_t is the first processor to trigger *estab*(). Assume first that p_t has called Algorithm 5 two consecutive times, without a $config$ being *completely* established between the two calls. Note that a processor can access *estab*() in either of

lines 14 or 17 but not both in a single iteration. A call to *estab*() initiates a reconfiguration and thus any subsequent check of p_t in line 10 returns False from p_t's *recSA* layer. Thus p_t cannot access lines 14 or 17 until the reconfiguration has been completed. This implies that p_t can never trigger for a second time unless the new $config$ has been established. Note that if p_t triggers, another processor satisfying the conditions of line 10 may trigger concurrently, but is also subject to the trigger-once limitation. On the other hand, due to the exchange of information in Algorithm 5, when one processor triggers other processors eventually find their proposals and join the reconfiguration. So not every single processor's Reconfiguration Management module needs to trigger. Convergence to a single $config$ is guaranteed by Algorithm 5.

We conclude by indicating that lines 15 and 18 reset both arrays $noMaj_t[\,]$ and $needReconf_t[\,]$ immediately after *estab*(). Thus the triggering data used for this event are not used again. Moreover, upon configuration change, the same arrays are again set to False for the processors that have not triggered Algorithm 5 themselves through Algorithm 6. We thus reach a new steady $config$ state, and no more triggerings can take place due to the same event that had caused the reconfiguration. □

Theorem 5.4.6. *Let R be an execution starting from an arbitrary system state. Within $O(N)$ asynchronous rounds, Algorithm 6 guarantees that R reaches an execution suffix R_{safe} which is a legal execution.*

Proof. By Lemmas 5.4.1 and 5.4.2, we are guaranteed that we reach a safe system state c_{safe} where stale information from the arbitrary initial state cannot force a triggering of new $config$. This is the suffix R_{safe}. Lemma 5.4.3 ensures that after we have reached c_{safe}, and until a new triggering takes place that is caused by a loss of majority or a majority of the $config$ deciding to reconfigure, the current $config$ will not be changed for any other reason. Lemma 5.4.5 guarantees that after a change, we return to a steady $config$ state. Hence, within $O(N)$ asynchronous rounds required by Algorithm 5, and another $O(1)$ asynchronous rounds, we reach to the legal execution R_{safe} . □

5.5 Joining Mechanism

Every processor that wants to become a participant, uses the snap stabilizing data-link protocol (see Section 5.1) so as to avoid introducing stale information after it establishes a connection with the system's processors. Algorithm 5 enables a joiner to obtain the agreed $config$ when no reconfiguration is taking place. Note that, in spite of having knowledge of this $config$, a processor should only be able to participate in the computation if the application allows it. In order to sustain the self-stabilization property, it is also important that a new processor initializes its application-related local variables to either default values or to the latest values that a majority of the configuration members suggest. The joining protocol, Algorithm 7, illustrates the above and introduces joiners to the system, but only as *participants* and not as $config$ *members*.

The critical difference between a participant and a joiner is that the first is allowed to send configuration information via the *recSA* layer, whereas the latter may only receive.

5.5.1 Algorithm description

The algorithm is executed by non-participants, while participants only execute the communication side (line 18).

The joiner's side. Upon a call to the *join*() procedure, a joiner sets all the entries of its *pass*[ ] array to False (line 5) and resets application-related variables to default values, (lines 8). The processor then enters a do-forever loop, the contents of which it executes only while it is not a participant (line 7). A joiner then enters a loop in which it tries to gather enough support from a majority of configuration members. In every iteration, the joiner sends a "Join" request (line 14) and stores the True/False responses by any configuration member p_j in $pass[j]$, along with the latest application *state* that p_j has send. If a majority of active members has granted a $pass$ = True and there is no reconfiguration taking place, then *participate*() is called to allow the joining processor to become a participant.

The participant's side. A participant only executes the do–forever loop (line 6), but none of its contents since it always fails the condition of line 7. Participants

Algorithm 7: Self-stabilizing Joining Mechanism; code for processor p_i

1 **Interfaces.** The algorithm uses following interfaces from Algorithm 5. *allowReco*() returns True if a reconfiguration is not taking place. *participate*() makes p_i a participant. $getConfig()$ returns the agreed configuration from Algorithm 5 or $\perp$ if reconfiguration is taking place. The *passQuery*() interface to the application, returns a *pass*, i.e., a Boolean True/False in response to a request to grant a joining permission to a joining processor.

2 **Variables.** *state*[] is array of containing application states, where *state*[i] represents p_i's local variables and *state*[j] the state that p_i most recently received by p_j. *pass*[] collects all the passes that p_i receives from configuration members.

3 **Functions.** *resetVars*() initializes all variables related to the application based on default values. *initVars*() initializes all variables related to the application, based on the states exchanged with the configuration members.

4 **procedure** *join*() **begin**

5 **foreach** $p_j \in FD$ **do** $pass[j] \leftarrow$ False;

6 **do forever begin**

7 **if** $p_i \notin FD[i].part$ **then**

8 *resetVars*();

9 **repeat**

10 **let** $comConf = getConfig()$;

11 **if** $allowReco() \wedge (|\{p_j \in comConf \cap FD[i] : pass[j] = \text{True}\}| > \frac{|comConf|}{2})$ **then**

12 *initVars*();

13 *participate*();

14 **foreach** $p_j \in FD[i].part$ **do send**("Join");

15 **until** $p_i \in FD[i].part$;

16 **upon receive** ("Join") **from** $p_j \in FD \setminus FD[i].part$ **do begin**

17 **if** $p_i \in config \wedge allowReco() =$ True **then send**$(\langle passQuery(), state_i \rangle)$;

18 **upon receive** $m = \langle pass, state \rangle$ **from** $p_j \in FD$ **do if** $p_i \notin FD[i].part$ **then** $\langle pass[j], state[j] \rangle \leftarrow m$;

however respond to join requests (line 17) by checking whether a joining processor has the correct configuration, and whether a reconfiguration is not taking place, as well as if the application can accept a new processor. If the above are satisfied, then the participant sends a *pass* = True and its applications' *state*, otherwise it responds with False.

5.5.2 Correctness

The term *legal join initiation* indicates a processor's attempt to become a participant by initiating Algorithm 7 on line 4, and not on any other line of the *join*() procedure. If the latter case occurred it would indicate a corruption to the program counter.

Lemma 5.5.1. *Consider an arbitrary initial state in an execution R. There are up to N possible instances of processors introducing corruption to the system. Within $O(N)$ asynchronous rounds we reach a corruption-free execution suffix of R.*

Proof. Processors may be found with an uninitialized or falsely initialized local state due to a transient fault in their program counter which allowed them to reach line 12 without a legal join initiation. In an arbitrary initial state, any processor with stale information may manage to become a participant. There can be up to N such processors, i.e., the maximal number of active processors. Nevertheless, a processor trying to access the system after this, is forced to start the execution of *join*() from line 4. From this, and the fact that stale information is removed within $O(N)$ asynchronous rounds (via Algorithm 5), the result follows. □

Lemma 5.5.2. *Consider any processor p_i performing a legal join initiation. In the existence of other participants in the system, this processor never becomes a participant through the join() procedure during reconfiguration.*

Proof. We consider the situation where participants exist and reconfiguration is taking place, thus *allowReco*() is False. In order for p_i to become a participant, it needs to gather a pass from at least a majority of the configuration members. This can only happen if a configuration is in place, and if each of these members is not reconfiguring. Thus if a pass is granted, it must be that during the execution more than a majority of True passes have arrived at p_i. Note that since the propagation of passes is continuous if a reconfiguration starts, then passes can also be retracted. Finally, since getting a majority of passes can coincide with the initialization of a reconfiguration, we note that due to asynchrony this processor is considered a participant of the previous configuration, since it has full knowledge of the system's state and is also known by the previous configuration members. □

Lemma 5.5.3. *Consider an execution R where Lemma 5.5.2 holds, such that during R, a processor p becomes a participant. Then p cannot cause a reconfiguration, unless there exists a majority of the configuration set, or if there is no majority of the $config$ that requires a reconfiguration.*

Proof. We assume that p enters the computation with a legal join initiation. If p triggers a reconfiguration in the absence of the above two cases, then this implies that p has managed to become a participant while carrying corrupt information which have triggered a reconfiguration either directly or indirectly (through Algorithm 6). Corruption can either be local or in the communication links. Since the snap-stabilizing data-link protocol runs before the processor calls *join*(), this removes

data-link corruption for newly joining participants. We turn to the case of a corrupt local state. By the legal join initiation assumption, p must have reset its state on line 8. Before joining, the majority of members must acknowledge the latest state of p and p initiates its variables to legal values. It is therefore impossible that p can become a participant while it carries a corrupt state. Therefore, p cannot cause a reconfiguration. □

Theorem 5.5.4. *Consider an arbitrary initial state of an execution R of Algorithm 7. Within $O(N)$ asynchronous rounds the system reaches an execution suffix in which every joining processor p will continue trying to join a participant if the application allows it. Additionally, this new processor cannot trigger a delicate reconfiguration before becoming a participant and cannot trigger a delicate reconfiguration without majority loss or majority agreement after it becomes a participant.*

Proof. By Lemma 5.5.1, within $O(N)$ asynchronous rounds the system reaches an execution suffix where all joining processors enter the computation with a legal join initiation and a configuration installed and known. We assume that a reconfiguration is not taking place, that messages sent infinitely often are eventually received infinitely often, and that the application interface invoked by the participating processors allows p to join. Then p will eventually have a failure detector including a majority of member processors and will send its "Join" request to a majority (line 14). Since there is no reconfiguration taking place, p must learn the current configuration from Algorithm 5, which should agree with the $config$ held by other processors. Thus each member must grant a pass to p by sending True through line 17. Therefore, p will gather a majority supporting its entrance and will eventually satisfy line 11. This allows it to reach line 13 and thus p becomes a participant. Notice that if the application does not give permission of entry via *passQuery*(), then p cannot become a participant unless this changes, but p will continue sending requests. Finally, Lemma 5.5.3 ensures that the new participant does not cause perturbations to the current configuration, and hence the result. □

5.6 Applications of the Reconfiguration Scheme

Several self-stabilizing algorithms for the message-passing system, are designed for a static membership set that may suffer crash failures. Our reconfiguration

scheme allows for such applications to endure more adverse membership dynamics. When a configuration exists and no reconfiguration is taking place, the applications work in the same way as in their static version, since they run their service on the configuration set in the same way as in the original static setting. A main consideration, however, is what takes place *during* a reconfiguration, and how the service continues *after* this event.

A general framework for adapting the static algorithm to form a reconfigurable one, involves developing an interface between the application algorithm that defines what takes place during a reconfiguration and after the reconfiguration. Namely, how is the algorithm adapted to work with the new set of processors (of the new configuration) that it will be using to provide service. This would involve changes in the size and content of data structures and the set of processors with which it communicates. We note that using this framework, the algorithms are *suspending*, namely, they do not provide service *during* reconfiguration, albeit we believe that it is possible (under certain conditions) to find more elaborate frameworks that are able to sustain service during reconfiguration.

In general, it remains an interesting open question whether a *self-stabilizing* service, such as reconfigurable SMR or distributed shared memory that does not suspend, is possible. In [93], Birman et al. discuss the tradeoffs of suspending and non suspending reconfiguration (such as the ones provided in [17] and [18]). It is argued, that suspending services provide simpler solutions, and may be enhanced for more efficient reconfiguration decisions so that the time for reconfiguration and state transfer before reconfiguration can be reduced. It is also part a future work to provide the complete details of a framework that will allow the most modular employment of the reconfiguration scheme by applications. We now briefly suggest three applications that can benefit from our scheme.

Reconfigurable Labeling and Counter Increment Algorithms. Recall the practically-self-stabilizing labeling scheme and counter algorithm of Section 4.3. We can now adjust that solution to benefit from our reconfiguration mechanism. We let the configuration members run the service. The configuration runs the labeling algorithm and maintains a globally maximal label and counter. Whenever a new counter is needed, then a member of the configuration runs the cunter algorithm to provide one greater one. In this way, every configuration that is established can be

regarded as an instance of the static case of the labeling and counter schemes presented in Section 4.3. Since the scheme has epochs containing processor identifiers, configuration members need to take extra care when moving from one configuration to the next. In particular, the label/counter structures and labels/counters that are moved to the next configuration need to be adjusted to refer only to the new configuration membership via an interface that can possibly be generalized for any application.

Reconfigurable Virtually Synchronous State Machine Replication. Recall the practically-self-stabilizing virtual synchrony algorithm of Section 4.4. The algorithm is coordinator-based and works on the primary component given the *supportive majority* assumption on the failure detectors. This assumption states that a majority of processors of the (fixed) processor set mutually never suspect some processor on their failure detectors throughout an infinite execution, given that this processor does not crash. The proof of Section 4.4 establishes that such a supported processor eventually becomes the coordinator throughout the execution. A configuration member can become the coordinator if no coordinator exists, by using the counter algorithm in order to use the counter as an identifier for its proposed process group. The proposer with the greatest counter becomes the coordinator. It is possible to achieve reconfigurable state replication by allowing the coordinator to decide when should reconfigurations take place, and also describe how the coordinator can control joins to the computation. The transfer of state from one configuration to the next is ensured by the introduction of new processors to the system with default (null) states and by the processors synchronizing their states before the coordinator moves to call reconfiguration.

Self-stabilizing Reconfigurable Emulation of Shared Memory. Birman et al. [93] show how a virtually synchronous solution can lead to a reconfigurable emulation of shared memory. Following this approach, and using self-stabilizing reconfigurable SMR solution, and the increment counter scheme of Section 4.3, we can obtain a self-stabilizing reconfigurable emulation of shared memory. Given a conflict-free configuration, a typical two-phase read and write protocol can be used for the shared memory emulation. In the event of a delicate reconfiguration, the coordinator of the virtual synchrony algorithm suspends reads and writes on the register and once

a new configuration is established, the emulation continues. Virtual synchrony ensures that the state of the system, in this case the state of the object, is preserved. In the event of a brute force reconfiguration (e.g., due to transient faults or violation of the churn rate), the system will automatically recover and eventually reach a legal execution (in this case the state of the system may be lost).

5.7 Chapter Summary

We presented the first self-stabilizing reconfiguration scheme that recovers automatically from transient faults, such as temporary violations of the predefined churn rate or the unexpected activities of processors and communication channels, using a bounded amount of local storage and message size. We discussed how this scheme could be used for the implementation of several dynamic distributed services, such as a self-stabilizing reconfigurable virtual synchrony, which in turn can be used for developing self-stabilizing reconfigurable SMR and shared memory emulation solutions. We use a number of bootstrapping techniques for allowing the system to always recover from arbitrary transient faults, for example, when the current configuration includes no active processors. We believe that the presented techniques provide a generic blueprint for different solutions that are needed in the area of self-stabilizing high-level communication and synchronization primitives, which need to deal with processor joins and leaves as well as transient faults.

Chapter 6

Self-Stabilizing Byzantine Fault Tolerance Based on Failure Detectors

We now proceed to the last part of this thesis. The presented self-stabilizing algorithm is the first to employ failure detectors to address the task of self-stabilizing BFT replication. Already in Chapter 1 we highlighted the contributions and the challenges of this task. We now proceed with the specific system settings and the presentation of the service.

6.1 Specific System Settings and Definitions

As a reminder of Chapter 3, we consider an asynchronous message-passing system with a *fixed* set of processors (servers or replicas) P, where $|P| = n$. At most $f = (n - 1)/5$ of processors may (intentionally or not) exhibit malicious (Byzantine) behavior, i.e., fail to follow the protocol (in Section 6.6 we increase this to $f = (n - 1)/3$). We denote the set of correct processors by C. We also assume that messages reaching p_j from p_i are guaranteed to have originated from p_i, unless they are the result of a transient fault, i.e., a malicious processor p_k (where $i \neq j \neq k$) cannot impersonate p_i sending a message to p_j [52].

Complexity metric. We use the metric of an *asynchronous round* of a fair execution R as in the previous chapter, namely, it is the shortest prefix of R in which every *correct* processor p_i completed an iteration I_i, and all messages p_i sent during I_i were received.

6.2 Solution Outline

Our solution is composed of three *modules*.

View Establishment module: This is the most critical and challenging module. It is composed of two parts: A coordinator automaton and a series of predicates (Algorithm 8) called by the automaton (Algorithm 9). It establishes a consistent view (and state) among $n - f$ replicas, where n is the number of replicas and f an upper bound on the number of faulty replicas as discussed in Section 3. Managing convergence to a consistent view in the presence of Byzantine processors injecting arbitrary messages, and in the existence of other stale information in local states and communication channels is very demanding and it is impossible without a series of assumptions [48–51]. To this respect, we present an automaton-based solution where convergence requires a fragment of the computation to be free of failures (still, note that even under this constraint, view establishment is very challenging as one can infer from Section 6.3). In Section 6.6, we relax this constraint.

Replication module: The replication module (given as Algorithm 10) follows the replication scheme by Castro and Liskov [38], but adjusted to also cope with stale information. When there appears to be a common view and hence a primary, the replicas progress the replication. In case an inconsistent replica state is detected (due to stale information), then this module requests a view establishment and falls back to a default state. As we prefer to use information theoretically secure schemes, rather than computationally cryptographic secure schemes based on message signing, we require that clients contact *all* replicas. The primary is still the one to decide the order, but the replicas, through a self stabilizing all-to-all exchange procedure, validate the requests suggested to be processed by the primary; a request is *valid* if it has been seen by a strong majority of correct $((n - f)/2 + f)$ replicas (see Section 6.4).

Primary Monitoring module: The primary is monitored by a view change mechanism, employing a failure detector (FD) to decide when a primary is suspected and, thus, a view change is required (Section 6.5). In case the primary is found to impede the replication progress, it is considered faulty and the module proceeds to change the primary, by installing the next view. We propose an implementation of a self-stabilizing FD that checks both the responsiveness of the replicas (including the primary), and whether the primary is progressing the state machine. Our re-

sponsiveness FD can be seen as a self-stabilizing version of the muteness FD given in [135] but adapted to an asynchronous environment following the technique discussed in [136]; our self-stabilizing implementation follows [26].

We follow [38] in their use of views, and we employ a bounded integer counter in the domain $[0, n-1]$, so that a processor in view i considers processor p_i as the view's primary. A processor that suspects the primary p_i as faulty and sees some support for this suspicion by other processors, requests a view change to view $i+1$. Since there are only at most f faulty processors there can be at most f consecutive faulty primaries before we reach to a non-faulty one, namely in the case where the identifiers of the faulty processors are consecutive. In case where stale information (due to transient faults) is detected, the system *establishes* a view by moving to a commonly-known system-imposed incorruptible view (e.g., view 0).

The first step towards providing safety is to ensure that a view is in place and so a primary is known to progress the replication. This is provided by the View Establishment module (Algorithms 8 and 9). During this process, it is possible that a processor will adopt a view that it regards as *adoptable*, or move to request a view establishment to move to the default fallback view and replica state. When there appears to be a common view for a majority of processors, called a *serviceable* view, then processors progress the replication unless they detect an inconsistent replica state, in which case, the replication module (Algorithm 10) requires a view establishment while it moves to a fallback default state. The failure detector (Algorithm 11) suspects the primary based on its responsiveness and on whether it progresses the replication. If a sufficient number of processors suspect the primary at least once, a view change process is initiated by the view change (primary monitor) module implemented as Algorithm 12.

Thresholds. Following the approach of [11], we define important thresholds that the algorithm uses to take decisions based on $n = 5f + 1$ processors. We then suggest how these thresholds can be adjusted for the optimal $n = 3f + 1$ (cf. Section 6.6). The benefit of $n = 5f + 1$ is that it gives simpler and "cleaner" solutions, and also requires less correct processors to progress the replication. We *establish* our view with $n - f$ processor support which is the maximal support that can be demanded. As such, a view that was indeed established (and was not the result of an arbitrary initial state), must be held by at least $\max\{n - 2f, n/2\}$ correct processors. A view needs to

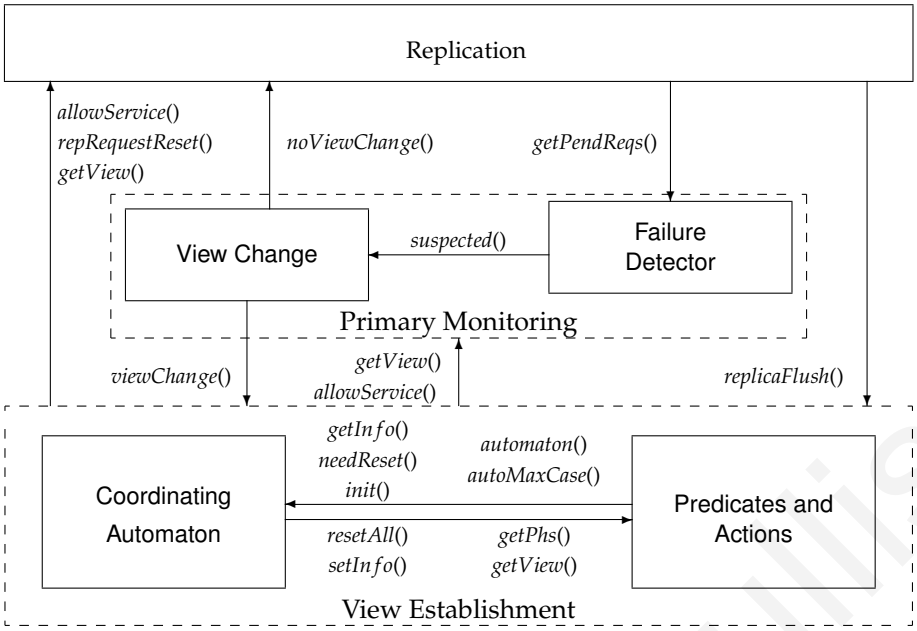

Figure 6.1: *Modules and Interface functions.* (Information flows from $\boxed{A} \rightarrow \boxed{B}$.) Within the View establishment module, the two components Coordinating Automaton, and Predicates and Actions have the following interfaces: $getInfo()$ and $setInfo()$ merely return and change the view. Function *needReset*() returns True/False on whether a reset is required, triggering one with *resetAll*(), which causes a reset of the view, the coordinating variables (via init) and the replica, via *replicaFlush*. The coordinating automaton calls upon the predicates and actions using *automaton*(•) and checks cases until it reaches the maximal case with *autoMaxCase*(). Function $getView(j)$ when called by p_j returns the current view if it can provide service, otherwise $\top$. If it is called by p_i with $i \neq j$ then it returns the last view reported by p_j to p_i. Function *allowService*() returns True if there is a serviceable view and no view establishment is taking place. In case of detected replica state corruption, the Replication module via *repRequestReset*() requests a view reset from the View Establishment module. Interface *replicaFlush*() imposes a fallback/default DEF_STATE on replication related variables when a reset of the view takes place. *getPendReqs*() returns a set of pending requests that need to be executed by the current primary, or a pending view change request that needs to be executed by the proposed primary. *suspected*() returns True if the primary is suspected. *noViewChange*() returns False when a view change is taking place. The Primary Monitoring module requests a view increment via *viewChange*() when there is sufficient evidence that the primary is not making progress.

provide the following properties: (i) Correct processors should adopt an established view, i.e., one supported by at least $n - 2f$ processors. This is an *adoptable* view. (ii) A view needs to provide safety for the replication service, so a majority of correct processors needs to have this view, i.e., we need $\lceil (n - f)/2 \rceil + f$ (*serviceability*). It is not hard to see that for $n = 5f + 1$, the two properties coincide on $3f + 1$. This is not the case for stronger f, for which the adoptability property is below serviceability (see Section 6.6.2).

Figure 6.1 depicts the modules, their components, and the interaction between

them. The View Establishment module is in Section 6.3, and the Replication module follows in Section 6.4. The Primary Monitoring in Section 6.5.2 includes both the Failure Detector and the View Change protocol. We conclude with discussion on optimizations.

6.3 View Establishment

This module provides a unique view to the correct processors, and conducts view changes upon the instruction of the primary monitoring module (Section 6.5.2). We start with the algorithm description and continue with the correctness proof.

6.3.1 Algorithm Description

Overview. The module is implemented as a Coordinating Automaton (Algorithm 8) and View Predicates and Actions (Algorithm 9). Algorithm 8 defines a two-state (or *phase*) automaton and imposes a lockstep transition of phases/views among at least $3f + 1$ correct processors in a unison manner. It does so by employing a witnessing mechanism by which a processor p_i may only proceed to a view or phase change if the following conditions hold:

Cond 1: p_i's view and phase were reported back by $4f + 1$ other processors.

Cond 2: p_i has reported back to the other processors that ***Cond 1*** is satisfied.

Cond 3: at least $4f + 1$ processors acknowledge that ***Cond 2*** was reported true by p_i.

Upon finding such a set of witnesses, p_i may proceed to perform an action and change to one of the two phases.

Phase/view transitions are rigorously defined as a series of automaton predicates and corresponding actions that can be executed only when a processor's local (view and phase-related) state is witnessed. These are seen in Algorithm 9. We depict the phase coordinating automaton of Algorithm 8 in Figure 6.2, and the view establishment predicates and actions of Algorithm 9 are presented and explained in Table 6.1.

In a nutshell, Phase 0 is a monitoring phase that checks for view change requests, view conflicts or replica state conflicts. The latter are detected by the Replication module. Upon finding a conflict or seeing a view change instruction, and given that

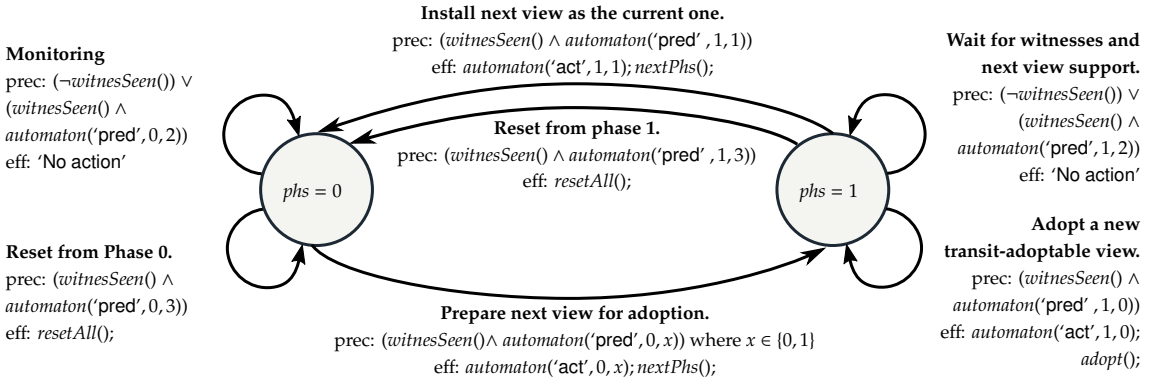

Figure 6.2: The view establishment coordinating automaton (Algorithm 8) for processor $p_i \in P$. The automaton(type, phase, case) function assigns different actions to different predicates *and* actions *per phase. Each predicate with the corresponding action is formally defined in Algorithm 9 (lines 9–26) and also appears in Table 6.1.*

Phase,Case	Predicates	Action
0, 0	$\exists v \in \{views_i[j]\}_{p_j \in P} : transitAdoble(j, 0, \text{'Follow'}) \wedge (v_i.cur \neq v.cur)$ (A view pair that was found to be adoptable is not p_i's view.)	$adopt_i(v)$ † (Adopt this view.)
0, 1	$vChange_i \wedge establishable_i(0, \text{'Follow'})$ (View change instructed by view primary monitoring.)	$nextView_i()$; † (Increment view.)
0, 2	$transitAdoble_i(i, 0, \text{'Remain'}) \vee (vp_i = \text{RST\_PAIR})$ (Monitoring)	Return 'No action'
0, 3	No transit adoptable view.	$resetAll_i()$;
1, 0	$\exists v \in \{views_i[j]\}_{p_j \in P} : transitAdoble(j, 1, \text{'Follow'}) \wedge (v_i.cur \neq v'.cur)$ (A view pair that was found to be adoptable and does not match p_i's *next* view.)	$adopt_i(v)$; (Adopt this view.)
1, 1	$establishable_i(vp_i, 1, \text{'Follow'})$ (The view intended to be installed appears establishable.)	$establish_i()$; † (Apply changes to view.)
1, 2	$transitAdoble(i, 1, \text{'Remain'})$ (Waiting to gather support for $vp_i.next$.)	Return 'No action'
1, 3	No transit adoptable view.	$resetAll_i()$;
For all actions where $automaton(\text{'act'}, \bullet, \bullet) \neq \text{'No action'}$, p_i also performs a $resetVchange_i()$.		
† These actions are also escorted with a call to *nextPhs*(). (Reset actions always move the phase to 0.)		

Table 6.1: The predicates for view transitions by case and phase.

this knowledge is known to the other processors through the witnessing mechanism that will be described next, the automaton moves to Phase 1 whilst taking appropriate action. At this point it is preparing to install a new phase and view. In the case of view change, its current is incremented by one, or in the case of conflicts, it is set to the default view. Again, upon seeing that this preparing of view change is witnessed and if this is justified by the views reported by the other processors, the algorithm installs the new view and moves back to the monitoring Phase 0. Replication is only possible when a majority of correct processors (i.e., $3f + 1$) with the same view is in Phase 0.

Local variables and information exchange. The module uses the variable type v*Pair*, which is a pair of *views* $\langle cur, next \rangle$, where each view is an integer in $\{0, 1, \ldots, n-1\} \cup \{\perp\}$. Processors have a common hard-coded fallback view DF_VIEW (say 0) and a reset *v*Pair RST_PAIR $= \langle \perp,$ DF_VIEW$\rangle$ used in case of corruption. Processor p_i maintains an array of *vPair*s called $views_i[n]$. The field $views_i[i]$ is p_i's current *vPair*. Specifically, $views_i[i].cur$ (with alias vp_i) is p_i's current view and *next* only differs to *cur* if a view transition is taking place, i.e., the automaton is in Phase 1, and awaits for witnesses to install $views_i[i].next$ as its current view. For regular view transitions triggered by the primary monitoring module, a processor at Phase 1 should have $vp_i.next = vp_i.cur+1$. Via the propagation mechanism of Algorithm 8, line 21, p_i sends $views_i[i]$ to every $p_j \in P$. Correspondingly, when p_i, receives a copy of $views_j[j]$ from p_j (Algorithm 8, line 22), it stores p_j's *vPair* in $views_i[j]$. When the boolean $vChange_i$ is True, this records a request by the primary monitoring mechanism of p_i to increment the view.

The integer phase array $phs[n]$ contains fields with values in $\{0, 1\}$ indicating automaton Phases 0 or 1. Field $phs_i[i]$ is p_i's phase and $phs_i[j]$ is the last reported phase by p_j acquired via the propagation mechanism. Algorithm 8 uses $echo_i[j]$ as an alias of the triple $\langle phs_i[j], witnesses_i[j], views_i[j] \rangle$. The boolean array $witnesses_i[n]$ stores $witnesses_i[j] =$ True if p_j has sent a message $m_{j,i}.\langle \bullet, (phs^i, witnesses^i, views^i) \rangle$ to p_i, such that $phs^i = phs_i[i]$ and $view^i = views_i[i]$. If $m_{j,i}$ also satisfies $witnesses^i = witnesses_i[i] =$ True, then p_j is added to the *witnesSet* set.

Detailed description of the View Establishment Coordinating Automaton (Algorithm 8). This algorithm essentially runs (or "coordinates") the automaton and is responsible for the propagation and receipt of information. It imposes a lockstep movement of processors through the automaton to ensure that they act on "informed" decisions, and that all correct processors can proceed to the next phase/view. To this end, it uses the following macros and functions.

- Macro $echoNoWitn_i(k)$ (line 6) checks whether the view and phase that processor p_k reported about p_i (i.e., the echo alias $echo_i[k]$), match p_i's view and phase.
- Macro $witnesSeen_i()$ (line 7) is correct when the $witnesSet_i$ set discussed before (and also including p_i), is of size greater than $4f$.
- Macro *nextPhs*() (line 8) proceeds the phase from 0 to 1 and from 1 to 0, also emptying the *witnesSet* set.

Algorithm 8: Self-stabilizing View Establishment: Coordinating Automaton; code for processor p_i

1 **Variables:** $phs[i]$ is an array of phases in $\{0,1\}$ where $phs[i]$ is p_i's phase and $phs[j]$ is p_j's last reported phase.
2 $witnesses[n]$ is an array of Booleans, where $witnesses[i]$ refers to the case where p_i observes that $4f+1$ processors had noticed the most recent value of $getInfo_i(i)$ (that returns $views[i]$) and $phs[i]$.
3 $witnesSet$ is a set of processors p_k for which p_i received $witnesses[k] = \mathsf{True}$.
4 $echo[n]$ is an array where $echo[i]$ is an alias to $(views[i], phs[i], witnesses[i])$ and $echo[j]$ is the most recent values that p_i received from p_j after p_j responded to p_i with the most recent values it received from p_i.
5 **Alias.** The processor set $P.w_i$ is an alias to $X \subseteq P : \{p_j \in witnesSet_i : (echo_i[i] = echo_i[j])\} \cup \{p_i\}$.
6 **Macros:** $echoNoWitn(k)$ = **return** $(\langle vp_i, phs[i]\rangle = \langle echo[k].views, echo[k].phs\rangle)$
7 $witnesSeen()$ = **return** $(witnesses[i] \wedge (|P.w_i| \geq 4f+1))$
8 $nextPhs()$ = $\{\langle phs[i], witnesses[i], witnesSet\rangle \leftarrow \langle (phs[i]+1 \mod 2), \mathsf{False}, \emptyset\rangle;\}$
9 **Interface functions:** $getPhs(k)$ = **return** $phs[k]$;
10 $init()$ = $\{witnesSet \leftarrow \emptyset$; **foreach** $p_j \in P$ **do** $\langle phs[j], witnesses[j]\rangle \leftarrow \langle 0, \mathsf{False}\rangle\}$;
11 **do forever begin**
12 **if** $Alg9.needReset()$ **then** $Alg9.resetAll()$;
13 $witnesses[i] \leftarrow (|\{p_j \in P : echoNoWitn(j)\}| \geq 4f+1)$;
14 $witnesSet \leftarrow witnesSet \cup \{p_j \in P : witnesses[j]\})\}$;
15 **if** $witnesSeen()$ **then**
16 **let** $case = 0$;
17 **while** $(\neg Alg9.automaton(\text{'pred'}, phs[i], case) \vee (Alg9.autoMaxCase(phs[i]) \geq case)$ **do** $case \leftarrow case + 1$;
18 **if** $(Alg9.autoMaxCase(phs[i]) \geq case)$ **then**
19 **let** $ret = Alg9.automaton(\text{'act'}, phs[i], case)$; `// Automaton executed`
20 **if** $(ret \notin \{\text{'No action'}, \text{'Reset'}\})$ **then** $nextPhs()$;
21 **foreach** $p_j \in P$ **do send** $\langle phs[i], witnesses[i], Alg9.getInfo(i), (phs[j], witnesses[j], Alg9.getInfo(j))\rangle$;
22 **upon receive** m **from** p_j **do if** $(valid(m,j))$ **then** $(phs[j], witnesses[j], echo[j]) \leftarrow m.(p,w,e)$; $Alg9.setInfo(m,j)$;

- The interface function $getPhs(k)$ (line 9) returns $phs_i[k]$ if called by p_i.
- The interface function $init()$ (line 10) resets the variables of the coordinating automaton algorithm to default values.

Line-by-line explanation. The algorithm begins with a stale information check (line 12) that can trigger a reset if local stale information is found. A processor p_i running the algorithm checks whether p_i is witnessed sufficiently, and raises its $witnesses_i[i]$ flag if at least $4f+1$ processors acknowledge its view and phase when each p_j of them satisfies $echoNoAll_i(j)$. A processor p_j is called a *witness* and is added to the set $witnesSet_i$ (line 14) if it notifies p_i that its witnesses flag $witnesses_j[j]$ is

1 **Variables:** The array $views[n]$ stores an array of n vPairs. Field $views[i]$ stores p_i's view and $views[j]$ where $j \neq i$ stores the latest view reported by processor p_j about itself. Alias $vp_j = views_i[j]$ and $phs(j) = Alg8.getPhs(j)$. Variable $vChange$ is a boolean. The type $mode \in \{$'Remain', 'Follow'$\}$.

2 **Constants:** DF_VIEW a default/fallback hardwired $vPair$. A reset vPair RST_PAIR $= \langle \perp,$ DF_VIEW$\rangle$.

3 **Macros:**

4 $staleV(k)$ = **return** $((Alg8.get[k] = 0 \wedge \neg legitPhsZero(views[k])) \vee (phs(j) = 1 \wedge \neg legitPhsOne(views[k])))$

5 $legitPhsZero($v$Pair\ vp)$ = **return** $(((vp.cur = vp.next) \vee (vp =$ RST_PAIR$)) \wedge typeCheck(vp))$

6 $legitPhsOne($v$Pair\ vp)$ = **return** $((vp.cur \neq vp.next) \wedge typeCheck(vp))$

7 $typeCheck($v$Pair\ vp)$ = **return** $((\forall x \in vp.\langle cur, next\rangle : x \in [0, n-1] \cup \{\perp\}) \wedge (vp.next \neq \perp))$

8 $valid(m, k)$ = **return** $((m.phs = 0 \wedge \neg legitPhsZero(m.views[k])) \vee (m.phs = 1 \wedge \neg legitPhsOne(m.views[k])))$

9 $sameVSet(int\ j, phase\ \phi) = \{p_k \in P : (phs(k) = \phi) \wedge (vp_k = vp_j) \wedge \neg staleV(k)\}$

10 $transitAdopble(int\ j, phase\ \phi, mode\ d)$ = **return**$(|sameVSet(vp_j, phs(j)) \cup transitSet(j, \phi, d)| \geq 3f + 1)$

11 $transitSet(int\ j, phase\ \phi, mode\ t) = \{p_k \in P : (phs(k) \neq \phi) \wedge transitCases(int\ j, vp_k, \phi, t) \wedge \neg staleV(k)\}$

12 $transitionCases(int\ j, vPair\ vp, phase\ \phi, mode\ t) = \{$**select**$(\phi, t)$**:**

13 **case** $(\bullet,$'Remain'$)$ = **return** $(vp.next = vp_j.cur)$

14 **case** $(0,$'Follow'$)$ = **return** $(vp.next = vp_j.cur + 1 \mod n)$

15 **case** $(1,$'Follow'$)$ = **return** $(vp.cur = vp_j.next);\}$

16 $adopt($v$Pair\ vp) = \{vp_i.next \leftarrow vp.cur\rangle\};$

17 $establishable(phase\ \phi, mode\ d)$ = **return**$(|sameVSet(vp_i, phs(i))| + |transitSet(\phi, d)| \geq 4f + 1)$

18 $establish() = \{vp_i.cur \leftarrow vp_i.next\};$

19 $nextView() = \{views[i].next \leftarrow (views[i].cur + 1 \mod n)\};$

20 $resetVchange() = \{vChange \leftarrow$ False$;\};$

Figure 6.3: Variables and Macros for the View Establishment Algorithm; code for p_i.

True. Finally, p_i satisfies the macro $witnesSeen()$ if every p_j in a $4f + 1$-strong set of witnesses also satisfy $echo_i[i] = echo_i[i]$, namely they have acknowledged p_i's view, phase and p_i's witnessing flag. Note that when the phase or view change, the witnessing procedure starts all over, requiring several algorithm iterations, and as will be proved later $O(1)$ asynchronous rounds.

Since $witnesSeen()$ (line 15) is the condition to allow the view change automaton to proceed, we note that we later prove that the fact that $witnesSeen_i()$ holds for p_i cannot stop $witnesSeen_j()$ from holding as well, and that one processor cannot move phases without waiting for the other processors. Lines 16 and 17 iterate through case, until the first case holds. This case is executed and if this not a non-action

Algorithm 9: Self-stabilizing View Establishment: View Predicates and Actions; code for processor p_i)

```
1  Interface functions:
2  function needReset() = return (staleV(i) ∨ Alg10.replicaFlush())
3  function resetAll() = {views_i[i] ← RST_PAIR; Alg8.init(); Alg10.repRequestReset(); return
   ('Reset');}
4  function viewChange() = {vChange ← True};
5  function getView(j) begin
6      if (j = i) ∧ (phs(i) = 0 ∧ witnesSeen()) then (if (allowService()) then return (vp_i.cur));
7      else return (views[j].cur);
8  function allowService() = return
   (((|sameVSet(vp_i, *)| ≥ 3f + 1) ∧ (phs[i] = 0 ∧ vp_i.cur = vp_i.next)))
9  function automaton(type, phase, case) begin
10     select(type, phase, case):
11         case ('pred', 0, 0) = return (∃vp ∈ {views_i[j]}_{p_j∈P} : transitAdoble(j, 0,
           'Follow') ∧ (vp_i.cur ≠ vp.cur))
12         case ('act', 0, 0) = {adopt(vp); resetVchange();}
13         case ('pred', 0, 1) = return (vChange ∧ establishable(0,'Follow'))
14         case ('act', 0, 1) = {nextView(); resetVchange();}
15         case ('pred', 0, 2) return (transitAdoble(i, 0,'Remain') ∨ vp_i = RST_PAIR))
16         case ('act', 0, 2) = return ('No action')
17         case ('pred', 0, 3) = return (True)
18         case ('act', 0, 3) = {resetAll(); resetVchange();}
19         case ('pred', 1, 0) = return (∃vp ∈ {views_i[j]}_{p_j∈P} : transitAdoble(j, 1,'Follow') ∧
           (vp_i.next ≠ vp.cur))
20         case ('act', 1, 0) = {adopt(vp'); resetVchange();}
21         case ('pred', 1, 1) = return (establishable(vp_i, 1,'Follow'))
22         case ('act', 1, 1) = {{if vp_i = RST_PAIR then Alg10.replicaFlush();}
           establish(); resetVchange();}
23         case ('pred', 1, 2) = return (transitAdoble(i, 1,'Remain'))
24         case ('act', 1, 2) = return ('No action')
25         case ('pred', 1, 3) = return (True)
26         case ('act', 1, 3) = {resetAll(); resetVchange();}
27 function autoMaxCase(phase) = {select(phase) case 0: return 3; case 1: return 3;}
28 function getInfo(k) = return (views[k]);
29 function setInfo(x, j) = {views[j] ← x.views)};
```

predicate the phase is incremented (lines 18–20). The propagation of information and its receipt is done via lines 21 and 22.

Detailed description of the View Establishment algorithm (Algorithm 9). This part of the module is composed of a series of macros and functions as follows:

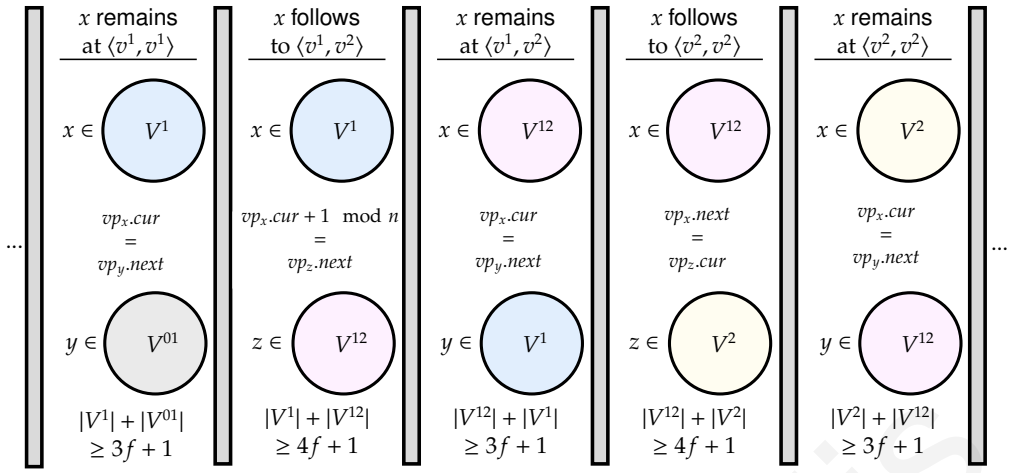

Figure 6.4: A change of view on the perspective of a processor p_x from view v_1 to view v_2. When $p_x \in V^1$ this implies that p_x has view v_1 and V^1 is the set of processors that have reported to p_x that they have v_1. Set V^{12} is the set of processors have a view pair $\langle v_1, v_2 \rangle$ and are thus transiting to v_2. The gray bars indicate the time required for witnesSeen$_x$() (Alg. 8, line 15) to hold in order for p_x to execute the automaton and perform the actions seen in the figure. The figure depicts how transit sets work to ensure that a change to the view pair never leads to a reset, since there are enough processors to support a remain *to this view pair, but allows for processors to* follow *to the next view once they see sufficient support.*

- Macros of lines 4–7 check for stale information in the view pairs and view pair structures.

- Macro *valid*(m, k) (line 8) checks whether message m by p_k is valid and not stale in its structure (though it might conform with the structure we are not guaranteed it is not stale, i.e., it might come from an arbitrary initial state of the communication links).

- Macro *sameVSet*(*vPair vp*, *phase* ϕ) (line 9) returns a set of processors that have *vp* as their current non-stale view and ϕ as their phase.

- Macro *transitSet*(*phase* ϕ, *mode* $\in$ {'Remain', 'Follow'}) (line 11) is called with 'Remain' when the processor wants to get the set that would support the lowest threshold to remain in its current view. If no such support is found then this will lead to a reset. The macro is also called with 'Follow' in order to get the set that could support following other processors to a new view or to a view change depending on the phase. To this end it employs the macro *transitionCases*() to give a per case response (cf. Figure 6.4).

- Macro *transitAdopble*(*phase* ϕ, *mode*) (line 10) takes the combined cardinalities of size of *sameVSet*() and *transitSet*() to determine whether the set of processors

that report to be transiting to view vp amount to more than $3f+1$ (cf. Figure 6.4). In this case it is considered safe to remain to the current view.

- Macro *adopt*(vp) (line 16) performs the actual assignment of vp. E.g., for processor p_i, it assigns $views_i[i] \leftarrow vp$.

- Macro *establishable*(*phase* ϕ, *vPair* vp) (line 17) checks whether the $4f+1$ threshold to *move* to a view change (phase 0) or to a new view (phase 1) is reached. It employs *transitSet*() and *transitCases*() as explained above, but with the 'Follow' mode.

- If $establishable_i()$ is satisfied when p_i is in phase 1, then $establish_i()$ installs the new view (line 18). If $establishable_i()$ is satisfied in phase 0, then macro $nextView_i()$ (line 19) increments the $vp_i.next$ and proceeds to phase 1.

- Macro *resetVchange*() (line 20) resets the flag variable *vChange* to False once the request by the Primary Monitoring module has been considered and has triggered a view change.

- Function *needReset*() returns True if *staleV*() = True or the replication algorithm requires a reset.

- Function *resetAll*() (line 3) calls for a complete reset of all modules when the view is found in an unanticipated state.

- Function *getView*() (line 5), when called with $getView_i(i)$, returns p_i's current view if this is serviceable, otherwise it returns $\top$. If is called by p_i as $getView_i(j)$ for $j \neq i$, then it returns the last view reported by p_j to p_i, i.e., $views_i[j]$.

- The *automaton*() function (line 9) is analyzed thoroughly in Table 6.1.

- Function *autoMax*(k) (line 27) returns the maximal number of case that the automaton predicates have depending on the phase k.

- Function $getInfo(k)$ (line 28) returns view-related information about processor p_k. Specifically it returns $views_i[k]$. Its setter counterpart is $setInfo(m_{k,i}, k)$ (line 29) that extracts the view from a message m coming from p_k and updates $views_i[k]$ with it.

6.3.2 Correctness

We proceed with the correctness proof by introducing the required definitions and notation.

Executions. A fair execution R is *mal-admissible* when, throughout R, the set of $f = n/5 + 1$ malicious processors remains the same. A fair execution is *mal-free* if throughout the execution, every malicious processor behaves as a correct one. A *consistency set* Σ is a subset $\Sigma \subseteq P : |\Sigma| \geq 4f+1$ such that every processor in Σ executes the algorithm as a correct processor (as defined in Section 3). If the processors of Σ do not change their phase throughout R, we call R a *phase-fixed* execution. If processors of Σ do not change any field of their view pair throughout R, we call R a *view-fixed* execution. Recall that C is the set of correct processors (see Section 6.1).

Threat. We denote by $L_{j,i}$ the set of messages in the communication link from processor p_j to p_i. The messages are either added by p_j or may be the result of an arbitrary initial state and the message number respects the link capacity. Given a system state c in a fair execution R, a message $m \in L_{j,i}$ is a *threat* to processor $p_i \in C$ at c if predicate $threat(m, i)$ holds, such that $threat(m, i) = \mathsf{True} \iff (\exists v \in \{views_i[\ell]\}_{p_\ell \in P} : (v = views_i[i] = views_i[j]) \wedge transAdopble_i(views_i[i]) \wedge (valid_i(m, j)) \wedge ((m.views[j].cur \neq v.cur) \vee (m.phs[j] \neq 0)))$. In other words, if p_i has view v (i.e., it is adoptable) and relies on p_j to make v adoptable locally, and there exists a message in $L_{j,i}$ stating that p_j does not support v, then p_i's view is in jeopardy.

Stable view. A stable view is a view that malicious processors cannot overthrow. Thus, a stable view implies convergence, and characterizes an execution composed of only a legal system state. Given a state c in a mal-admissible execution R, the *supporters' set* of view pair $vp = \langle v, v \rangle$ is denoted by $V^{sup}(vp)$, such that

$$V^{sup}(vp) = \{p_i \in C : (phs_i[i] = 0) \wedge (views_i[i] = vp) \wedge \neg staleV_i(i)\}.$$

We say that a view v of view pair $vp = \langle v, v \rangle$ is a *stable view* if $\exists X \subseteq V^{sup}(vp)$ such that the following properties hold:

1. $|X| \geq 3f + 1$,
2. $\forall p_i \in X \; ((X \subseteq witnesSet_i) \wedge (|\{p_j \in X : phs_i[j] = 0 \wedge \neg staleV_i(j)\}| \geq 3f + 1))$,
3. $\forall p_i, p_j \in X \; (\nexists m \in L_{j,i} : (threat(m, i))$.

We refer to the X with the maximal number of processors as the *strong* support set for stable view v and write $V^{ssup}(v)$.

View reset. A *view reset*, or simply a *reset* is a call to *resetAll*() (Alg. 8, line 3). This has the effect of setting variables to default values and also returning the phase of the coordinating automaton to 0.

Task description. The view establishment task $\mathcal{VE}$ includes all the system states of mal-admissible executions in which there is a stable view, or where a stable view is followed by a view change leading to a new stable view.

Proof Outline. We establish the correctness of the View Establishment module by first proving the convergence from an arbitrary state where stale information may exist and a stable view may be absent to a state with stable view. En route to prove the convergence theorem (Theorem 6.3.10), we first show that stale information that is locally detectable and messages with detectable stale content are removed within $O(1)$ asynchronous rounds (Claim 6.3.1 and Corollary 6.3.2). We then continue to prove a series of invariants that hold for a view-fixed execution and prove that the automaton guarantees that processors that are correct, exchange their local information within $O(1)$ asynchronous rounds (Lemma 6.3.4).

Lemma 6.3.8 carries on to prove that, within $O(n)$ asynchronous rounds, the module brings the correct processors to a common view and phase, and any subsequent change to these is performed in lockstep by $4f + 1$ processors of Σ. Lemma 6.3.9 shows that a single malicious adversary's activity in combination with an arbitrary initial state can block the system from converging to a stable view. To overcome this obstacle, we assume for the convergence theorem (Theorem 6.3.10) that we have mal-free[1] executions. Theorem 6.3.11 proves closure, i.e., if there was convergence to a stable view, then within $O(1)$ asynchronous rounds of a mal-admissible execution the system remains to a stable view even if there are view changes (instructed by the Primary Monitoring module). We note that we do the proof for an execution R that is a mal-admissible one, and in which there is a consistency set Σ. If a result diverges from this rule, it is clearly stated.

Lemma 6.3.1. *Let R be a mal-admissible execution R of the View Establishment module starting in an arbitrary system state. Within $O(1)$ asynchronous rounds, it holds that* $\forall p_i \in C$ $(staleV_i(i) = \mathsf{False})$.

[1] We revisit this assumption in Section 6.6

Proof. Consider processor $p_i \in C$ that performs at least one complete iteration of Algorithm 8. This includes an execution of line 12. This is a call to $Alg9.needReset_i()$. If $Alg9.staleV_i(i) = \mathsf{True} \Rightarrow Alg9.needReset_i() = \mathsf{True}$ that leads to an execution of $Alg8.resetAll_i()$. This is an assignment of $views_i[i] \leftarrow \mathsf{RST\_PAIR}$. Since $\mathsf{RST\_PAIR}$ is never stale (by $legitPhsZero(\mathsf{RST\_PAIR}) = \mathsf{True}$), we proceed to show that p_i never introduces stale information back to its state.

If $Alg9.staleV_i(i) = \mathsf{False}$, by examination Algorithm 9, p_i assigns a value to $views_i[i]$ only via *adopt*() (Fig. 6.2, line 16), *establish*() (Fig. 9, line 18) and *nextView*() (Fig. 9, line 19), i.e., automaton actions for cases $(0,0)$, $\langle 1,0\rangle$ and $\langle 0,1\rangle$ correspondingly. Since both $adopt_i(v)$ and $establish_i(v)$ imply that a view v adopted never satisfies *staleV*() (the check inside $sameVset(\bullet)$ and $transitSet(\bullet)$), it is not possible to install a stale view. Macro $nextView_i()$ merely increments the $view_i[i].next$ that was not stale, since in a complete iteration, the passing from line 12 will have imposed a $\mathsf{RST\_PAIR}$ for $views_i[i]$. □

Since every correct processor takes the above step to clean their local stale information before performing their send actions (Alg. 22, line 21), we have the following side result of Lemma 6.3.1 given as a corollary.

Corollary 6.3.2. *Within $O(1)$ asynchronous rounds of a mal-admissible execution R of the View Establishment module, it holds that $\forall p_j \in P$ and $\forall p_k \in C$ any message m added by p_k to $L_{k,j}$, never encodes $valid_j(m,k) = \mathsf{True}$.*

Claim 6.3.3. *Consider an execution R of View Establishment module where Corollary 6.3.2 holds. Then:*
(i) when R does not include a view reset it holds that: R view-fixed $\iff$ *R* phase-fixed,
(ii) otherwise it holds that: R view-fixed $\implies$ *R* phase-fixed, but *R* phase-fixed $\not\Rightarrow$ *R* view-fixed.

Proof. By the cases of Alg. 9, function *automaton*(), we see that the cases of predicates with an action returning 'No action' incur no changes to phase or view. The predicates that have a corresponding action that is not a call to *resetAll*(), imply both a view change and a phase change. This proves (i). On the other hand, predicates with corresponding actions including a call to *resetAll*(), imply that the phase returns to 0. Thus, if the phase is already at 0, then the view pair changes to $\mathsf{RST\_PAIR}$ but the view remains 0. This proves (ii). □

Lemma 6.3.4. *Let R be a view-fixed execution of Algorithm 8 with a consistency set Σ. Within $O(1)$ asynchronous rounds, the system either calls a reset, or it reaches a state $c^* \in R$ in which invariants (1) to (5) hold.*

(1) *Within $O(1)$ asynchronous rounds, the system reaches a state $c' \in R$ in which for any processors $p_i, p_j \in \Sigma$, it holds that $(phs_i[i] = phs_j[i]) \wedge (views_i[i] = views_j[i])$.*

(2) *Suppose that invariant (1) holds in every system state of R. Within $O(1)$ asynchronous rounds, the system reaches a state $c' \in R$ in which for any processors $p_i, p_j \in \Sigma$, it holds that $(echo_i[j].phs = phs_i[i]) \wedge (echo_i[j].views = views_i[i])$.*

(3) *Suppose that invariants (1) and (2) hold in every system state of R. Within $O(1)$ asynchronous rounds, the system reaches a state $c' \in R$ in which for any processor $p_i \in \Sigma$, it holds that $witnesses_i[i] = \mathsf{True}$.*

(4) *Suppose that invariants (1) to (3) hold in every system state of R. Within $O(1)$ asynchronous rounds, the system reaches a state $c' \in R$ in which for any processors $p_i, p_j \in \Sigma$, it holds that $witnesses_j[i] = \mathsf{True}$ and $p_i \in witnesSet_j$.*

(5) *Suppose that invariants (1) to (4) hold in every system state of R. Within $O(1)$ asynchronous rounds, the system reaches a state $c' \in R$, in which it holds that $echo_i[i] = echo_i[j]) = \mathsf{True}$ and $witnesSeen_i() = \mathsf{True}$ in R.*

Proof. Suppose that a reset does not take place during R. We prove invariants (1) to (5) hold within $O(1)$ asynchronous rounds.

(1) By the repeated propagation of $m_{i,j} = \langle phs_i[i], \bullet, \bullet, views_i[i] \rangle$ (line 21), within $O(1)$ asynchronous rounds $p_j \in \Sigma$ receives and stores $m_{i,j}.phs$ in $phs_j[i]$ and $m_{i,j}.views$ in $views_j[i]$ (line 22 and the assumption that messages that are sent infinitely often are received infinitely often). Thus, $(phs_i[i] = phs_j[i]) \wedge (views_i[i] = views_j[i])$ and this completes the proof for invariant (1).

(2) By similar arguments as for invariant (1), we argue that by invariant (1), processor p_j sends $m_{j,i} = \langle \bullet, (phs_j[i], \bullet, getInfo_j(i) = views_j[i]) \rangle$ to processor p_i infinitely often, p_i receives $m_{j,i}$ infinitely often. Thus, within $O(1)$ asynchronous rounds, p_i stores $m_{j,i}$ in $echo_i[j]$ (and does so in every subsequent state in R after the first assignment), such that $echo_i[j] = (phs_i[j], \bullet, views_i[j]) = m_{j,i}.(phs_j[i], \bullet, views_i[j]) = (phs_i[i], views_i[i])$, hence the result. Note that this invariant directly implies that $echoNoAll_i(j) = \mathsf{True}$.

(3) By the last remark in the proof of invariant (2), within $O(1)$ asynchronous rounds, there are $4f+1$ processors $p_k \in \Sigma$ for which $echoNoWitn_i(k) = \mathsf{True}$. Thus the condition $(|p_k \in P : echoNoWitn_i(k)| \geq 4f+1)$ is satisfied and $witnesses_i[i] = \mathsf{True}$.

(4) By similar arguments as for invariant (1) applied to the $witnesses_i[i]$ field, $witnesses_i[i] = \mathsf{True} = witnesses_j[i]$ holds within $O(1)$ asynchronous rounds. This implies that when p_j next executes line 14, $p_i \in P : witnesses_j[i]$ is satisfied and p_i is added to $witnesSet_j$, and thus invariant (4) is satisfied.

(5) From the arguments for invariants (1) and (2) applied to $witnesses_i[i]$ rather than $phs[i]$ and $views[i]$, within $O(1)$ it holds that $echo_i[i] = echo_i[j]$. By applying invariants (1) – (4) to p_j, and since both $p_i, p_j \in \Sigma$, it holds that $p_j \in witnesSet_i$. Since $witnesses_i[i] = \mathsf{True}$ and since all $p_j \in \Sigma \setminus \{p_i\}$ satisfy the conditions $p_j \in witnesSet_i$ and $echo_i[i] = echo_i[j]$, and by the fact that $|\Sigma| \geq 4f+1$, $witnesSeen_i() = \mathsf{True}$ (line 7) within $O(1)$ asynchronous rounds. Hence the result. □

Remark: Each processor $p_j \in \Sigma$ that does not reset, changes the values of ($phs_j[j]$, $witnesses_j[j]$, $views_j[j]$) from a view v to v', following four consecutive transition states: (i) $(0, \mathsf{True}, \langle v, v\rangle)$, (ii) $(1, \mathsf{False}, \langle v, v'\rangle)$, (iii) $(1, \mathsf{True}, \langle v, v'\rangle)$ (iv) $(0, \mathsf{False}, \langle v', v'\rangle)$. This four-stage transition requires the execution of Alg. 8, line 18. In particular, to go from (i) to (ii), the $automaton_j(\text{'}pred\text{'}, 0, case)$ needs to hold for cases either 1 or 2. To go from (iii) to (iv), $automaton_j(\text{'}pred\text{'}, 1, 0)$ needs to hold. Both cases call $nextPhs()$ that causes $witnesses_j[j]$ to become False. For transitions (ii) to (iii) and (iv) to (i), Algorithm 8 is responsible to assign True to $all_j[j]$.

Claim 6.3.5. *Suppose that during the first $O(1)$ asynchronous rounds of R, it holds that no $p_i \in \Sigma$ changes ($phs_i[i]$, $witnesses_i[i]$, $views_i[i]$). Then either, the system takes a step with a call to reset, or the invariants of Lemma 6.3.4 hold.*

Proof. The claim defines a view-fixed execution. Thus there is either a call to reset, or the invariants of Lemma 6.3.4 hold. □

Claim 6.3.6. *Suppose that during R, it holds that $p_i \in \Sigma$ changes ($phs_i[i], witnesses_i[i], views_i[i]$) at most once, while all other $p_k \in \Sigma : k \neq i$ do not change ($phs_k[i], witnesses_k[i], views_k[i]$). Within $O(1)$ asynchronous rounds the system either, takes a step with a call to reset, or the invariants of Lemma 6.3.4 hold.*

Proof. If there is not a call to a reset then by Claim 6.3.3 the conditions of the claim imply a view fixed execution. This implies that the invariants (1) to (5) can hold.

Suppose towards a contradiction that some $p_k \neq p_i$ takes a step a_k changing $(phs_k[j], witnesses_k[j], views_k[j])$, *after* $O(1)$ asynchronous rounds from the starting state of R. If this change is due to a reset, then the proof is done. Assume this is not due to a reset. Then it must be that the invariants of Lemma 6.3.4 hold. By line 15, a_k occurs within $O(1)$ asynchronous rounds from the starting state of R. This, though, contradicts our assumption about a_k, and hence the result. □

Claim 6.3.7. *Suppose that during R each processor $p_i \in \Sigma$ changes $(phs_i[i], witnesses_i[i], views_i[i])$ at most x times. Within $O(xn)$ asynchronous rounds the system either, takes a step with a call to reset, or the invariants of Lemma 6.3.4 hold.*

Proof. Without loss of generality we assume that Σ remains the same throughout R. We build the proof inductively, by first proving the result for $x = 1$. For this case, let (again without loss of generality) p_{k_1} be the first processor that changes $(phs, witnesses, views)$, say in step $a_{k_1} \in R_1^1$, where $R = R_1^1 \circ R_2^1 \circ \cdots$. Step a_{k_1} occurs within $O(1)$ asynchronous rounds (Claim 6.3.6). Let suffix R_2^2 start immediately after a_{k_1}, and in this, processor p_{k_2} makes a step a_{k_2}. This again takes $O(1)$ asynchronous rounds from the initial state of R_2^1. In this vein we continue to construct the execution R such that $R = R_1^1 \circ R_2^1 \circ \cdots \circ R_n^1$, such that p_{k_n} takes step a_{k_n}, again within $O(1)$ asynchronous rounds. Thus, for $x = 1$ require $O(n)$ asynchronous rounds. Intuitively, we build the cases where $x > 1$ by appending to R_n a series of suffixes $R_1^2 \circ \cdots \circ R_n^2$ in which each processor takes a step with its second. Thus, inductively, execution $R = R_1^1 \circ \cdots \circ R_n^x$ within $O(xn)$ asynchronous rounds. □

Lemma 6.3.8. *Let R be an execution of Algorithm 8 and Σ the consistency set. Suppose that there exists a processor $p_j \in \Sigma$ that changes $(phs_j[j], witnesses_j[j], views_j[j])$ more than 9 times. In this case, within $O(n)$ asynchronous rounds during R, the system either calls a reset, or it reaches a state $c^* \in R$ in which Equation 1 holds.*

Equation 1.

$$(\exists S : (p_j \in S) \wedge (|S| \geq 4f + 1) \wedge (\forall_{p_k, p_\ell \in S} : \tag{6.1}$$
$$(((phs_k[\ell], views_k[\ell]) = (phs_j[j], views_j[j])) \wedge ((echo_j[k] = echo_j[j]) \wedge witnesses_j[j] = \mathsf{True})) \wedge$$
$$(\exists_{m \in L_{j,k} \cup L_{k,j}} \Rightarrow m = (phs_j[j], witnesses_j[j] = \mathsf{True}, views_j[j], echo_j[j])) \wedge \tag{6.2}$$
$$(\exists_{m \in L_{k,\ell} : j \notin \{k,\ell\}} \Rightarrow m = (phs_j[j], \bullet, views_j[j], (phs_j[j], \bullet, views_j[j]))) \wedge \tag{6.3}$$
$$(p_j \in witnesSet_k) \wedge \tag{6.4}$$
$$(witnesses_j[j] \wedge (|witnesSet_j \cup \{p_j\}| \geq 4f + 1)))) \tag{6.5}$$

Proof. Let R' be a prefix of R that includes $O(xn)$ asynchronous rounds during which at least one processor $p_j \in \Sigma$ takes at least $z \geq 9$ steps that change $(phs_j[j], witnesses_j[j], views_j[j])$ without a reset taking place. Denote with $c_{j,y}$ the system state that precedes the step $a_{j,y}$ of p_j that changes this triple for the y^{th} time.

A. Parts (4) and (5) of Equation 1 hold. Assume that a processor $p_j \in \Sigma$ is the first to make at least $z \geq 9$ changes to $(phs_j[j], witnesses_j[j], views_j[j])$. Then by Remark 6.3.2, p_j cycles through the four transition phases at least twice, and further implies that p_j changes $(phs_j[j], witnesses_j[j], views_j[j]) = (1, \mathsf{True}, \bullet)$ twice, and executes $automaton_j(\text{`act'}, 1, 0)$ twice and then changes to $(0, \mathsf{True}, \bullet)$ once more. The latter has two implications: (1) Alg. 8, line 15 condition of the if statement holds when $automaton_j(\text{`act'}, 1, 0)$ is called, (2) since the call to $automaton_j(\text{`act'}, 1, 0)$ is followed by a call to $nextPhs_j()$ (Alg. 8, line 20), then $witnesSet_j$ becomes empty, and transitions to $(0, \mathsf{False}, \bullet)$ by the definition of $nextPhs()$.

Consequently, between the first and the second execution of $automaton_j(\text{`act'}, 1, 0)$, implication (2) holds and p_j empties $witnesSet_j$ and repopulates $witnesSet_j$ with $p_j \in \Sigma$ again. Therefore, at $c_{j,z-2}$, that is immediately before reaching $(0, \mathsf{False}, \bullet)$ for the second time, $(1, \mathsf{True}, \bullet)$ holds, i.e., part (5) of Equation 1 that $(witnesses_j[j] \wedge (|witnesSet_j \cup \{p_j\}| \geq 4f + 1))$ holds. Since no processor p_k makes a step immediately after $c_{j,z-2}$ other than p_j, no p_k empties $witnesSeen_k$ at $c_{j,z-2}$. By Claim 6.3.7, the above requires $O(9n) \in O(n)$ asynchronous rounds.

B. The values of $(phs, witnesses, views)$ **change concurrently and in a unison manner for any processor in** Σ**.** We assume that no resets take place within R. We prove that if $p_j, p_k \in \Sigma$ proceed in unison, this implies that if p_k obtains a view v^+ and does

Situation	Phase, View	Possible Automaton Cases		Resulting situation
(1)	$(0, \langle v^+, v^+ \rangle)$	$(0,1)$	**(1a)**	**(3)**
		$(0,2)$	**(1b)**	**(1)**
(2)	$(0, \langle v^-, v^- \rangle)$	$(0,0)$	**(2a)**	**(3)**
		$(0,1)$	**(2b)**	**(3)**
		$(0,2)$	**(2c)**	**(2)**
(3)	$(1, \langle v^-, v^+ \rangle)$	$(1,1)$	**(3a)**	**(1)**
		$(1,2)$	**(3b)**	**(3)**
(4)	$(1, \langle v^-, v^* \rangle)$ with $^* \neq {}^+$	$(1,0)$		**(3)**

Table 6.2: Case analysis for Lemma 6.3.8, distinguishing the different cases (situations) where the automaton acts and which overall effects it has.

not change this within $O(1)$ asynchronous rounds, then p_j with some view $v^- \neq v^+$ also changes its view to v^+. We define the following three sets regarding the global support to views :

(i) $V^- = \{p_k \in P : views_k[k] = \langle v^-, v^- \rangle\}$,

(ii) $V^{\pm} = \{p_k \in P : views_k[k] = \langle v^-, v^+ \rangle\}$,

(iii) $V^+ = \{p_k \in P : views_k[k] = \langle v^+, v^+ \rangle\}$.

We proceed with the case analysis that we distinguish by the starting and resulting **(phase, view (pair))** states.

The proof is by exhaustive case analysis, for which we define the possible situations that a processor p_x meets in moving from view v^- to v^+ in Table 6.2. We assume that the conditions of Alg. 8 line 15, hold for p_x, i.e., $witnesSeen_{x_j}() = \mathsf{True}$, and is (without loss of generality) the first processor to change the triple.

Before p_x changes the triple – Situation (2c). Since p_x does not change the view then this is Situation **(2b)** where $3f + 1 \leq |V^-| + |V^{\pm}| < 4f + 1$ (i.e., $automaton(\text{`pred'}, 0, 2) = \mathsf{True}$).

From $(\mathbf{0}, \langle \mathbf{v}^-, \mathbf{v}^- \rangle)$ to $(\mathbf{1}, \langle \mathbf{v}^-, \mathbf{v}^+ \rangle)$ – Situation (2b). The step that p_x takes includes that $automaton(\text{`pred'}, 0, 1) = \mathsf{True}$. I.e., $|V^-| + |V^{\pm}| \geq 4f + 1$ and $vChange_{x_j} = \mathsf{True}$. Note that if $vChange_{x_j} = \mathsf{True}$ it is guaranteed that another $4f$ processors have $vChange = \mathsf{True}$ by the Primary Monitoring mechanism as proved therein. If p_x is the first to proceed, then $|V^{\pm}| = 0$. Note that this ensures unison for $4f + 1$ processors, since these need to have the same view and *vChange* flag, for any of these to proceed.

Before p_x moves from $(\mathbf{1}, \langle \mathbf{v}^-, \mathbf{v}^+ \rangle$ to $(\mathbf{1}, \langle \mathbf{v}^+, \mathbf{v}^+ \rangle)$ – Situation (3b). Since p_x does not change the view then this is Situation **(2b)**, where $|V^-| + |V^{\pm}| \geq 3f + 1$ and $|V^{\pm}| + |V^+| < 4f + 1$ (i.e., $automaton(\text{`pred'}, 1, 2) = \mathsf{True}$). Note that since p_x moved to

this point having seen $|V^-| + |V^\pm| \geq 4f + 1$, and by Parts (4) and (5) of Equation 1, p_x does not proceed to a new step until its view is witnessed.

From $(\mathbf{1}, \langle \mathbf{v}^-, \mathbf{v}^+ \rangle)$ to $(\mathbf{1}, \langle \mathbf{v}^+, \mathbf{v}^+ \rangle)$ – Situation (3a). The step that p_x takes includes that *automaton*('pred', 0, 1) = True, i.e., $|V^\pm| + |V^+| \geq 4f + 1$. If p_x is the first to proceed $|V^\pm| = 4f + 1$. If not, then there must be a number of processors in $|V^+|$. We note that it is impossible for both of *automaton*('pred', 0, 1) and *automaton*('pred', 0, 2) to fail for p_x, since the witnessing mechanism implies that another $4f + 1$ processors are aware of this view and taking their steps must move to this view. Note that this is the desired case, at which we reach Situation **(1)**.

From $(\mathbf{0}, \langle \mathbf{v}^-, \mathbf{v}^- \rangle)$ to $(\mathbf{1}, \langle \mathbf{v}^-, \mathbf{v}^+ \rangle)$ – Situation (2a). Since processors proceed with $4f+1$, it is possible that some f processors are left behind the process. Since this is not a case for reset, *automaton*('pred', 0, 0) becomes True upon finding that $|V^\pm| + |V^+| \geq 3f + 1$. That this will be True before a reset takes place (*automaton*('pred', 0, 3)), is implied by the order of execution of the automaton cases, and also by the FIFO communication and witnessing mechanism.

From $(\mathbf{1}, \langle \mathbf{v}^-, * \rangle)$ to $(\mathbf{1}, \langle \mathbf{v}^-, \mathbf{v}^+ \rangle)$ (where $* \neq v^+$) – Situation (2a). This is Situation **4**, the respective for Situation **(2a)** in phase 1.

For both Situations **(2a)** and **(4)** we note that progress may be possible without these processors (as they are at most f), but within 1 step (and $O(1)$ asynchronous round) they can enter back to the view held by most of the processors, and thus be included in the $4f + 1$ required to satisfy Situations **(2b)** and **(3a)**. Note that the combined cardinalities of V^-, V^+ and $V^\pm$ encapsulate abstractly the notion of *transitAdoble*() (Alg. 8, line 10) and *establishable*() (Alg. 8 line 17) that allow some overlap of processors in previous current and next views. We conclude that the witnessing mechanism and the thresholds implemented by the automaton impose a lockstep behavior in the changes of views (and thus phases).

C. Parts (1), (2) and (3) of Equation 1 hold. By Part B of the proof we infer that indeed there exists a set S' of size $4f+1$, that is accumulated just before *estable*() = True (in *automaton*('pred', 0, 1) and *automaton*('pred', 1, 1)) holds. The first processor that makes a transition to either phase 1 by Situation **3b** of the previous part of the proof, or to phase 0 by Situation **3a** (because it is the first one for which *estable*() = True) needs to satisfy the conditions of line 15. This processor will take the step to change the triple (*phase*, *witnesses*, *views*). The set S' that enabled the satisfaction of *estable*() = True can

be identified as the set S of Equation 1, which thus exists, making the proof correct. □

We proceed to prove that the consistency set that was assumed to exist in many of the previous results is not attainable in the presence of Byzantine behavior.

Lemma 6.3.9. *Let R be a mal-admissible execution. There exists an initial state in which no consistency set Σ can make progress.*

Proof. We define a system state that is the result of a transient fault. We let $|\Sigma| = 5f$ and $\exists \Sigma', \Sigma'' \subset \Sigma : (\Sigma' \cup \Sigma'' = \Sigma) \wedge (|\Sigma'| = 3f) \wedge (|\Sigma''| = 2f)$. Note that the definition implies that $(\Sigma' \cap \Sigma'' = \emptyset)$. Also, $\forall p_i \in \Sigma'$ $(phs_i[i] = 0, wit_i[i] = \mathsf{True}, views_i[i] = v)$ where v is some view, and $\forall p_j \in \Sigma''$ $(phs_i[i] = 0, wit_i[i] = \mathsf{True}, views_j[j] = \mathsf{RST\_PAIR})$. Moreover, $P \setminus \Sigma = \{p_c\}$ some processor p_c that is malicious by not responding, and $\forall p_i \in \Sigma'$ $(views_i[c] = v)$ and $\forall p_j \in \Sigma'$ $((views_j[c] = v') \wedge (v' \neq v))$.

We argue that there is no Σ excluding p_c that can make progress and establish a stable view. Within $O(1)$ asynchronous rounds all processors in Σ satisfy *witnesSeen*() and thus can execute an automaton predicate. We note that $\forall p_i \in \Sigma'((automaton_i(\text{`pred'}, 0, 2)) = \mathsf{True})$ since they see $v$ as *transitAdoble*(), and that $\forall p_j \in \Sigma'((automaton_j(\text{`pred'}, 0, 2)) = \mathsf{True})$ since they have views equal to $\mathsf{RST\_PAIR}$. But $automaton_i(\text{`act'}, 2, 0)$ is a 'No Action' predicate. Thus there can be no progress, and the system always remains in an illegal system state without a stable view. □

In view of this case that can be generalized to include a group of cases, we argue that in the absence of malicious behavior the system can converge to a stable view. We thus use the notion of a mal-free execution. In the case of Lemma 6.3.9, a mal-free execution forces p_c to state a view, that will either, support v, or force a reset, and thus there is progress.

Theorem 6.3.10 (Convergence)**.** *Consider a mal-free execution of the View Establishment module starting in an arbitrary state. Within $O(n)$ asynchronous rounds, the system reaches a state in which there is a stable view v. Moreover, every correct processor eventually adopts view v.*

Proof. By Lemma 6.3.1 and Corollary 6.3.2 any stale information satisfying the definition of $staleV_i()$ is removed from the local state and the communication channels of $p_i \in P$. Within one complete iteration of Algorithm 8, there is a call to $needReset_i()$ (line 12) that also checks whether the replication module requests a view reset (via function $Alg6.5.repRequestReset_i()$). This takes $O(1)$ asynchronous rounds.

If $needReset_i()$ = True then there is a call to $resetAll_i()$ that sets $(phs_i[i], witnesses_i[i], views_i[i])$ to $(0, \mathsf{False}, \mathsf{RST\_PAIR})$. If there is a stable view v, then by its propagation, within $O(1)$ asynchronous rounds the automaton predicates as proved in Lemma 6.3.8, Part. B, will allow p_i to make a step by changing its triple to $(1, \mathsf{False}, v)$. Within $O(n)$ asynchronous rounds, p_i will make the step towards $(0, \mathsf{False}, v)$ which installs the view.

If, on the other hand, there is no such view, then within $O(n)$ asynchronous rounds (by Lemma 6.3.8), $transitAdoble_i()$ should not hold for any processor, and at least $4f+1$ processors should set their triples to $(0, \mathsf{False}, \mathsf{RST\_PAIR})$. The system then proceeds to install a view via the two automaton predicates $automaton_i(\text{`pred'}, 0, 1)$ and $automaton_i(\text{`pred'}, 1, 1)$. The latter has the action that $views_i[i]$ takes the value $\langle \mathsf{DF\_VIEW}, \mathsf{DF\_VIEW} \rangle$ which becomes stable. This completes the convergence proof. □

Theorem 6.3.11 (Closure)**.** *Consider a mal-admissible execution R, starting with a state that encodes a stable view v. Either, v remains stable throughout R, or within $O(1)$ asynchronous rounds the system reaches $c \in R$ with a new stable view v'.*

Proof. Assuming convergence under a mal-free execution, the only factor that further challenges the correctness of the view establishment module is the presence of malicious behavior during mal-admissible executions. Since a stable view is defined on correct processors, and since the consistency set Σ was defined for $4f + 1$ correctly behaving processors, the proof carries through. Changes to the table view are conducted either: (i) through the view change (automaton predicates $automaton_i(\text{`pred'}, 0, 1)$ and $automaton_i(\text{`pred'}, 1, 1)$), or (ii) by the adoption of an adoptable view (via the automaton predicates $automaton_i(\text{`pred'}, 0, 0)$ and $automaton_i(\text{`pred'}, 1, 0)$).

Suppose that the view change from v to v' is supported by $3f+1$ correct processors that satisfy $automaton(\text{`pred'}, 0, 2)$. It is possible that the f malicious processors may support such a change if, for example, v' is malicious. Due to the unison manner of phase-view progress proved in Lemma 6.3.8, Part B, the correct processors always proceed with a $3f + 1$ core. This ensures that while trying to accumulate $4f + 1$ support to satisfy the *estable*() conditions, a correct processor p_i is always able to satisfy $transitAdoble(i, phs, \text{`Remain'})$ within $O(n)$ asynchronous rounds. Thus, view v remains stable even through the transition, and this is not lost until v' is installed

by correct processors one-by-one.

The predicates $automaton_i(\text{'pred'}, 0, 0)$ and $automaton_i(\text{'pred'}, 1, 0)$ and the definition of *transitSet*() allow slow correct processors to catch up with installing the view within $O(1)$ asynchronous rounds. This provides liveness by allowing correct processors to join the correct view and thus provide service. □

6.4 State Replication Algorithm

The replication module (Algorithm 10) conducts SMR if there is a serviceable view. Our protocol follows Castro and Liskov [38], deviating only when catering for self-stabilization. In particular, (i) we introduce specific bounds for all our structures, (ii) we require that clients communicate their requests to all replicas. We proceed with a description of our solution and its correctness.

6.4.1 Preliminaries

Clients and requests. Processors receive requests from a known fixed set of clients C, where $|C| = K$. Following the typical well-formedness condition, clients do not send a new request before a previous one is complete, i.e., until it receives $f + 1$ identical responses. It is beyond the scope of this work to establish whether the content of a given request is malicious [38], as we concentrate on the server side.

The BFT replication task. Consider the set of correct processors $C \subset P$, and a set of client requests $\mathcal{K} = \{\kappa_1, \kappa_2, \cdots, \kappa_K\}$. We define the BFT replication task to be that all processors in C agree on a total order of execution of the requests of $\mathcal{K}$. Moreover, the client that issued the request eventually receives $f+1$ identical request responses. After a transient fault takes place, safety (i.e., identical replica state) may be violated, until the system converges back to a legal state.

Sequence numbers. To impose a total order in the execution of requests, the primary assigns a unique sequence number $sq \in [0, \mathsf{MAXINT}]$ to each received request. This is an integer incremented from a practically inexhaustible counter e.g., a 64-bit one[2]. A transient fault may corrupt the counter to attain its maximum value abruptly. In

[2]If $\mathsf{MAXINT} = 2^{64}$ then, incremented per nanosecond, it can last for 500 years (virtually an infinity for any existing system).

Structures, Variables and Constants of Algorithm 10.

Constants – Notation: K the size of the clients set. $K\sigma$ the watermark from which the primary is allowed to use above the sequence number of its last executed request, to assign to pending requests, where σ us a system-defined constant. $\mathsf{DEF\_STATE} = \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \mathsf{False}, \mathsf{False} \rangle$ a default (or incorruptible fallback) state for queues and variables of the *rep* structure (see below). We define the relation $\rightsquigarrow$ between two states A, B, such that $A \rightsquigarrow B$ implies that A is a prefix of B (or vice versa) or that they are equal.

Variables: A request q by client c is formed as follows: $q = \langle c, t, o \rangle$ where t is a totally-ordered timestamp (local to c) and o is the requested operation. An accepted request takes the form $req = \langle$(request) q, (view) v, (seq. num.) $sq\rangle$. The replica's structure $rep[n] = \langle repState, rLog, pendReqs, reqQ, lastReq, conFlag, viewChanged \rangle$ where *repState* is the replica's state. *rLog* is a list of at most MAXINT requests storing $\langle req, xSet \rangle$ where *xSet* is the set of those that committed or claim to have executed a request (with $|xSet| \geq n$).
Queue *pendReqs* holds the requests received from clients, and has size σK. *reqQ* holds at most $3\sigma K$ requests in process request messages of the form $\langle req, (status)\ st \in \langle \mathsf{PRE-PREP}, \mathsf{PREP}, \mathsf{COMMIT} \rangle \rangle$. *lastReq*$[K]$ an array of holding the last executed request of each client $\langle$(request) q, (reply) $r\rangle$. *conFlag* a boolean field True when a state conflict is detected, and *viewChanged* a boolean field True when a change of view/primary is detected. Referring to fields in $rep_i[i].field$, we may omit the $rep[i]$ part. *seqn* is an integer counter in $[0, \mathsf{MAXINT}]^3$ which is the maximal known to have been assigned to a request. *needFlush* is a boolean. The variable *flush* is a boolean that the view establishment module modifies to demand a reset of the replica state after a view establishment. Variable *prim* stores the last reading of the primary's identifier from the view establishment algorithm.

Figure 6.5: Self-stabilizing Byzantine Replication Algorithm structures, variables and constants; code for processor p_i.

this case we reset the view, the state and reset *sq* to 0. Note that during view changes we do not reset the sequence number.

While we cope with transient faults corrupting the request counter, we may still have a malicious primary that tries to propose arbitrarily high sequence numbers to the requests in order to exhaust the counter. We follow Castro and Liskov in restricting a faulty primary from exhausting the counter by imposing an upper and lower bound on the sequence numbers that other processors will accept from the primary. We bound the sequence numbers *sq* that the primary can use for a request in any given instance of the execution to σK, where K is the cardinality of the clients set C, and σ is a system defined integer constant. Under this bound, the primary can only assign a sequence number to a pending request if *(i)* this is the lowest unassigned one that it is locally aware of, and *(ii)* if this sequence number is not σK away from the sequence number of the last executed request.

1 **Macros:** $flushLocal() = \{seqn \leftarrow 0$; **foreach** $p_j \in P$ **do** $rep[j] \leftarrow \perp\}$;
2 $msg(status\ t, int\ j)$ = **return** $\{x : x \in rep[j].reqQ.req \wedge x.status = t\}$;
3 $lastExec()$ = **return** $(\max_{x \in rLog}(x.sq))$;
4 $lastCommonExec()$ = **return** $(x : \max(x.req.sq : (x \in \{rep[j].rLog\}_{p_j \in P}) \wedge (|\{p_k \in P : x \in rep[k].rLog\}| \geq 3f + 1)))$;
5 $conflict() = (|\{p_j \in P : rep[j].conFlag = \text{True}\}| \geq 4f + 1)$;
6 $comPrefStates(d) = \{$**if** $(\exists S \subseteq \{rep[j].repState\}_{p_j \in P} : (x, y \in S \Leftrightarrow (x \leftrightsquigarrow y)) \wedge (|S| \geq d))$ **then return** S; **else return** $\emptyset\}$;
7 $getDsState() = \{$**if** $(((\exists \Pi \subseteq P : (\Pi = \{p_j \in P : rep_i[j] \leftrightsquigarrow (X = findConsState(comPrefStates(2f + 1), 2f + 1))\}) \wedge (2f + 1 \leq |\Pi| < 3f + 1))\ (\exists \Pi' \subseteq P : \forall p_k \in \Pi'(rep_i[k] = \perp)) \wedge (|\Pi| + |\Pi'| \geq 4f + 1))$ **then return** (X); **else return** $(\perp);\}$
8 $double()$ = **return** $(\exists x, x' \in rep[i].reqQ : x.q = x'.q \wedge x \neq x')$;
9 $staleReqSeqn()$ = **return** $((lastExec() + \sigma K > \text{MAXINT})$;
10 $unsupReq()$ = **return** $(\exists x \in \{rep[i].reqQ.q\} : (|\{p_j \in P : x \in \{rep[j].reqQ.q\}\}| < 2f + 1))$
11 $staleRep()$ = **return** $(staleReqSeqn() \vee unsupReq() \vee (\exists x \in \{rep[i].rLog\} : |x.xSet| \leq 3f + 1) \vee double())$;
12 $knownPendReqs()$ = **return** $(\{x \in pendReqs : (|\{p_j \in P : x \in rep[j].\{pendReqs, reqQ.req\}\}| \geq 3f + 1)\})$
13 $knownReqs(status\ t)$ = **return** $(\{x \in reqQ : (|\{p_j \in P : x.req \in rep[j].msgQ.req\} \wedge x.status \in t| \geq 3f + 1)\})$
14 $delayed()$ = **return** $(lastExec() < lastCommonExec() + 3K\sigma)$
15 $existsPPrepMsg(x, prim)$ = **return** $(\exists y \in msg(\text{PRE}-\text{PREP}, prim) : y.req.q = x)$
16 $unassignedReqs()$ = **return** $(\{x \in pendReqs : (\neg existsPPrepMsg(x, prim)) \wedge x \notin knowReqs(\{\text{PREP}, \text{COMMIT}\})\})$
17 $acceptReqPPrep(x, prim)$ = **return** $((x \in knownPendReqs()) \wedge (\exists y \in msgQ : (y.req.q = x.q) \wedge existsPPrepMsg(y, prim) \wedge (y.v = prim) \wedge (lastExec() \leq y.sq < lastExec() + \sigma K) \wedge (\nexists z \in rep[i].\{reqQ.req, rLog.req\} : (z.q = y.q) \wedge (z.sq = y.sq)))$
18 $committedSet(x)$ **return** $(\{p_j \in P : (x \in msg(\{\text{COMMIT}\}, j)) \vee (x \in rep[j].rLog.req)\})$
19 **Interface functions:**
20 $getPendReqs() = \{$**if** $\neg viewChanged$ **then return** $(knownPendReqs() \cap unassignedReqs())$ **else return** {'View Change'}}
21 $repRequestReset() = \{$**if** $needFlush$ **then** $\{needFlush \leftarrow \text{False}$; **return** $(\text{True})\}$ **else return** $(\text{False});\}$
22 $replicaFlush() = \{flush \leftarrow \text{True};\}$

Figure 6.6: Variables and Macros for the BFT Replication algorithm; code for processor p_i.

6.4.2 Algorithm Description

We proceed with a detailed presentation of the Algorithm 10.

Variables. A request q by client c is formed as a triple $\langle c, t, o\rangle$ where t is a totally-ordered timestamp (local to c) and o is the requested operation. We call this an *unassigned* request since it has not been assigned a sequence number. An *assigned* request takes the form $req = \langle$(request) q, (view) v, (seq. num.) $sq\rangle$. The replica's structure $rep[n] = \langle$(replica state) *repState*, (executed req. log) *rLog*, (pending req. queue) *pendReqs*, (requests under process queue) *reqQ*, (last per client executed request) *lastReq*, (last assigned sq. num.) *seqn*, (conflict flag) *conFlag*$\rangle$, where *repState*

Algorithm 10: Self-stabilizing Byzantine Replication; code for processor p_i

do forever begin
  if $(\neg viewChanged \wedge Alg9.allowService())$ **then** $viewChanged \leftarrow ((rep[i] \neq \perp) \wedge (Alg9.getView(i) \neq prim))$;
  $prim \leftarrow Alg9.getView(i)$;
  if $(viewChanged = \mathsf{True} \wedge prim = i)$ **then**
    if $(\exists X \subseteq P : (p_j \in X \Leftrightarrow (rep_j[j].viewChange = \mathsf{True} \wedge prim = Alg9.getView(j))) \wedge (|X| \geq 4f + 1))$ **then**
      $renewReqs(X)$; $findConsState(comPrefState(3f + 1))$; $viewChanged \leftarrow \mathsf{False}$;
  else if $(viewChanged) \wedge (rep[prim].\langle viewChanged, prim\rangle = rep[i].\langle \mathsf{False}, prim\rangle) \wedge (|\{p_j \in P : prim = Alg9.getView(j)\}| \geq 4f + 1) \wedge checkNewVstate(prim)$ **then** $rep[i] \leftarrow rep[prim]$; $vChanged \leftarrow \mathsf{False}$;
  let $(X, Y) = (findConsState(comPrefStates(3f + 1)), getDsState(prim))$;
  if $(X = \perp \wedge Y \neq \perp)$ **then** $X \leftarrow Y$;
  if $(\neg(conFlag[i] \leftarrow (X = \perp)) \wedge (\neg(rep[i].repState \leftrightsquigarrow X) \vee (rep[i].repState = \mathsf{DEF\_STATE}) \vee delayed())$ **then** $rep[i] \leftarrow X$;
  if $staleRep() \vee conflict()$ **then** $flushLocal()$; $rep[i] \leftarrow \perp$; $needFlush \leftarrow \mathsf{True}$;
  if $flush$ **then** $flushLocal()$;
  $pendReqs.enqueue(\{knownPendReqs()\})$;
  if $(Alg9.allowService() \wedge \neg needFlush)$ **then**
    if $noViewChange() \wedge \neg viewChange$ **then**
      if $(prim = i)$ **then**
        foreach $x \in rep[i].pendReqs$ **do**
          if $(seqn < lastExec() + \sigma K)$ **then**
            $reqQ.add(\{\langle\langle x, prim, (seqn \leftarrow seqn + 1)\rangle, \mathsf{PRE-PREP}\rangle, \langle\langle x, prim, (seqn \leftarrow seqn + 1)\rangle, \mathsf{PREP}\rangle\})$;
        else foreach $x \in knownPendReqs() \setminus unassignedReqs() : x \notin \{rep[j].reqQ\}_{p_j \in P}.req.q$ **do**
          if $(acceptReqPPrep(x, prim))$ **then** $reqQ.add(\langle y, \mathsf{PREP}\rangle)$;
        foreach $x \in reqQ : knownReqs(\{\mathsf{PREP}\})$ **do** $x.status \leftarrow \mathsf{COMMIT}$; $pendReqs.remove(x)$;
        foreach $x \in reqQ : knownReqs(\{\mathsf{PREP}, \mathsf{COMMIT}\}, i)\}$ **do**
          if $(|X = committedSet(x)| \geq 3f + 1) \wedge (x.sq = lastExec() + 1)$ **then**
            $lastReq.enqueue(\langle\{x\}, apply(x)\rangle)$; $rLog.add(\langle x, X\rangle)$; $pendReqs.remove(x)$; $reqQ.remove(x)$;
  foreach $p_j \in P \setminus \{p_i\}$ **do send** $rep[i]$ **to** p_j;
  foreach $(cl, x) \in C \times lastReq : x.req.q.c = cl$ **do send** x **to** cl;;
Upon receipt $m = rep$ **from** p_j **do if** $allowService()$ **then**
  if $(noViewChange())$ **then** $rep[j] \leftarrow m$ **else** $rep[j].repState \leftarrow m.repState$;
Upon receipt of request q **from client** c **do if** $(noViewChange() \wedge allowService())$ **then** $pendReqs.enqueue(q)$;
Upon receipt of $m = \langle q, \mathsf{ACK}\rangle$ **from client** c **do** $lastReq.remove(\langle m.q, \bullet, \bullet\rangle)$;

is the replica's state (the replicate) which is an ordered sequence log. We define the relation $\leftrightsquigarrow$ between two states A, B with prefixes A', B', such that $A \leftrightsquigarrow B$ implies

that A is a prefix of B (or vice versa) or that they are equal, i.e., $A \leftrightsquigarrow B \iff \exists A', B' : (A' = B \vee B' = A)$. List *rLog* has a size of at most MAXINT processed requests storing $\langle req, xSet \rangle$ where *xSet* is the set of those that committed or claim to have executed a request (with $|xSet| \geq n$). A default fallback state DEF_STATE = $\langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \text{False}, \text{False} \rangle$.

Queue *pendReqs* holds the requests received from clients, and has size σK. Queue *reqQ* holds at most $3\sigma K$ requests messages of the form $\langle req, (status)\ st \in \langle \text{PRE-PREP}, \text{PREP}, \text{COMMIT} \rangle \rangle$. Array *lastReq*[$K$] holds the last request reply $\langle$(request) q, (client) c, (reply) $r\rangle$ to each client. The field *seqn* is an integer counter in [0, MAXINT] that is the maximal known to have been assigned to any request. The variable *conFlag* a boolean field that is True when a state conflict is detected, and *viewChanged* a boolean field that is True when a change of view/primary is detected. Referring to fields in $rep_i[i].field$, we may omit the $rep[i]$ part.

The variable *needFlush* is a boolean reporting to the view establishment module whether a reset is required. The variable $flush$ is a boolean that the view establishment module modifies to demand a reset of the replica state after a view establishment. Variable *prim* stores the last reading of the primary's identifier from the view establishment algorithm.

Operators, Macros and Interface functions.

- Operator *enqueue*(x) adds an element (or set of elements) x to a queue. If any element enqueued already exists, then only the most recent copy of it is kept and it is carried to the back of the queue.
- Operator *remove*(x) removes element x from a structure while *add*(x) adds element x to a structure.
- Operator *apply*() executes any request operations that are known to be committed.
- Operator $findConsState(S, x)$ returns a consolidated replica state *rep*[] based on a set of processor states S with common *non-empty repState* prefix and consistency among request queues *reqQ* and *pendReqs*(). It returns $\perp$ if such a replica state set does not exist (indicated as $\emptyset$); It produces dummy requests in the case where at least $3f + 1$ processors appear to have committed a sufficient number of requests but they have no evidence of a previous request exists or is assigned.

This request is blocking the execution of the requests that follow.

- Operator *renewReqs*() is executed by a new primary, in order to issue a consistent set of pending requests messages for *reqQ* and *pendReqs* where these are now allocated for execution to the new view.
- Operator *checkNewVstate*() checks the state proposed by a newly installed primary after a view change. This involves checking whether the proposed pre-prepare messages of committed processors are verified by another $3f + 1$ processors and the new state has a correct prefix as per $findConsState()$.
- Macro $flushLocal()$ resets all local values of *rep*[].
- Macro $msg(status\ t, int\ j)$ returns all the requests the p_j reported to p_i that have a specific status or set of statuses (e.g., $\mathsf{PRE-PREP}$ or $\{\mathsf{PREP}, \mathsf{COMMIT}\}$).
- Macro *lastExec*() returns the last request sequence number executed by p_i.
- Macro *lastCommonExec*() returns the last request that p_i sees locally to have been executed by at least $3f + 1$ processors.
- Macro $conflict()$ returns True if $4f + 1$ processors report to have their conflict flag *conFlag* = True.
- Macro $comPrefStates(d)$ returns a set of *repState*s that satisfy the common prefix relation $\rightsquigarrow$ for at least than d processors. If no such exists then it returns $\emptyset$.
- Macro *getDsState*() returns a prefix suggested by $2f+1 \leq x < 3f+1$ processors, with the requirement that there exist another y processors that have $rep[] = \perp$ such that $x + y \geq 4f + 1$. If such a prefix doesn't exist it returns $\perp$.
- Macro *double*() returns True if the *reqQ* of p_i contains two copies of a request and they have different view or sequence number.
- Macro *staleReqSeqn*() returns True if the sequence number has reached the maximal counter value MAXINT.
- Macro *unsupReq*() returns True if a request exists in *reqQ* less than 2f+1 times.
- Macro *staleRep*() = returns True if any of *double*(), *unsupReq*() or *staleRep*() are True.
- Macro *knownPendReqs*() returns a set of requests that appear in the $rep_i[i].pendReqs$ and also appear in the message queues of at least another $3f+1$

processors.

- Macro *knownReqs*(*status t*) returns a set of requests that appear in the $rep_i[i].reqQ$ and of at least another $3f + 1$ processors and have a status in t.
- Macro *unassignedReqs*() returns the set of pending requests for which p_i has neither seen a PRE−PREP message from the primary, nor has it seen $3f + 1$ processors that have a PREP message for the same client request.
- Macro *acceptReqPPrep*(x, *prim*) returns True if there is a pre-prepare message from the primary *prim* for a request x and the request content is the same for $3f + 1$ processors with the same sequence number and view identifier.
- Function *getPendReqs*() returns to the calling Algorithm the set of requests in $rep_i[i].pendReqs$ that were not assigned a sequence number by the primary and also appear in the request queues of that other processors report to p_i.
- Function *repRequestReset*() allows the view establishment algorithm to set the *needFlush* flag of the replication module to False after it has taken into consideration the fact that it requires a reset. This ensures that the view establishment module does not trigger a reset repeatedly due to a single request.
- Function *replicaFlush*() allows the view establishment algorithm to demand a reset of the replica's state because it has performed a reset, which is a sign of a corrupt state. It does so by setting the flag $flush$ to True.

Detailed Description. Algorithm 10 implements the replication procedure by first checking and handling replica state conflicts possibly requiring a reset (lines 10–12). It then processes messages that have not been assigned a sequence number (lines 14–20), and then proceeds to maintain the queues of prepared and committed requests, before applying effects to the replica for committed requests (lines 21–24). It finally propagates information to the other replicas and to the clients (lines 25–26). Lines 27 – 30 define the receive side of the communication. A more detailed description from processor p_i's view follows.

Line 2 checks if there was a completed view change that resulted in itself being the primary, and in this case (and only in the first iteration) it sets Boolean $vChanged_i$ = True. Line 3 reads the current value of the view/primary into *prim*. If p_i detects that it has become the primary line 4, then by line 5 it waits for $4f + 1$ processors with

viewChanged flags to accumulate before executing *renewReqs*() and $findConsState$(), which will have the following effects: it creates a pre-prepare message for every committed request that had not been executed by $4f + 1$ processors. It also creates a consolidated state with a common *repState* prefix and *rLog* prefix. (line 6). If *viewChanged* holds then if there is a set of $4f+1$ processors with the same view and the primary of this view appears with a flag *viewChanged* = False and *checkNewVstate*() checks the consistency of the state and the requests that the new primary has sent, then p_i sets $vChanged_i() =$ False (line 7).

Line 8 draws the consolidated state based on $3f + 1$ and by *getDsState*() the consolidated state in the special case where there are $2f + 1$ to $3f$ correct processors with a common replicate prefix and $2f$ to $f+1$ corresponding correct processors with in the DEF_STATE. A processor among the latter group will not find the first case of consolidated state but will be able to eventually (as we argue in the proof) find the second type of consolidate state. Line 10 checks for conflicts (and sets the conflict flag to True) and also adopts a common prefix (as a state transfer) in case its prefix is obsolete and the requests it has do not allow it to catch up by executing the replica. In case of conflict or detected local stale information the local variables are reset, and there is a request to the view establishment module to perform a view change (line 11). If the view establishment module performed a reset, then it instructs for a reset of all local variables in the other modules (line 12). Until this point, the action were orientated towards self-stabilization and ensuring (based on local information) that there are no stale information from an arbitrary initial state.

Line 13 adds to *pendReqs* any requests that were not received by the communication with the client (line 29), but they appear as being known by another $3f + 1$ processors. Pending requests are processed by the primary (lines 16 – 18) by assigning them the next available sequence number restricted by an integer parameter $\sigma \cdot K$ from the sequence number of the last reported executed request. Non-primary nodes (line 19) wait for the primary's ordering of the requests before adding them to the prepared views, or in case $3f + 1$ processors appear to have a common sequence number for this request, it accepts this and adds the request to the *reqQ* with status PREP (lines 19–20). Line 21 moves *known* messages from prepare to commit status, and removes this request from pending. (Note that as per the definition PBFT [38], the PRE−PREP and PREP phases define the order within a specific view, while commits across any view.) If $3f + 1$ processors appear to have reached status

COMMIT for a request (line 22) this is considered as committed. Line 23 executes the committed clients' requests (by calling *apply*()) and adds them in order in the *rLog* queue, but also in *lastReq*, which is used to inform clients of their last executed requests (line 26).

Requests are removed from *lastReq* once they are acknowledged (line 30) or once they are dequeued by newer requests. The iteration is completed by sending the replica information to the other processors (line 25). Lines 27–28 treats the receipt of *rep*[] from other processors. If a view establishment is taking place, this is discarded, if a view change is taking place then only the *repState* part is stored as no new requests are accepted during a view changes.

6.4.3 Correctness

Definitions. Two correct processors p_i, p_j with replica states S_i, S_j have a *common state prefix* (CSP), if the replica state that applied the least number of transitions (say, S_i) is equal to a prefix of the state of the other processor, i.e., $S_i = S'_j : S_j = S'_j \circ S''_j$. Correct processors p_i, p_j have a *consistent CSP* (CCSP) if they have a CSP and their transitions history is identical, and the system does not encode a corrupted message or local stale information that can make the two states to divert. Definition 6.4.1 specifies the replication task. A system state satisfying the replication task $\mathcal{RT}$ defined below is a $safe$ one. Moreover, $\mathcal{RT}$ defines the set of legal executions.

Definition 6.4.1 (Replication task)**.** *Consider an execution R starting in an arbitrary state c. A* safe system state *satisfying the replication task $\mathcal{RT}$ is a state in which there exists a set of correct processors $P_s \subseteq C$, where $|P_s| \geq 2f + 1$, that have:*
(a) *a CCSP,*
(b) *a common history of executed requests $rLog = rLog' \circ rLog''$, where:*

(i) Prefix $rLog'$ is an identically ordered log of executed messages with sequence numbers $[0, k]$*, and*

(ii) Suffix $rLog''$ (where $|rLog''| \geq 3K\sigma$) may differ in every processor in P_s, such that at least one $p \in P_s$ has $rLog'' = \emptyset$ and other processors can have $0 \geq |rLog''| \geq 3K\sigma$. For all processors in P_s, all the requests in $rLog''$ have sequence numbers in $[k + 1, k + 3K\sigma]$.
(c) The communication link $L_{i,j}$ between two correct processors $p_i, p_j \in P_s$ does not contain a corrupt rep[•] *that can force the receiving end to adopt a repState that can lead to a call to $flushLocal()$ or an assignment needFlush* ← False.

Proof Outline. The proof only considers mal-admissible executions, and, unless stated otherwise, it assumes that a view is in place and known by every correct processor. Initially, it establishes that within $O(1)$ asynchronous rounds there is no local stale information (Claim 6.4.1). We then deduce that the system manages within $O(1)$ asynchronous rounds to establish that there is a CCSP, that the order of executions is agreed upon, and that the communication channels cannot cause corruption. The proof continues to show that if there is a CCSP satisfying the $\mathcal{RT}$, then either every correct processor will take this CCSP, or a reset takes place at most once in the execution (Lemma 6.4.9). Lemma 6.4.3 suggests that after a view reset, there is a safe system state and correct processors start from a common default CCSP DEF_STATE. This leads to Theorem 6.4.10 which is the convergence theorem. We conclude with the closure theorem (Theorem 6.4.11) that proves that after convergence (i.e., within $O(1)$ asynchronous rounds), correct processors executing Algorithm 10 always satisfy $\mathcal{RT}$ and perform state replication. We also argue that the safety (i.e., the integrity of the state and request order) is not affected upon a primary (view) change.

Claim 6.4.1. *Within one complete iteration of Algorithm 10, a correct processor $p_i \in C$ has no local stale information as the ones defined by predicate staleRep() (Fig. 6.6, line 11).*

Proof. A complete iteration of Algorithm 10 contains a call to $staleRep_i()$ (line 11), which, if it returns True, will allow a call to $flushLocal_i()$. By the definition of *flushLocal*() the local structures are set to a default (empty) state. We argue that *staleRep*() never holds again.

Specifically, *double*() never holds again since before adding a request from *pendReqs* to *reqQ* it is checked not to be a double entry by condition $\nexists z \in rep_i[i].\{reqQ_i, rLog_i\} : (z.sq = y.sq))$ where y is a request in the primary's PRE−PREP requests that was assigned a sequence number (line 19). Condition *staleReqSeqn*() does not hold since the *seqn* after a reset is set to 0 and does not reach the MAXINT unless a transient fault takes place. Also, *unsupRep*() = False since every message is added to the *reqQ* only if it is known to be acknowledged by $3f + 1$ processors. Since the queues are emptied then there is no local support for any corrupt request, and for requests received, the support arrives with new messages from other processors. Line 19 adds only when $3f + 1$ support exists for a message, and thus there are $2f + 1$ correct processors that will always support this message, thus dissatisfying the requirements of *usnsupRe*() for less than $2f + 1$ support for a request. Finally,

$(\exists x \in \{rep[i].rLog\} : |x.xSet| \leq 3f + 1)$ does not hold after a reset, since the condition to add a request to *rLog* is to have an $|xSet| \geq 3f + 1$ by line 23.

We conclude that, upon the execution of the first complete iteration of every correct processor in the system, the result holds. □

Claim 6.4.2. *Let R be a fair execution starting in an arbitrary initial state. Within $O(1)$ asynchronous rounds, the system reaches a state $c \in R$ where there are no corrupt messages from the initial arbitrary state that can make part* (c) *of $\mathcal{RT}$ be* False.

Proof. By the specification of the data link protocols (Section 3) defining FIFO communication, and by the fair execution assumption, the completion of an asynchronous round implies the receipt (by correct processors) of all the messages that a correct processor sent at that iteration. This implies that when these messages were received, all the corrupt messages will also have been received. Thus the link carries no other messages that come from the initial arbitrary state. The result follows. □

Claim 6.4.3. *Let R be an execution starting in an arbitrary initial state such that part (b) of the $\mathcal{RT}$ is not satisfied. Within $O(1)$ asynchronous rounds, the system reaches a state $c \in R$ in which (b) is satisfied or there is a call for a view reset.*

Proof. By the communication (lines 25 and 28) a processor p_i sends $rLog_i$ and receives $rep_j[j].rLog$ for all processors $p_j \in C$ and possibly of some malicious processors. Since part (b) of $\mathcal{RT}$ does not hold, it must be that more than $3f + 1$ correct processors p_k will have that $delayed_k()$ = True since $(lastExec_i() < lastCommonExec_i() + 3K\sigma)$. We note that in this case, the processors cannot catch up with the replication (the $reqQ()$ can only accommodate $3K\sigma$ messages) and $findConsState()$ will return $\perp$ based on the received information, letting *conFlag* = True. Thus, each of processors p_k raises their *conFlag* and is led to a reset. The result follows since the reset state is a safe one in which (b) is satisfied, as proved in the following result. □

Claim 6.4.4. *A system state with a default replica state* DEF_STATE *held by $3f + 1$ correct processors is a safe system state.*

Proof. DEF_STATE satisfies the definitions of $\mathcal{RT}$. Part (a) of $\mathcal{RT}$ is satisfied since the DEF_STATE is trivially a CCSP which forms a common *repState*. Part (b) holds since the $pendReqs_i$ and $reqQ_i$ queues are all emptied upon a $localFlush_i()$, so there cannot be inconsistencies in the message queues, nor a *delayed*() message sequence number. Part (c) holds by the reasoning of the following Claim 6.4.2. □

Claim 6.4.5. *Let R be an execution starting in an arbitrary state and $c \in R$ be the system state that completes a view reset. Within $O(1)$ asynchronous rounds the system reaches a safe system state with respect to the replication task.*

Proof. By the view establishment algorithm, *automaton*('act', 1, 1) has a first action that is a call to *replicaFlush*(). This sets variable $flush$ of Algorithm 10 to True. In the next iteration of Algorithm 10 by p_i, $flush_i$ = True and this satisfies the condition of line 12. There is thus a call to $flushLocal_i()$ in the very next iteration after the view change. A view reset is followed by at least $3f + 1$ correct processors because establishing a view requires $4f + 1$ processors. This implies that the above call to $flushLocal()$ is performed by at least $3f + 1$ correct processors. This satisfies parts (a) and (b) of the task $\mathcal{RT}$ directly, and (c) is also satisfied since Claim 6.4.2 holds. The result comes within $O(1)$ asynchronous rounds following the $O(n)$ asynchronous rounds required by the View Establishment module for the view reset (Theorem 6.3.10), and by Claim 6.4.4. □

Lemma 6.4.6. *Let R be an execution starting in a state that does not have a CCSP among $2f + 1$ correct processors that satisfies the definition of $\mathcal{RT}$. Within $O(n)$ asynchronous rounds at least $3f + 1$ correct processors $p_i \in C$ reach a reset state $rep_i[i]$ = DEF_STATE.*

Proof. We assume, as a worse case, that the f malicious processors attempt to support some corrupt state prefix S_c that exists in the system state. Since there is no CCSP, there are $y < 2f + 1$ correct processors that may have S_c. Within a single asynchronous round, p_i with or without $repState_i = S_c$ will send and receive the states of every single correct processor via lines 25 and 28. In the next iteration, line 8 will assign $X_i = findConsState_i(comPrefStates_i()) = \perp$, since $comPrefStates_i()$ will return $\emptyset$, i.e., it will not find a common state prefix among $3f + 1$ processors. By line 10 p_i assigns $conFlag_i[i] \leftarrow$ True. Within another $O(1)$ asynchronous round, every correct processor p_j exchanges its $rep_j[j]$ and decides upon $rep_j[j].conFlag$ = True, while it is informed of $rep_i[i].conFlag$ = True. The result considers the $O(n)$ asynchronous rounds required for the view establishment.

In the next iteration ($conflict_i()$ = True) implies the execution of $flushLocal_i()$ and a set of the flag *repRequestReset* = True (line 11). This initiates a view reset by Algorithm 8. Lemma 6.4.5 leads to the result. □

Lemma 6.4.7. *Let R be a fair execution starting in an arbitrary initial state. Within $O(1)$*

asynchronous rounds, the execution has a suffix R' that contains no violations of conditions (a), (b) and (c) characterizing $\mathcal{RT}$, unless $2f + 1 \leq |P_s| < 3f$.

Proof. Claims 6.4.2, 6.4.3, and Lemma 6.4.6 tackle deviations from each of conditions (c), (b) and (a) respectively, leading to either a state reset or the adoption of a CCSP satisfying the definition of $\mathcal{RT}$. □

Lemma 6.4.8. *Consider an execution R starting in a safe system state and where processor p_i executes a request that is not part of the reqQ of a majority of correct processors. Within $O(1)$ asynchronous rounds, p_i reassigns a correct CCSP.*

Proof. This request may be the result of an initial arbitrary state. The result follows from the definition of the safe system state. Since there exists a CCSP, namely S_u, known by $3f + 1$ processors, within $O(1)$ asynchronous rounds, p_i has knowledge of the CCSP. The $findConsState()$ operator and the conditions of line 11 will reassign the CCSP to $rep_i[i]$ since $rep_i[i].repState \not\leftrightarrow S_u$, and since $rep_i[i].rLog$ differs from $3f+1$ the other processors. Given the arguments of Lemma 6.4.9 we deduce the result. □

Lemma 6.4.9. *Let R be an execution with a state S_u that is a CCSP satisfying parts (a) to (c) of the definition of $\mathcal{RT}$, and where P_s is the set of processors that have (some prefix of) state S_u. Within $O(1)$ asynchronous rounds, every correct processor p_i has $rep_i[i].repState \leftrightsquigarrow S_u$.*

Proof. Within $O(1)$ asynchronous rounds, by the communication mechanism (lines 25 and 27) the processors of P_s propagate S_u to all the correct processors. We study the following two cases.

Case 1 – ($|\mathbf{P_s}| \geq \mathbf{3f + 1}$). Every correct $p_i \notin P_s$ with $rep_i[i].repState \not\leftrightarrow S_u$ may initially apply $rep_i[i].repState = \mathsf{DEF\_STATE}$. Upon receiving S_u from the $3f+1$ correct processors, p_i has an iteration in which line 8 assigns $X_i = findConsState(comPrefStates(3f + 1))$ which is equal to S_u. Since $rep_i[i].repState$ is either equal to $\mathsf{DEF\_STATE}$ or not a prefix of S_u, by line 10 it assigns $x_i = S_u$ to $rep_i[i].repState$ and this gives the result.

Case 2 – ($\mathbf{2f + 1 \leq |P_s| < 3f}$). Note that initially in this case, the state is dependent on the malicious processors' good will, since they may retract their support and force a collapse of the state. Again, the propagation of S_u by the members of S_u takes place and it is possible that another f malicious processors (including the primary) are pretending to hold S_u. Since we have no guarantees that $findConsState(comPrefStates(3f + 1))$ will return S_u, we use the macro $getDsState()$ (line 7). This will return S_u only when the combined number of processors that

have S_u or the DEF_STATE as $rep[]$ is at least $4f + 1$. By the reasoning of Claim 6.4.6 correct processors that do not have S_u, will, within $O(1)$ asynchronous rounds, move to the DEF_STATE. Finally, $|P_s| \cup |\{p_j \in C : rep_j[j] = \text{DEF\_STATE}\}| \geq 4f + 1$. This satisfies *getDsState*() that returns S_u. By lines 8–10 we have that $rep_i[i] = S_u$. Note that the *getDsState*() is strong enough to impose that, once achieved, the state is no longer dependent on malicious processors since even if malicious processors retract their support, this leaves $3f + 1$ correct processors, enough to provide support as per Case 1. □

From the above we deduce the following convergence theorem.

Theorem 6.4.10 (Convergence)**.** *Consider an execution R of Algorithm 10 starting in an arbitrary state. Within $O(n)$ asynchronous rounds, the system reaches a safe system state $c_{safe} \in R$.*

Proof. Claims 6.4.1 – 6.4.3 show that local stale information as well as corrupt message in the communication links are removed. Claim 6.4.4 shows that if the DEF_STATE is a CCSP satisfying $\mathcal{RT}$ then this defines a safe system state. Lemma 6.4.6 proves that in the absence of a CCSP satisfying $\mathcal{RT}$ there is a call to Algorithm 8 that result to a safe system state (with DEF_STATE as the CCSP). Lemma 6.4.7 suggests that any violation to the $\mathcal{RT}$ are eventually eliminated from the system for the case where the CCSP is supported by more than $3f + 1$ correct processors, and Lemma 6.4.9 shows how the system will either reach to a safe system state with $|P_s| \geq 3f + 1$, or the execution reaches view reset that again results to a safe system state by Lemma 6.4.3. Essentially Lemma 6.4.9 shows that consensus is reached. Hence, Algorithm 10 allows convergence to the system task $\mathcal{RT}$ within $O(n)$ asynchronous rounds. □

Theorem 6.4.11 (Closure)**.** *Consider an execution R of Algorithm 10 starting in a safe system state c. Any subsequent state $c' \in R$ is a safe system state. Then R is a legal execution with respect to the replication task $\mathcal{RT}$ (Definition 6.4.1).*

Proof. We proceed to prove closure incrementally. Claim 6.4.12 establishes closure in the presence of a correct primary that progresses the state machine without delays, thus, the primary monitoring mechanism of other processors does not request a change. We define our liveness criteria for this assumption in the Primary Monitoring

module. As a second step, Claim 6.4.13 proves that safety is not lost if the primary acts maliciously. We conclude with Claim 6.4.14 showing that safety is retained between any number of view changes.

Liveness Assumption 1. *Let R be an execution of Algorithm 10 starting in a safe initial state. Throughout execution R, the primary p_i replicates at a pace that allows $|lastExec_j() - lastExec_i()| < \sigma K$ to hold, where $p_j \in C$ is the first processor to execute the committed request $lastExec_j()$. (It is possible to have $i = j$.)*

Note that in order to commit, $3f+1$ processors are required to appear as committed to this request with the same request number (line 21). Thus there exist at any point $2f + 1$ correct processors satisfying the $|lastExec_j() - lastExec_i()| < \sigma K$ condition of the above liveness assumption.

Claim 6.4.12. *Let R be a mal-admissible execution starting with a safe system state c in a view i with a correct primary p_i, and a view change is never requested by the Primary Monitoring module. Any subsequent state $c' \in R$ is safe.*

Proof. Note that a client may only issue a single request to the system and until this is executed, it does not send a new one, but it continuously propagates its request to all the processors in P. We assume that Liveness Assumption 1 holds.

Step 1 – At least 2f + 1 processors $\mathbf{p_j} \in C$ add $\tilde{\mathbf{q}}$ to $\mathbf{rep_j[j].pendReqs}$.

By the continuous propagation of requests on behalf of the clients, and the fair execution assumption, $\tilde{q}$ should be delivered to every p_j and added to $rep_j[j].pendReqs$ (line 29). A correct processor may also add a request to *pendReqs* via line 13 when $3f + 1$ processors appear to acknowledge the receipt of $\tilde{q}$ to one another via the propagation of *rep*[]. Since p_i is correct and communicating, it must receive $\tilde{q}$ in one of the two ways above.

Step 2 – Primary $\mathbf{p_i}$ assigns a sequence number to $\tilde{q}$ if $\mathbf{seqn_i} < \mathbf{lastExec_i}() + \sigma\mathbf{K}$.

Line 16 is satisfied only by the primary of the view, and line 17 is satisfied only if $seqn_i < lastExec_i() + \sigma K$. Since a client never injects two different requests before the first one is processed, there can be only K requests in the pending queue of the primary. A correct primary that is progressing replication always among the fastest $3f + 1$ processors (something desired for a primary) will only need to stay within K from $lastExec_i()$. But, since liveness-wise this is not something that one would like to assume, and to allow more flexibility for progress, the system defines σ to grant the

primary a margin of σK rather than just K request numbers. This is still restrictive, since $\sigma K \ll \mathsf{MAXINT}$, and so a primary cannot attempt to exhaust the counter by assigning a value close to MAXINT. The primary removes $\tilde{q}$ from $rep_i[i].pendReqs$ and adds it to $rep_i[i].reqQ_i$ with status $\mathsf{PRE-PREP}$, the current view (line 17), and the sequence number, and propagates this via line the communication mechanism. It also moves this request to status PREP, since it does not require to verify the sequence number and its validity as the other processors need to do. The replication procedure hereon does not involve the primary as a coordinating entity, and the primary performs replication as any of the other processors.

Step 3 – Within $O(1)$ asynchronous rounds, at least $2f+1$ processors $p_j \in C$ execute the operation of $\tilde{q}$ and add it to $rep_j[j].rLog$ in the order instructed by the primary. Within an asynchronous round after Step 2 holds, all correct processors p_j receive $rep_i[i].reqQ$, which implies that they receive $\langle\langle \tilde{q}, i, seqNumber\rangle, \mathsf{PRE-PREP}\rangle$. They accept this only if the conditions of line 19 hold (requiring at least $3f+1$ processors to have observed this request with the same view and $seqNum$), and the conditions of line 20 hold. This requires the uniqueness of the sequence number assigned, and that it is also within $lastExec_j() + \sigma K$. Hence, every p_j that remains within the $lastExec_j() + \sigma K$ bound, accepts this message given that $|lastExec_j() - lastExec_i()| < \sigma K$ holds (as per our assumption).

Upon acceptance every such p_j removes $\tilde{q}$ from $rep_j[j].pendReqs$ stopping the Failure Detector from inquiring about $\tilde{q}$ from this point onward. This adds it to $rep_i[i].reqQ_i$ with status PREP, along with the current view (line 17), and the sequence number $seqNumber$, and propagates this via the communication mechanism. Line 22 confirms that the request is committed if this appears as status PREP, or COMMIT, or is executed in $rLog$, only if this is reported by $3f$ processors. Within one asynchronous round, every correct processor p_j that had accepted $\tilde{q}$ propagates a PREP message for $\tilde{q}$, and receives at least $3f+1$ such evidence (where the f may be from malicious processors) from a set of processors X.

This implies that $\tilde{q}$ is committed and may be executed. The assumption of a correct primary p_i, implies p_i never skips a request number when assigning to a request. Thus condition $\tilde{q}.seqNumber = lastExec_j() + 1$ of line 23 holds once all previous requests are executed. We note that $findConsState()$ is assumed to produce dummy requests in case $3f+1$ processors appear to have committed a sufficient

number of requests but there is no evidence that a previous request number exists or is assigned. Upon execution of $\tilde{q}$ the safe state is preserved per the $\mathcal{RT}$ definition. and this concludes the proof. □

Claim 6.4.13. *Let R be a mal-admissible execution starting with a safe system state c in a view i with a malicious primary p_m. Any subsequent state $c' \in R$ is safe.*

Proof. We prove that none of conditions (a), (b) or (c) of $\mathcal{RT}$ is violated by the malicious actions of p_m, while p_m may also collude with the other malicious processors.

Condition (a) is not violated – A malicious primary is limited to the following actions that deviate from those of a correct primary:

1. Stop assigning sequence number to requests or stop propagating them.
2. Assigning a request number that was either reused, or does not obey the $lastExec_i() + \sigma K$ bound (which forces a violation of the Liveness Assumption 1).
3. Sending different request number to different processors.
4. Sending a *rep*[] that does not comply with the CCSP.
5. Modify a client's request.

Case (1) does affect correct processors, since at most K client requests will be accumulated in *pendReqs* of each processor. Clients will then wait for responses and will not send new requests. This is clearly a liveness issue to be settled in Section 6.5.
Case (2) does not affect the state of every correct processor p_j, since for a malicious request q_m the *acceptPPrepMsg*() macro protects against accepting a sequence number violating $lastExec_j() < q_m.seqNumber \leq lastExec_j() + \sigma K$.
Case (3) cannot violate the correctness of the replica, by the fact that a set X of $3f + 1$ processors need to appear as agreeing to progress a request under the same sequence number. As such, there can be no other group Y of processors of size $3f + 1$ such that $|X \cap Y| \leq f + 1$. This implies that there always exists $p_j \in C \cap X \cap Y$ that is aware of the first assignment of the sequence number, and does not allow a reassignment.
Case (4) does not affect correctness, since $findConsState()$ returns a state that is based on $3f + 1$ processors and gives no privilege to the primary's *rep*[].
Case (5) is proved by the observation that macro *acceptPPrepMsg*() requires that the q_m request matches the client's request and another $3f + 1$ processors' requests. Thus it is impossible for p_m to force a different request than that of the client.

We conclude that Condition (a) of $\mathcal{RT}$ is not violated.

Condition (b) is not violated – This condition is derived from the proofs of Lemma 6.4.12 and Condition (a), since every request that we examine is executed according to the order set by the primary, or if not set properly, a request is not executed and there is no progress in *rLog*.

Condition (c) is not violated – We have proved that any message sent by the primary or the other malicious processors cannot make correct processors to accept a different order of execution, since a consensus of $3f + 1$ is initially required on the order.

This concludes the proof. □

Claim 6.4.14. *Let R′ be a suffix of execution R as described in Claim 6.4.13, in which there was a view change. Every c ∈ R′ is a safe state and satisfies the conditions of $\mathcal{RT}$.*

Proof. Consider the prefix R of R' with CCSP S_u, and correct processors p_i, p_j, where p_j has a request $\tilde{q} \in rep_j[j].pendReqs$. We assume that a view change takes place and is concluded when the first processor p_k in the system has the new view $v = i$ (i.e., p_i is the new primary), and $allowService_k() = \mathsf{True}$. Every correct processor p_k that had the previous view, executes line 2 to find that $(rep_k[k] \neq \perp)$ is True, and $(Alg9.getView(i) \neq prim) = \mathsf{True}$, thus setting $rep_k[k].viewChange = \mathsf{True}$. Note that any subsequent iteration does not allow line 2 to modify this line since $rep_k[k].viewChanged \neq \mathsf{False}$. There are now two cases.

Case 1 – Primary p_i is correct. This implies that once p_i has $Alg9.getView_i(i) = i$ it detects a change of view and sets $rep_i[i].viewChange = \mathsf{True}$. This also satisfies line 4. Since every correct p_k has $rep_k[k].viewChanged = \mathsf{True}$ and propagates this via line 25, within $O(1)$ asynchronous rounds $\forall p_k \in P(rep_k[k].viewChanged = \mathsf{True})$, and line 5 holds.

The primary executes *renewReqs*() on a set of $4f + 1$ processors that have view i, and have their $viewChange = \mathsf{True}$. For $\tilde{q}$ of processor p_j, this implies that if was prepared by at least one processor, then within the $4f + 1$ processors, there exist $f + 1$ correct ones that have prepared $\tilde{q}$. This is due to the intersection of the $4f + 1$ that p_i chooses with the $3f + 1$ required that prepared the request. Since these are more than the malicious, p_i can safely find the prepared requests even in the presence of f colluding processors. Operator *renewReqs*() creates a new pre-prepare message for each prepared requests and adds the commit message to the *reqQ*. Operator *findConsState*() makes sure that the state is consistent with the contents of *reqQ* and *rLog*. The primary may use these to catch up with the replication if it is not up-to-date

(i.e., in a state transfer way. Thus p_i reaches to a $rep_i[i]$ that does not omit requests that may have been prepared in the previous view with a sequence number sq, by adding them to the current view with the same sq. It then propagates this $rep_i[i]$ with $rep_i[i].viewChanged = \mathsf{False}$ (lines 6 and 26).

At the side of correct p_j, $rep_j[j].viewChanged = \mathsf{True}$ until the primary p_i sends a consolidated state. The conditions of line 7 eventually apply for every correct processor, since they all detect a view changed, and should see within $O(1)$ asynchronous rounds of p_i's actions that $(rep_j[j].\langle viewChanged, prim\rangle = rep_j[i].\langle \mathsf{False}, prim\rangle)$. If $4f+1$ processors have the same view, then p_j calls $checkNewVState_j()$. Essentially it checks the procedures done by p_i at line 6 to check the correctness of $rep_i[i]$. Upon success $rep_i[i]$ is adopted.

Case 2 – Primary $\mathbf{p_i}$ is malicious. In this case the primary may not progress again, but this will cause a second view change. A correct processor p_j does not change $rep_j[j].viewChanged = \mathsf{True}$ until the first response of the primary. It also does not perform replication and accept requests. By the reasoning stated for Case 1, and that p_j bases their acceptance of the primary's state on $checkNewVState_j()$ and $4f + 1$ processors' states, it follows that p_j's state cannot take up any corrupt values proposed by the faulty primary.

Since in both cases every correct processor takes the correct state without loss of requests, or malicious addition of others to the state, we conclude that the conditions of RT hold. □

By the above claims we conclude that Theorem 6.4.11 holds. □

6.5 Primary Monitoring

The primary is monitored by a view change mechanism, which employs a failure detector to decide when a primary is suspected and, thus, a view change is required. View change facilitates the liveness of the system, since if a malicious processor does not correctly progress the replication it is changed. We proceed to present the two parts of the module.

Algorithm 11: Self-stabilizing Failure Detector; code for processor p_i

1 **Constants:** T an integer threshold.

2 **Variables:** $beat[n]$ is an integer heartbeat array where $beat[j]$ corresponds to p_j's heartbeat and $beat[i]$ is unmodified and remains 0. $FDset$ is the set of processors that are responsive according to their heartbeat, cnt is a counter related to the primary or a proposed primary of p_i's current view. $primSusp[n]$ is a boolean array of {True/False} where $primSusp[j]$ indicates whether processor p_j suspects the primary of its current view or not. $curCheckReq$ the requests' set (of size at most σK) that is currently being checked for progress. $prim$ holds the most recently read primary from the view establishment module.

3 The token passing mechanism (Sec. 3) piggybacks $FDset[i]$ and $primSusp[i]$ when sent to other processors, and updates fields $FDset[j]$ and $primSusp[j]$ upon receipt of the token from p_j.

4 **Macro** $reset()$ sets all fields of $primSusp[\bullet]$ to False, set $curPendReqs$ to $\emptyset$ and $beat[\bullet]$ and $cnt[\bullet]$ to 0.

5 **Interface function:**

6 $suspected() = (|\{p_j \in P : (getView(j) = getView(i)) \wedge (primSusp[j] = \mathsf{True})\}| \geq 3f + 1))$

7 **Upon receipt of** $token_j$ **from** p_j **begin**

8 // *(Responsiveness check.)*

9 $beat[j] \leftarrow (beat[i] \leftarrow 0)$;

10 **foreach** $p_k \in P \setminus \{p_j, p_i\}$ **do** $beat[k] \leftarrow beat[k] + 1$;

11 $FDset \leftarrow \{p_\ell \in P : beat[\ell] < T\}$;

12 // *(Primary-progress check.)*

13 **if** $prim \neq getView(i)$ **then foreach** $p_j \in P$ **do** $reset()$;

14 $prim \leftarrow getView(i)$;

15 **if** $(Alg9.allowService() \wedge noViewChange())$ **then**

16 **if** $(j = prim)$ **then**

17 **if** $(\exists x \in curCheckReqs : x \notin getPendReqs()) \vee (curCheckReq = \emptyset)$ **then**

18 $(cnt[j], curCheckReq) \leftarrow (0, getPendReqs())$

19 **else** $cnt[j] \leftarrow cnt[j] + 1$;

20 **else if** $(prim = getView(j))$ **then** $primSusp[j] \leftarrow token_j.primSusp$;

21 **foreach** $\{p_k \in P \setminus \{p_{prim}\}\}$ **do** $\{cnt[k] \leftarrow 0\}$; // *reset all counters except primary's*;

22 **if** $prim = i$ **then** $cnt[i] \leftarrow 0$;

23 **if** $(\neg primSusp[i])$ **then** $primSusp[i] \leftarrow ((p_{prim} \notin FDset) \wedge (cnt[i] > T))$;

24 **else if** $(\neg Alg9.allowService())$ **then** $reset()$;

6.5.1 Failure Detection

We base our FD (Algorithm 11) on the token-passing mechanism described in Section 3.3, and follow the approach of [53] to check both: (i) the responsiveness of processors, (ii) that the primary p_{prim} is progressing the state machine.

Responsiveness check (lines 9–11). Every processor p_i maintains a heartbeat integer

counter for every other processor p_j. Whenever processor p_i receives the token from processor p_j over their data link, processor p_i resets p_j's counter to zero and increments all the counters associated with the other processors by one, up to a predefined threshold value T. Once the heartbeat counter value of a processor p_j reaches T, the FD of processor p_i *suspects* p_j to be unresponsive. In other words, the FD at processor p_i considers processor p_j to be active if and only if the heartbeat associated with p_j is strictly less than T. Note that malicious processors can intentionally remain unresponsive and then become responsive again. A correct processor cannot distinguish this behavior from inaccuracies of the FD (due to packet delays) that make a correct processor appear briefly unresponsive, but eventually appears as responsive when its delayed packets are received. Nevertheless, the use of the FD is to suggest which processors are responsive at a given time.

Primary progress check (lines 13–24). To achieve liveness, processors need to be able to check whether the primary is progressing the state machine by imposing order on the requests received as instructed by Algorithm 10. The responsiveness FD can only suspect a non-responsive primary. A faulty primary can be very responsive at the level of packets, and thus evade suspicion by sending messages that are unrelated to the requests of the clients. A different type of detection is required, that examines how the primary is proceeding with assigning sequence numbers to requests.

To this end, the primary's progress check uses the heartbeat FD to provide liveness but not safety as follows. The failure detector of p_i holds a set of requests *curCheckReq* that it drew from the replication module the last time that progress was detected by the primary. Processor p_i uses the interface $getPendReqs_i()$ (in line 17) to check for progress in relation to $curCheckReqs_i$. Since $getPendReqs_i()$ returns the requests that are not prepared but are known by $3f + 1$, the primary cannot be suspected for requests that existed in the system or that are introduced by malicious processors. Note that in the case where a view change has taken place, *getPendReqs*() returns 'ViewChange' that implies that the progress requested by the new primary is to return a new consolidated state as detailed in Section 6.4.

If at least one request of *curCheckReq* is removed from the currently pending requests, then the counter $cnt_i[prim]$ is reset to 0, and *curCheckReqs* is updated by *getPendReqs* (line 18). Note that the approach is self-stabilizing, since if *curCheckReqs* is the result of an arbitrary initial state this is cleaned within one iteration of the

algorithm. If there was no progress, then the primary's counter *cnt*[] is incremented (line 19). Due to the possibility of corruption, we reset the *cnt*[] of every non-primary in every round (line 21). Line 23 determines whether a primary p_j is locally suspected if its counter is beyond the threshold for responsiveness or request progress and sets $primSusp[j] = \mathsf{True}$. The interface *suspected*() (line 6) considers the primary as suspected if $3f + 1$ processors consider it as suspected.

Suspicion of primary. If processor p_i suspects its primary, i.e., $primSusp[i] = \mathsf{True}$, then while the primary does not change, $primSusp[i]$ remains True. This is to prevent a malicious primary from manipulating the failure detectors of correct processors continuously and ensuring that once a primary has been suspected by at least $2f + 1$ processors at most once, it will be changed. So this "forces" a malicious primary to either be suspected by correct processors and cause a view change or to progress the replication. Note that in the worse case, there are only f malicious primaries consecutively before reaching to a correct one. This is not a perfect failure detector since it allows for $2f$ correct processors to err on suspecting the primary once or more. We thus form the following failure detection assumption to complement Liveness Assumption 1.

Liveness Assumption 2. *Consider an execution R of Algorithm 10 starting at a safe system state with a correct primary p_i. Then throughout R, no more than $2f + 1$ correct processors may suspect p_i, i.e., $\nexists S \subseteq C : (\forall p_j \in S\ (primSusp_j = \mathsf{True})) \wedge (|S| \geq 2f + 1)$.*

Correctness. We prove that Algorithm 11 is self-stabilizing. In particular we prove the following Lemma.

Lemma 6.5.1. *Consider an execution R of Algorithm 11 starting in an arbitrary state. Then only at most one correct primary p_{prim} may be suspected as a result of stale information.*

Proof. Processor p_i running Algorithm 11 may suspect correct p_{prim} due to corruption in $primSusp_i, beat_i[prim]$ or $cnt_i[prim]$. Starting in an arbitrary state may lead to $primSusp = \mathsf{True}$, either by corruption of *primSusp*, or of assignment due to corruption by the variables checked at line 23. A falsely initiated *beat*[*prim*] becomes False again upon the first receipt of the counter of p_{prim}, i.e., line 9.

The primary progress check can have corruption in *curCheckReqs* and thus check for requests that were never issued (line 18), but this will be deemed as progress. If $cnt[prim] > T$ then this is amended only if p_{prim} progresses the replication, such the

p_i's replication module presents a different requests set to *curCheckReqs* (per line 18's condition). By Claim 6.4.12 this is done in $O(1)$ asynchronous rounds.

If the above stale information cause a view change (by *suspected*() = True then we note that by the initialization of *primSusp*, $beat_i[\bullet]$ and $cnt_i[\bullet]$, line 21 sets all these variables to default 0 values. The result follows, since for any new primary the counters start from 0, *primSusp* = False, and the pending requests set is set to the $\emptyset$. This completes the proof. □

6.5.2 View Change upon Suspected Primary

Algorithm outline. A processor propagates messages about which processors have reported to require a view change. If $3f + 1$ processors appear to have suspected the primary, then the processor stops providing service even if itself has not suspected the primary itself. The above guarantees that since f of $3f + 1$ processors may be malicious, at least $2f + 1$ correct processors have firmly suspected the primary. The replication mechanism is left with $3f$ processors, which is not enough to make progress, so the view change is forced upon the system by the $2f+1$ correct processors who are the majority of correct processors. The view change is initiated upon seeing that the intersection of those that require view change becomes $4f + 1$.

Variables. The local state of processor p_i for this algorithm is the tuple $vcm_i[n] = \langle vStatus, prim, needChange, needChgSet \rangle$ where $vcm_i[i]$ holds p_i's values and $vcm_i[j]$ holds p_j's last value copy of $vcm_j[j]$ that was sent via the communication in lines 16–17. Variable $vStatus \in \{\text{OK}, noService, vChange\}$ is the status of the processor regarding the requirement to change view, where OK requires no change, noService is a stop to service provision, and vChange is the status when the service is stopped and $4f + 1$ processors are found to require a view change. Variable *prim* holds the reading of primary in the last iteration of the algorithm, the boolean *needChange* is True when p_i requires a change, and *needChgSet* is that set of processors that appear to p_i as requiring a view change. DEF_STATE is a default value for $vcm = \langle \text{OK}, getView(), \text{False}, \emptyset \rangle$.

Macros and Functions. Algorithm 12 uses the macros and provides the interface functions that follow.

- Macro *cleanState*() imposes the DEF_STATE to each entry of $vcm_i[\bullet]$.

Algorithm 12: Self-stabilizing View Change; code for processor p_i

1 **Variables:** Tuple $vcm[n] = \langle vStatus, prim, needChange, needChgSet \rangle$ where $vStatus \in \{\mathsf{OK}, noService, vChange\}$, $needChange$, is a boolean in $\{\mathsf{True}, \mathsf{False}\}$ and $needChgSet$ a set of processors that appear to require a view change. $vcm[i]$ holds p_i's values and $vcm[j]$ holds p_j's last gossiped value to p_i. DEF_STATE is a default value for $vcm = \langle \mathsf{OK}, getView(), \mathsf{False}, \emptyset \rangle$.

2 **Macros:** $cleanState() = \{$**foreach** $p_j \in P$ **do** $vcm[j] \leftarrow$ DEF_STATE;$\}$

3 $supChange(int\ x) =$ **return** $(\exists X \subseteq P : (\forall p_j, p_{j'} \in X : vcm[j].prim = vcm[j'].prim) \wedge (|\bigcap_{p_k \in X} vcm[k].needChgSet| \geq 3f + 1) \wedge (|X| \geq x))$

4 **Interface function:** $noViewChange() = (vStatus = \mathsf{OK})$;

5 **do forever begin**

6 **if** $prim \neq getView(i)$ **then** $cleanState()$;

7 $(prim, needChange) \leftarrow (getView(i), Alg11.suspected())$;

8 **if** $(Alg9.allowService())$ **then**

9 **if** $(prim = getView(i)) \wedge (vStatus \neq \mathsf{vChange})$ **then**

10 $needChgSet \leftarrow needChgSet \cup \{p_j \in P : getView(i) = getView(j) \wedge vcm[j].needChange = \mathsf{True}\}$;

11 **if** $((|\{vcm[j].vStatus = \mathsf{noService}\}_{p_j \in P}| < 2f + 1))$ **then** $vStatus \leftarrow OK$;

12 **if** $(vStatus = \mathsf{OK}) \wedge (supChange(3f + 1))$ **then** $vStatus \leftarrow \mathsf{noService}$;

13 **else if** $(supChange(4f + 1))$ **then** $vStatus \leftarrow \mathsf{vChange}$; $viewChange()$;

14 **else if** $(prim = getView(i)) \wedge (vStatus = \mathsf{vChange})$ **then** $viewChange()$;

15 **else** $cleanState()$;

16 **foreach** $p_j \in P$ **do send** vcm;

17 **Upon receive** m **from** p_j **do** $vcm[j] \leftarrow m$;

- Macro $supChange(x)$ returns True if there is a set of processors size x with the same primary, and this set supports a view change, and also each member of the set sees an intersection of $needChgSet$ sets of size at least $3f + 1$.
- Interface function $noViewChange()$ returns True if the status is not noService nor vChange, i.e., $status = \mathsf{OK}$.

Detailed description. Primary p_i first checks if the primary has changed based on the current reading of $getViews()$ (line 9), and the previous reading of this function. Processor p_i resets the status and variables only if the status is not yet *vChange*. Processor p_i executing Algorithm 9 first reads the FD (line 7), and then checks whether the view establishment module returns a current view (line 8).

If any of these two conditions fail, p_i adds processors that have their *needChange* flag to True to the *needChgSet* (line 10). Line 11 resets the status to OK if there is no support to change the view, and it copes with arbitrary changes to the status.

The algorithm then moves from status OK to noService if there are more than $3f + 1$ processors in *needChgSet* (line 12). If the processor sees $4f + 1$ processors in *needChgSet* it moves to status *vChange* and calls the *viewChange*() interface function of the view establishment module to initiate the view change procedure to the next view (line 13). While a view does not change, it holds a set of processors *needChgSet*, and it adds to this any processors that report having seen *suspected*(). While in status *vChange* and the view not having changed, the algorithm renews its request to the view establishment module (line 14). Line 15 captures the case where the view change has finished and the local variables are set to their defaults. Lines 16 and 17 implement the communication between the processors.

Correctness. We define the following task description for the view change module. An execution R is legal with respect to the view change task $\mathcal{VT}$, if:
(i) There are no calls to *vChange*() by correct processors if the Liveness Assumptions 1 and 2 hold for primary $p_i \in C$.
(ii) If the current primary p_m is suspected by $2f + 1$ correct processors, then a view change is initiated that results to a new view $m + 1$.

Theorem 6.5.2. *Consider an execution R starting in an arbitrary state, that respects Liveness Assumptions 1 and 2. Within $O(n)$ asynchronous rounds the system reaches an execution suffix R' which belongs to the legal executions of $\mathcal{VT}$.*

Proof. We prove the result with the following two lemmas. Lemma 6.5.3 proves that Condition (i) (that defines the convergence property of $\mathcal{VT}$) eventually holds, and correspondingly Condition (ii) that defines the closure property of $\mathcal{VT}$ is shown by Lemma 6.5.4 to hold within $O(n)$ rounds.

Lemma 6.5.3 (Convergence – Condition (i) of $\mathcal{VT}$)**.** *Consider an execution $R = R' \circ R''$ starting in an arbitrary state. If there is a prefix R' of R' that has at most one call to viewChange() that is completed by the View Establishment module, then Condition (i) of $\mathcal{VT}$ holds throughout the suffix R'' of R'.*

Proof. We assume that a set of processors $p_i \in C$ calls $viewChange_i()$ due to stale information in the state of $vcm_i[\bullet]$, and this causes a view change to eventually initiate at the View Establishment module. Line 14 renews the request in every iteration until the View Establishment module eventually starts the procedure to change the view. Upon detection of a change of primary by line 6 the $cleanState_i()$

procedure imposes the DEF_STATE. This ensures that if the new primary is not suspected, no variable of $vcm_i[i]$ will lead to a view change on p_i's behalf.

Since a view change cannot take place if $3f+1$ correct processors do not converge to this (see automaton predicate $(0,1)$ and $(1,1)$ of Algorithm 9), then there exist at least $3f+1$ processors that exchange correct and default $vcm[\bullet]$ information when the view change takes place or after this. By this, *supChange*() cannot hold neither due to initial corrupt information, nor due to existing. This is because a view change at Algorithm 9 already takes $O(1)$ and thus any messages with stale $vcm[\bullet]$ are received before line 6 is run. This implies that the new primary initiates at p_i's view change state with a DEF_STATE. □

Following the above Lemma 6.5.3, we assume that the primary is changed in a prefix R' of an execution $R = R' \circ R''$, and R'' starts when the first correct processor p_i that gets the new primary via $Alg9.getView_i(i)$ executes line 6.

Lemma 6.5.4 (Closure – Condition (ii) of $\mathcal{VT}$)**.** *Consider an execution prefix R'' where the Liveness Assumptions 1 and 2 hold throughout. Then either the primary p_k is correct and is the primary throughout R'', or p_k acts in a way that violates the liveness assumptions and a view change is initiated.*

Proof. We study the two cases.

Primary $\mathbf{p_k} \in C$. Since during p_k's primacy, the Liveness Assumptions are never violated, this that there can be up to $2f$ correct processors p_i that suspect p_k at some point (by a call to $Alg11.suspected()$) and add themselves to *needChgSet*, along with the other $2f-1$ correct processors, and possibly f malicious processors that want to overthrow p_k. Thus there cannot be a call to *supChange*() that will return True, since the last condition that $|X| \geq 3f+1$ is never satisfied for the correct processors. Hence, p_k remains the primary throughout.

Primary $\mathbf{p_k} \notin C$ and violates Liveness Assumptions 1 and 2. If p_k violates the Liveness Assumptions then this implies it was suspected at least once by $2f+1$ processors. This does not still cause a change of primary. Another f (malicious or correct) processors are required to report a suspicion of the primary. The primary though must make progress, otherwise it will be suspected by more correct processors, so it is forced to provide some liveness.

Nevertheless, if $supChange_i(3f+1)$ is called by correct p_i (line 12) an returns True, then this is True at every correct processor, because of the need for an intersection

of *needChgSet* at $3f + 1$ processors. Assume that $2f + 1$ correct processors have exchanged their *needChgSet* and find $3f + 1$ common processors in *needChgSet*. Then they set *vStatus* $\leftarrow$ NoService and by line 11 this is irrevocable since they are at least $2f + 1$ and so the condition of this line is not met to set the *vStatus* back to OK.

The progress of replication is dependent on decisions with $3f + 1$ support, but at this point only at most $3f$ continue to have *vStatus* = OK. Thus there is no more progress, and the primary is eventually suspected by the FDs of the other correct processors, since it will eventually seize to have available sequence numbers (violation of Liveness Assumption 1). Within $O(1)$ asynchronous rounds, every correct processor proceeds to build a *needChgSet* of size $3f + 1$. This satisfies $supChange_i(4f + 1)$ for every correct processor p_i and thus enter *vStatus* = *vChange*.

This is not changed until the view changes (line 6), and thus, since view establishment also requires $4f + 1$ processors to proceed, at least $3f + 1$ correct processors are lead to view changes which suffices to drive the rest as well, either by satisfying *supChange*() or through the View Establishment module (automaton predicates $0, 1$ and $1, 1$). By showing that a view change was initiated we complete this proof, since installing the new view is handled by other modules. □

The two lemmas combined prove that we reach to the execution suffix R' that always satisfies the $\mathcal{VT}$ specification, and hence the result. □

6.6 Extensions

6.6.1 Relaxing the Assumptions for View Establishment

Recall that the convergence proof for view establishment (Section 6.3) assumed malfree executions (that is, a view is guaranteed to be established in the absence of Byzantine behavior). We now discuss how we can relax this assumption.

Tolerating crash failures during view establishment. We can first weaken this assumption by allowing establishment in the presence of faulty processors that can be non-responsive (either intentionally or they have crash-failed). Such assumptions can give rise to problematic (extreme, but still plausible) scenarios as the one described in Lemma 6.3.9. We can settle such issues, by deploying our responsiveness FD (note that the FD detector is not used in Section 6.3) and have the *liveness assumption* that a majority (i.e., $3f + 1$) of correct, mutually responsive processors,

support the stable view. One can see that the proofs written for the view establishment convergence under the above liveness assumption and in conjunction with the responsiveness FD lead to convergence in the presence of unresponsive (or crashed) faulty processors while dealing with stale information.

Tolerating malicious behavior during view establishment. As expected, to be able to establish a view in the presence of both malicious behavior *and* stale information, stronger assumptions are required (cf. [48]).

Such a liveness assumption would be *k-admissibility:* Assume a ratio of k between the fastest token round trips in a data link to the fastest non-faulty processors and the slowest non-faulty processors. In other words, under k-admissibility, a ratio of k is assumed between the fastest and the slowest non-faulty processor (when a fast correct processor exchanges k tokens, then at least one token is exchanged by the slowest correct processor). We can consider an *event-driven FD* implemented as follows: When p_i broadcasts a message over the data-link token to all its neighbors, p_i resets a counter for each attached link and starts counting the number of data-link tokens arriving back to it, until at least $n - 2f$ distinct counters reach a value that is at least k. Then, under k-admissibility, processor p_i can safely assume that values from all non-faulty processors (i.e., $n - f$) arrived.

In other words, given k, the above simple FD can be tuned to ensure correct processors get replies from all correct processors. (In a synchronous system k would be small, and as asynchrony increases, k would need to increase.) Also, if a solution takes decisions based on a threshold (fraction) of processors (as in our solution), then k can be reduced, hence making the liveness assumption requiring "less synchrony". In some sense, this FD can be considered as an *on-demand* failure detector than can be tuned based on the "level" of synchrony of the system. The introduction of this FD allows us, for the first time, to avoid the constant overhead of background bookkeeping.

The event-based failure detector allows us to restate our view establishment convergence proof (Theorem 6.3.10), provided that we have k-admissibility. This ensures that we can find a consistency set Σ in the presence of malicious behavior. Since all the other results are proved for mal-admissible executions, the correctness becomes immediate once we circumvent the result of Lemma 6.3.9. This is what we achieve with the proposed failure detector, and so we have the following result.

Corollary 6.6.1. *Consider a mal-admissible and k-admissible execution of the View Establishment module enhanced with the event-based failure detector and starting in an arbitrary state. Within $O(n)$ asynchronous rounds the system reaches a state in which there is a stable view v. Moreover, every correct processor eventually adopts v.*

6.6.2 Optimality

Our solution presented in Section 6.4 assumes that at most $f = (n-1)/5$ of the processors are faulty. As discussed there, to establish a view, we require the agreement of $n - f$ processors (i.e., $4f + 1$), to adopt an established view and correct processors to sustain it, $\max\{n - 2f, n/2\}$ processors must support it (i.e., $3f + 1$), whereas in order to make decisions regarding the progress of the replication (servicable view) we need $\lceil (n - f)/2 \rceil + f$ processors (i.e., $3f + 1$) to agree (strong majority of correct processors). This is also the threshold we need for a primary to be suspected, and $n - f$ to proceed to change the view.

We have parameterized our solution so that these thresholds can be adjusted for different ratios between faulty and correct processors (as explained, we used $n = 5f + 1$ as it makes the presentation easier to follow). In particular, for the *optimal resilience* (cf. [11]) ratio of $f' = (n-1)/3$, and going over the correctness proofs of our solution, it follows that establishing a view will require agreement of $2f' + 1$ processors, adopting a view will require the support of $\frac{3}{2}f' + 1$ processors, whereas serviceabilty requires $2f' + 1$ processors to agree (the same threshold required in [38]).

6.7 Chapter Summary

We presented the first self-stabilizing BFT algorithm based on an implementable failure detector to provide liveness. The approach is modular and allows for suggested extensions and also to achieve optimal resilience. The result paves the way towards self-stabilizing distributed blockchain system infrastructure.

Chapter 7

Conclusions and Future Work

7.1 Summary

In this thesis we study important problems in the field of self-stabilizing State Machine Replication for asynchronous message-passing systems, and we address them by providing algorithmic solutions with proven guarantees. In particular, we present the first, to our knowledge, practically-self-stabilizing virtually synchronous SMR protocol for a static set of crash-prone processors, the first self-stabilizing reconfiguration algorithm for a dynamic crash-prone set of processors, and the first self-stabilizing Byzantine-Fault-Tolerant SMR that is based on failure detectors. We briefly summarize the contributions.

Our stabilizing virtually synchronous SMR protocol (Chapter 4) satisfies the guarantee of practically-self-stabilization, a newer notion of stabilization that has recently attracted research interest. The service works on a static set of crash-prone processors. We build our SMR protocol by combining several stabilizing components of independent interest. We first present a practically-self-stabilizing labeling and counter scheme. Our counter extension is modular and it is more efficient in relation to the recently proposed one of [26]; it only requires sending a pair of labels rather than a vector of labels. The counter is readily extendable to implement a multi-writer multi-reader shared memory emulation. Together with a suitable proposed multicast service and a failure detector to provide membership, we construct the first practically-self-stabilizing virtually synchronous replicated state machine implementation, with proven guarantees.

In Chapter 5, we present the first self-stabilizing reconfiguration scheme. It automatically recovers from transient faults arising from temporary violations of the

predefined churn rate, or the unexpected activities of processors and communication channels. Our blueprint for self-stabilizing reconfigurable distributed systems can withstand a temporal violation of such assumptions, and recover once conditions are resumed. It achieves this with only a bounded amount of local storage and message size. While non-stabilizing systems do not offer guarantees once the quorum configuration is lost, our solution guarantees recovery to a correct behavior after the collapse of the quorum system. Indeed, the solution can even bootstrap in the case where processors have conflicting knowledge of the configurations and none of the members of the known configurations are active. Our scheme enables self-stabilizing services to run on a dynamic set of processors, and by this it enables the deployment of long-lived self-stabilizing services. We address the problems of when to initiate a reconfiguration and of how to choose the configuration set as application-based decisions. We thus provide the application with the capability to address its needs via suitable interfaces to evaluate the current configuration, and propose a new configuration.

In Chapter 6, we provide a novel asynchronous self-stabilizing Byzantine-fault-tolerant replicated state machine service. Diverging from the approach of the existing work on self-stabilizing BFT [39], we do not use clock synchronization and timeouts, but rather, we base our solution on a self-stabilizing failure detector that we detail, and for which we provide a proposed implementation. This is the first work to combine self-stabilization and BFT replication that is based on failure detectors, thus encapsulating weaker synchronization guarantees than [39]. To relax the constraints on Byzantine behavior that are required due to the impossibility results, we propose a series of optimizations. To this end we propose a novel tunable event-based (unreliable) failure detector.

The thesis employs a novel approach toward the design and proof of the self-stabilizing algorithms proposed. In particular, in the work in Chapter 5, and more apparently in the work on BFT SMR (Chapter 6), we structure our solutions to impose a lockstep progress of processors through an automaton's states. We complement our results with guarantees given in suitable metrics. The results have bounded requirements in local memory and message sizes.

7.2 Future Directions and Objectives

This line of work leads to some challenging future research directions. A natural ultimate aim of this work would be to deploy a self-stabilizing reconfigurable Byzantine-tolerant SMR service. To this end one would first need to have a firm grasp of a reconfigurable non-stabilizing version of BFT. Unfortunately, at the moment, the work of even non-stabilizing reconfigurable BFT is scanty. This is underlined in Section 2.3.2.

This work has significant potential applications to emerging topics. An extension of our current work, especially of the self-stabilizing PBFT, is to construct self-stabilizing blockchain system infrastructure, and study the impact of the self-stabilization property to the system's performance in relation to existing non-stabilizing approaches. Another direction is to use our approach to create coordinated self-stabilizing cloud-based microservices and thus guarantee eventual consistency to facilitate robust higher-level services [137].

One open issue arising from the reconfiguration scheme of Chapter 5, is to study whether we can have *non-suspending* self-stabilizing reconfigurable services. Since we establish liveness for our work using failure detection, it is also seems natural to establish what is the weakest failure detector required to solve such problems given added difficulty of transient faults.

Another important direction is to put the theoretical correctness guarantees of the completed works to test. Implementations of these algorithms should be prototyped and experimentally evaluated with proper simulations. A possible evaluation can move along the following non-exhaustive list of criteria.

- Testing the stabilization time of the modules given different types of initial corrupt states.
- Testing scalability, i.e., how the number of processors affects stabilization time and delays for view installation (in the virtual synchrony SMR service) and for reconfiguration.
- Test against non-stabilizing implementations to see the actual cost of stabilization. This will provide an indication of the tradeoffs between having a valuable fault-tolerant feature like self-stabilization, against the increased cost in message exchange and some (possible) delays in reconfiguration time.

The algorithms presented can certainly accept many optimizations, with the easiest being to make stabilization related messages more infrequent at the cost of slower stabilization. In this sense the cost of stabilization is *tunable*. One may choose to reduce stabilization-related communication at the expense of slower stabilization. Of course, a bounded increase in communication costs cannot capture the benefits of having a service that can automatically recover from state corruption. As long as the necessary infrastructure and the program are unaffected, the proposed services will eventually return the system to its desired behavior, contrary to other solutions that can provide no guarantees for such scenarios. One may also study the case for making the above solutions *adaptive*, so that stabilization-related messages become very frequent only when the system is stabilizing (e.g., when there is brute reconfiguration), but this may not always be possible.

Bibliography

[1] B. Charron-Bost, F. Pedone, and A. Schiper, Eds., *Replication: Theory and Practice*, ser. Lecture Notes in Computer Science, vol. 5959. Springer, 2010. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11294-2

[2] R. van Renesse and R. Guerraoui, "Replication techniques for availability," in *Replication: Theory and Practice*, ser. Lecture Notes in Computer Science, B. Charron-Bost, F. Pedone, and A. Schiper, Eds., vol. 5959. Springer, 2010, pp. 19–40. [Online]. Available: https://doi.org/10.1007/978-3-642-11294-2_2

[3] K. Birman, D. Freedman, Q. Huang, and P. Dowell, "Overcoming cap with consistent soft-state replication," *Computer*, vol. 45, no. 2, pp. 50–58, Feb 2012.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. [Online]. Available: http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx

[5] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[6] M. Schneider, "Self-stabilization," *ACM Computer Survey*, vol. 25, no. 1, pp. 45–67, Mar. 1993. [Online]. Available: http://doi.acm.org/10.1145/151254.151256

[7] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998. [Online]. Available: http://doi.acm.org/10.1145/279227.279229

[8] K. Birman, "A history of the virtual synchrony replication model," in *Replication*, ser. Lecture Notes in Computer Science, B. Charron-Bost, F. Pedone, and A. Schiper, Eds. Springer Berlin Heidelberg, 2010, vol. 5959, pp. 91–120. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11294-2_6

[9] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Trans. Comput. Syst.*, vol. 5, no. 1, pp. 47–76, Jan. 1987. [Online]. Available: http://doi.acm.org/10.1145/7351.7478

[10] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: a comprehensive study," *ACM Computing Surveys (CSUR)*, vol. 33, no. 4, pp. 427–469, 2001.

[11] I. Abraham, D. Malkhi *et al.*, "The blockchain consensus layer and bft," *Bulletin of EATCS*, vol. 3, no. 123, 2017.

[12] M. Herlihy, "Blockchains and the future of distributed computing," in *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, 2017, p. 155. [Online]. Available: http://doi.acm.org/10.1145/3087801.3087873

[13] I. Abraham and D. Malkhi, "Bvp: Byzantine vertical paxos," *Distributed Cryptocurrencies and Consensus Ledgers (DCCL)*, 2016.

[14] S. Ølnes, J. Ubacht, and M. Janssen, "Blockchain in government: Benefits and implications of distributed ledger technology for information sharing," *Government Information Quarterly*, vol. 34, no. 3, pp. 355–364, 2017. [Online]. Available: https://doi.org/10.1016/j.giq.2017.09.007

[15] K. Birman and H. Sohn, "Hosting dynamic data in the cloud with isis2 and the ida DHT," in *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems, TRIOS@SOSP 2013, Farmington, PA, USA, November 3, 2013*, 2013, pp. 10:1–10:15. [Online]. Available: http://doi.acm.org/10.1145/2524211.2524212

[16] S. Jha, J. Behrens, T. Gkountouvas, M. Milano, W. Song, E. Tremel, S. Zink, K. Birman, and R. van Renesse, "Building smart memories and high-speed cloud services for the internet of things with Derecho," in *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24 - 27, 2017*, 2017, p. 632. [Online]. Available: http://doi.acm.org/10.1145/3127479.3134597

[17] S. Gilbert, N. A. Lynch, and A. A. Shvartsman, "Rambo: a robust, reconfigurable atomic memory service for dynamic networks," *Distributed Computing*, vol. 23, no. 4, pp. 225–272, 2010. [Online]. Available: http://dx.doi.org/10.1007/s00446-010-0117-1

[18] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer, "Dynamic atomic storage without consensus," *J. ACM*, vol. 58, no. 2, p. 7, 2011. [Online]. Available: http://doi.acm.org/10.1145/1944345.1944348

[19] S. Dolev, *Self-Stabilization*. MIT Press, 2000.

[20] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, vol. 17, no. 11, pp. 643–644, 1974.

[21] J. Brzezinski, M. Szychowiak, and D. Wawrzyniak, "Self-stabilization in distributed systems-a short survey," in *Foundations of Computing and Decision Sciences*, 2000.

[22] M. G. Gouda, R. R. Howell, and L. E. Rosier, "The instability of self-stabilization," *Acta Informatica*, vol. 27, no. 8, pp. 697–724, 1990. [Online]. Available: http://dx.doi.org/10.1007/BF00264283

[23] L. Lamport, "1983 invited address solved problems, unsolved problems and non-problems in concurrency," in *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing (PODC'84)*. New York, NY, USA: ACM, 1984, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/800222.806731

[24] I. Salem and E. M. Schiller, "Practically-self-stabilizing vector clocks in the absence of execution fairness," in *Proceedings of the 6th International Conference on Networked Systems (NETYS'18)*, 2018, technical report at http://arxiv.org/abs/1712.08205.

[25] J. E. Burns, M. G. Gouda, and R. E. Miller, "Stabilization and pseudo-stabilization," *Distrib. Comput.*, vol. 7, no. 1, pp. 35–42, Nov. 1993. [Online]. Available: http://dx.doi.org/10.1007/BF02278854

[26] P. Blanchard, S. Dolev, J. Beauquier, and S. Delaët, "Practically self-stabilizing paxos replicated state-machine," in *Proceedings of the 2nd International Conference on Networked Systems (NETYS'14)*, 2014, pp. 99–121. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-09581-3_8

[27] N. Alon, H. Attiya, S. Dolev, S. Dubois, M. Potop-Butucaru, and S. Tixeuil, "Practically stabilizing SWMR atomic memory in message-passing systems," *Journal of Computer and System Sciences*, vol. 81, no. 4, pp. 692–701, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.jcss.2014.11.014

[28] S. Dolev, R. I. Kat, and E. M. Schiller, "When consensus meets self-stabilization," *J. Comput. Syst. Sci.*, vol. 76, no. 8, pp. 884–900, 2010. [Online]. Available: https://doi.org/10.1016/j.jcss.2010.05.005

[29] K. P. Birman, T. A. Joseph, T. Räuchle, and A. El Abbadi, "Implementing fault-tolerant distributed objects," *IEEE Trans. Software Eng.*, vol. 11, no. 6, pp. 502–508, 1985. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.1985.232242

[30] R. V. Renesse, K. P. Birman, and S. Maffeis, "Horus: A flexible group communication system," *Communications of the ACM*, pp. 76–83, 1996.

[31] A. S. Tanenbaum and M. van Steen, *Distributed systems - principles and paradigms (2nd edition)*. Pearson Education, 2007.

[32] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *Journal of the ACM*, vol. 42, no. 1, pp. 124–142, Jan. 1995. [Online]. Available: http://doi.acm.org/10.1145/200836.200869

[33] L. Lamport, "On interprocess communication. part I: basic formalism," *Distributed Computing*, vol. 1, no. 2, pp. 77–85, 1986. [Online]. Available: https://doi.org/10.1007/BF01786227

[34] N. A. Lynch and A. A. Shvartsman, "Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts," in *Digest of Papers of the 27th Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, 1997, pp. 272–281. [Online]. Available: http://dx.doi.org/10.1109/FTCS.1997.614100

[35] P. M. Musial, N. C. Nicolaou, and A. A. Shvartsman, "Implementing distributed shared memory for dynamic networks," *Commun. ACM*, vol. 57, no. 6, pp. 88–98, 2014. [Online]. Available: http://doi.acm.org/10.1145/2500874

[36] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982. [Online]. Available: http://doi.acm.org/10.1145/357172.357176

[37] M. Correia, G. S. Veronese, N. F. Neves, and P. Veríssimo, "Byzantine consensus in asynchronous message-passing systems: a survey," *IJCCBS*, vol. 2, no. 2, pp. 141–161, 2011. [Online]. Available: https://doi.org/10.1504/IJCCBS.2011.041257

[38] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, 1999, pp. 173–186.

[39] A. Binun, T. Coupaye, S. Dolev, M. Kassi-Lahlou, M. Lacoste, A. Palesandro, R. Yagel, and L. Yankulin, "Self-stabilizing byzantine-tolerant distributed replicated state machine," in *Stabilization, Safety, and Security of Distributed Systems - 18th International Symposium, SSS 2016, Lyon, France, November 7-10, 2016, Proceedings*, 2016, pp. 36–53. [Online]. Available: https://doi.org/10.1007/978-3-319-49259-9_4

[40] S. Dolev and J. L. Welch, "Self-stabilizing clock synchronization in the presence of byzantine faults," *J. ACM*, vol. 51, no. 5, pp. 780–799, 2004. [Online]. Available: http://doi.acm.org/10.1145/1017460.1017463

[41] S. Dolev, C. Georgiou, I. Marcoullis, and E. M. Schiller, "Self-stabilizing virtual synchrony," in *Proceedings of the 17th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS 2015*, 2015, pp. 248–264. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-21741-3_17

[42] ——, "Practically-self-stabilizing virtual synchrony," *Journal of Computer and System Sciences*, vol. 96, pp. 50–73, 2018. [Online]. Available: https://doi.org/10.1016/j.jcss.2018.04.003

[43] I. Marcoullis, "Self-stabilizing middleware services," in *Proceedings of the Doctoral Symposium of the 17th International Middleware Conference, Middleware Doctoral Symposium 2016, Trento, Italy, December 13, 2016*, 2016, pp. 2:1–2:4. [Online]. Available: http://doi.acm.org/10.1145/3009925.3009927

[44] S. Dolev and T. Herman, "Superstabilizing protocols for dynamic distributed systems," *Chicago J. Theor. Comput. Sci.*, vol. 1997, 1997. [Online]. Available: http://cjtcs.cs.uchicago.edu/articles/1997/4/contents.html

[45] S. Dolev, E. Schiller, and J. L. Welch, "Random walk for self-stabilizing group communication in ad hoc networks," *IEEE Trans. Mob. Comput.*, vol. 5, no. 7, pp. 893–905, 2006. [Online]. Available: https://doi.org/10.1109/TMC.2006.104

[46] S. Dolev, C. Georgiou, I. Marcoullis, and E. M. Schiller, "Self-stabilizing reconfiguration," in *Networked Systems - 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings*, 2017, pp. 51–68. [Online]. Available: https://doi.org/10.1007/978-3-319-59647-1_5

[47] ——, "Self-stabilizing reconfiguration," in *Proceedings of the Posters and Demos Session of the 17th International Middleware Conference, Middleware Posters and Demos 2016, Trento, Italy, December 12-16, 2016*, 2016, pp. 13–14. [Online]. Available: http://doi.acm.org/10.1145/3007592.3007600

[48] J. Beauquier and S. Kekkonen-Moneta, "Fault-tolerance and self-stabilization: impossibility results and solutions using self-stabilizing failure detectors," *Int. J. Systems Science*, vol. 28, no. 11, pp. 1177–1187, 1997. [Online]. Available: https://doi.org/10.1080/00207729708929476

[49] E. Anagnostou and V. Hadzilacos, "Tolerating transient and permanent failures (extended abstract)," in *Distributed Algorithms, 7th International Workshop, WDAG '93, Lausanne, Switzerland, September 27-29, 1993, Proceedings*, 1993, pp. 174–188. [Online]. Available: https://doi.org/10.1007/3-540-57271-6_35

[50] S. Dubois, M. Potop-Butucaru, and S. Tixeuil, "Dynamic FTSS in asynchronous systems: The case of unison," *Theor. Comput. Sci.*, vol. 412, no. 29, pp. 3418–3439, 2011. [Online]. Available: https://doi.org/10.1016/j.tcs.2011.02.012

[51] S. Dubois, M. Potop-Butucaru, M. Nesterenko, and S. Tixeuil, "Self-stabilizing byzantine asynchronous unison," *J. Parallel Distrib. Comput.*, vol. 72, no. 7, pp. 917–923, 2012. [Online]. Available: https://doi.org/10.1016/j.jpdc.2012.04.001

[52] S. Dolev, K. Eldefrawy, J. A. Garay, M. V. Kumaramangalam, R. Ostrovsky, and M. Yung, "Brief announcement: Secure self-stabilizing computation," in *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, 2017, pp. 415–417. [Online]. Available: http://doi.acm.org/10.1145/3087801.3087864

[53] R. Baldoni, J. Hélary, M. Raynal, and L. Tanguy, "Consensus in byzantine asynchronous systems," *J. Discrete Algorithms*, vol. 1, no. 2, pp. 185–210, 2003. [Online]. Available: https://doi.org/10.1016/S1570-8667(03)00025-X

[54] S. Dolev, C. Georgiou, I. Marcoullis, and E. M. Schiller, "Self-stabilizing byzantine tolerant replicated state machine based on failure detectors," in *Cyber Security Cryptography and Machine Learning - Second International Symposium, CSCML 2018, Beer Sheva, Israel, June 21-22, 2018, Proceedings*, 2018, pp. 84–100. [Online]. Available: https://doi.org/10.1007/978-3-319-94147-9_7

[55] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990. [Online]. Available: http://doi.acm.org/10.1145/98163.98167

[56] F. B. Schneider and L. Zhou, "Implementing trustworthy services using replicated state machines," *IEEE Security & Privacy*, vol. 3, no. 5, pp. 34–43, 2005. [Online]. Available: https://doi.org/10.1109/MSP.2005.125

[57] H. Attiya and J. Welch, *Distributed computing: fundamentals, simulations, and advanced topics, 2nd Edition*. John Wiley & Sons, 2004, vol. 19.

[58] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985. [Online]. Available: http://doi.acm.org/10.1145/3149.214121

[59] J. Aspnes, "Randomized protocols for asynchronous consensus," *Distributed Computing*, vol. 16, no. 2-3, pp. 165–175, 2003. [Online]. Available: http://dx.doi.org/10.1007/s00446-002-0081-5

[60] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *Journal of the ACM*, vol. 43, no. 4, pp. 685–722, 1996. [Online]. Available: http://doi.acm.org/10.1145/234533.234549

[61] E. Gafni and L. Lamport, "Disk paxos," *Distributed Computing*, vol. 16, no. 1, pp. 1–20, 2003. [Online]. Available: http://dx.doi.org/10.1007/s00446-002-0070-8

[62] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006. [Online]. Available: http://dx.doi.org/10.1007/s00446-006-0005-x

[63] ——, "Byzantizing paxos by refinement," in *Proceedings of the 25th International Symposium on Distributed Computing (DISC'11)*, 2011, pp. 211–224. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24100-0_22

[64] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, 2006, pp. 335–350. [Online]. Available: http://www.usenix.org/events/osdi06/tech/burrows.html

[65] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, 2012, pp. 261–264. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett

[66] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[67] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996. [Online]. Available: http://doi.acm.org/10.1145/226643.226647

[68] D. Dolev and D. Malki, "The transis approach to high availability cluster communication," *Commun. ACM*, vol. 39, no. 4, pp. 64–70, Apr. 1996. [Online]. Available: http://doi.acm.org/10.1145/227210.227227

[69] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos, "Totem: A fault-tolerant multicast communication system," *Commun. ACM*, vol. 39, no. 4, pp. 54–63, 1996. [Online]. Available: http://doi.acm.org/10.1145/227210.227226

[70] A. Fekete, N. Lynch, and A. Shvartsman, "Specifying and using a partitionable group communication service," *ACM Transactions on Computer Systems (TOCS)*, vol. 19, no. 2, pp. 171–216, 2001.

[71] A. Bartoli, "Implementing a replicated service with group communication," *Journal of Systems Architecture*, vol. 50, no. 8, pp. 493–519, 2004. [Online]. Available: http://dx.doi.org/10.1016/j.sysarc.2003.11.003

[72] K. Birman, A. Schiper, and P. Stephenson, "Lightweight causal and atomic group multicast," *ACM Trans. Comput. Syst.*, vol. 9, no. 3, pp. 272–314, Aug. 1991. [Online]. Available: http://doi.acm.org/10.1145/128738.128742

[73] K. P. Birman, "The process group approach to reliable distributed computing," *Commun. ACM*, vol. 36, no. 12, pp. 37–53, Dec. 1993. [Online]. Available: http://doi.acm.org/10.1145/163298.163303

[74] ——, *Guide to Reliable Distributed Systems - Building High-Assurance Applications and Cloud-Hosted Services*, ser. Texts in Computer Science. Springer, 2012. [Online]. Available: http://dx.doi.org/10.1007/978-1-4471-2416-0

[75] R. van Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. A. Karr, "Building adaptive systems using ensemble," *Softw., Pract. Exper.*, vol. 28, no. 9, pp. 963–979, 1998. [Online]. Available: https://doi.org/10.1002/(SICI)1097-024X(19980725)28:9⟨963::AID-SPE179⟩3.0.CO;2-9

[76] R. Guerraoui and A. Schiper, "Transaction model vs. virtual synchrony model: Bridging the gap," in *Theory and Practice in Distributed Systems, International Workshop, Dagstuhl Castle, Germany, September 5-9, 1994, Selected Papers*, 1994, pp. 121–132. [Online]. Available: https://doi.org/10.1007/3-540-60042-6_9

[77] S. Dolev and E. Schiller, "Communication adaptive self-stabilizing group membership service," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 7, pp. 709–720, 2003.

[78] A. Schiper and A. Sandoz, "Primary partition "virtually-synchronous communication" harder than consensus," in *Distributed Algorithms, 8th International Workshop, WDAG '94, Terschelling, The Netherlands, September 29 - October 1, 1994, Proceedings*, 1994, pp. 39–52. [Online]. Available: https://doi.org/10.1007/BFb0020423

[79] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*, 2010. [Online]. Available: https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems

[80] M. Raynal, *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013. [Online]. Available: https://doi.org/10.1007/978-3-642-32027-9

[81] H. Attiya and A. Bar-Or, "Sharing memory with semi-byzantine clients and faulty storage servers," *Parallel Processing Letters*, vol. 16, no. 4, pp. 419–428, 2006. [Online]. Available: https://doi.org/10.1142/S0129626406002745

[82] D. Malkhi and M. K. Reiter, "Byzantine quorum systems," *Distributed Computing*, vol. 11, no. 4, pp. 203–213, 1998. [Online]. Available: https://doi.org/10.1007/s004460050050

[83] A. N. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "Depsky: Dependable and secure storage in a cloud-of-clouds," *TOS*, vol. 9, no. 4, pp. 12:1–12:33, 2013. [Online]. Available: http://doi.acm.org/10.1145/2535929

[84] S. Dolev, S. Dubois, M. G. Potop-Butucaru, and S. Tixeuil, "Crash resilient and pseudo-stabilizing atomic registers," in *Proceedings of the 16th International Conference on the Principles of Distributed Systems (OPODIS'12)*, 2012, pp. 135–150. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-35476-2_10

[85] S. Bonomi, M. Potop-Butucaru, and S. Tixeuil, "Stabilizing byzantine-fault tolerant storage," in *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, 2015, pp. 894–903. [Online]. Available: https://doi.org/10.1109/IPDPS.2015.89

[86] S. Bonomi, S. Dolev, M. Potop-Butucaru, and M. Raynal, "Stabilizing server-based storage in byzantine asynchronous message-passing systems: Extended abstract," in *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, 2015, pp. 471–479. [Online]. Available: http://doi.acm.org/10.1145/2767386.2767441

[87] F. Bonnet, X. Défago, T. D. Nguyen, and M. Potop-Butucaru, "Tight bound on mobile byzantine agreement," *Theor. Comput. Sci.*, vol. 609, pp. 361–373, 2016. [Online]. Available: https://doi.org/10.1016/j.tcs.2015.10.019

[88] J. A. Garay, "Reaching (and maintaining) agreement in the presence of mobile faults (extended abstract)," in *Distributed Algorithms, 8th International Workshop, WDAG '94, Terschelling, The Netherlands, September 29 - October 1, 1994, Proceedings*, 1994, pp. 253–264. [Online]. Available: https://doi.org/10.1007/BFb0020438

[89] S. Bonomi, A. D. Pozzo, and M. Potop-Butucaru, "Optimal self-stabilizing synchronous mobile byzantine-tolerant atomic register," *Theor. Comput. Sci.*, vol. 709, pp. 64–79, 2018. [Online]. Available: https://doi.org/10.1016/j.tcs.2017.08.020

[90] D. Peleg and A. Wool, "Crumbling walls: A class of practical and efficient quorum systems," *Distributed Computing*, vol. 10, no. 2, pp. 87–97, 1997. [Online]. Available: http://dx.doi.org/10.1007/s004460050027

[91] M. Vukolic, *Quorum Systems: With Applications to Storage and Consensus*, ser. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012. [Online]. Available: http://dx.doi.org/10.2200/S00402ED1V01Y201202DCT009

[92] S. Dolev, S. Gilbert, N. A. Lynch, A. A. Shvartsman, and J. L. Welch, "Geoquorums: implementing atomic memory in mobile *ad hoc* networks," *Distributed Computing*, vol. 18, no. 2, pp. 125–155, 2005. [Online]. Available: http://dx.doi.org/10.1007/s00446-005-0140-9

[93] K. Birman, D. Malkhi, and R. van Renesse, "Virtually synchronous methodology for dynamic service replication," Microsoft Research, Tech. Rep. MSR-TR-2010-151, 2010. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=141727

[94] V. Bortnikov, G. Chockler, D. Perelman, A. Roytman, S. Shachor, and I. Shnayderman, "Reconfigurable state machine replication from non-reconfigurable building blocks," *CoRR*, vol. abs/1512.08943, 2015. [Online]. Available: http://arxiv.org/abs/1512.08943

[95] M. K. Aguilera, I. Keidar, D. Malkhi, J. Martin, and A. Shraer, "Reconfiguring replicated atomic storage: A tutorial," *Bulletin of the EATCS*, vol. 102, pp. 84–108, 2010. [Online]. Available: http://albcom.lsi.upc.edu/ojs/index.php/beatcs/article/view/43/48

[96] A. Spiegelman, I. Keidar, and D. Malkhi, "Dynamic reconfiguration: A tutorial," *OPODIS 2015*, 2015.

[97] L. Lamport, D. Malkhi, and L. Zhou, "Reconfiguring a state machine," *SIGACT News*, vol. 41, no. 1, pp. 63–73, 2010. [Online]. Available: http://doi.acm.org/10.1145/1753171.1753191

[98] H. Attiya, H. C. Chung, F. Ellen, S. Kumar, and J. L. Welch, "Simulating a shared register in an asynchronous system that never stops changing - (extended abstract)," in *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, 2015, pp. 75–91. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-48653-5_6

[99] E. Gafni and D. Malkhi, "Elastic configuration maintenance via a parsimonious speculating snapshot solution," in *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, 2015, pp. 140–153. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-48653-5_10

[100] L. Jehl, R. Vitenberg, and H. Meling, "Smartmerge: A new approach to reconfiguration for atomic storage," in *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, 2015, pp. 154–169. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-48653-5_11

[101] R. Baldoni, S. Bonomi, A. M. Kermarrec, and M. Raynal, "Implementing a register in a dynamic distributed system," in *Distributed Computing Systems, 2009. ICDCS '09. 29th IEEE International Conference on*, June 2009, pp. 639–647.

[102] G. V. Chockler, S. Gilbert, V. Gramoli, P. M. Musial, and A. A. Shvartsman, "Reconfigurable distributed storage for dynamic networks," *J. Parallel Distrib. Comput.*, vol. 69, no. 1, pp. 100–116, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2008.07.007

[103] L. Jehl and H. Meling, "The case for reconfiguration without consensus: Comparing algorithms for atomic storage," in *20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain*, 2016, pp. 31:1–31:17. [Online]. Available: https://doi.org/10.4230/LIPIcs.OPODIS.2016.31

[104] A. Nogueira, A. Casimiro, and A. Bessani, "Elastic state machine replication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 9, pp. 2486–2499, 2017. [Online]. Available: https://doi.org/10.1109/TPDS.2017.2686383

[105] A. N. Bessani, J. Sousa, and E. A. P. Alchieri, "State machine replication for the masses with BFT-SMART," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, 2014, pp. 355–362. [Online]. Available: https://doi.org/10.1109/DSN.2014.43

[106] R. Rodrigues and B. Liskov, "Rosebud: A scalable byzantine-fault-tolerant storage architecture," Massachusetts Institute of Technology, Department of Computer Science, Tech. Rep., 12 2003.

[107] C. Cachin, "State machine replication with byzantine faults," in *Replication: Theory and Practice*, ser. Lecture Notes in Computer Science, B. Charron-Bost, F. Pedone, and A. Schiper, Eds., vol. 5959. Springer, 2010, pp. 169–184. [Online]. Available: https://doi.org/10.1007/978-3-642-11294-2_9

[108] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Trans. Comput. Syst.*, vol. 27, no. 4, pp. 7:1–7:39, 2009. [Online]. Available: http://doi.acm.org/10.1145/1658357.1658358

[109] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin one's wheels? byzantine fault tolerance with a spinning primary," in *28th IEEE Symposium on Reliable Distributed Systems (SRDS 2009), Niagara Falls, New York, USA, September 27-30, 2009*, 2009, pp. 135–144. [Online]. Available: https://doi.org/10.1109/SRDS.2009.36

[110] P. Aublin, R. Guerraoui, N. Knezevic, V. Quéma, and M. Vukolic, "The next 700 BFT protocols," *ACM Trans. Comput. Syst.*, vol. 32, no. 4, pp. 12:1–12:45, 2015. [Online]. Available: http://doi.acm.org/10.1145/2658994

[111] M. Pires, S. Ravi, and R. Rodrigues, "Generalized paxos made byzantine (and less complex)," in *Stabilization, Safety, and Security of Distributed Systems - 19th International Symposium, SSS 2017, Boston, MA, USA, November 5-8, 2017, Proceedings*, 2017, pp. 203–218. [Online]. Available: https://doi.org/10.1007/978-3-319-69084-1_14

[112] J. Martin and L. Alvisi, "Fast byzantine consensus," *IEEE Trans. Dependable Sec. Comput.*, vol. 3, no. 3, pp. 202–215, 2006. [Online]. Available: https://doi.org/10.1109/TDSC.2006.35

[113] J. P. Bahsoun, R. Guerraoui, and A. Shoker, "Making BFT protocols really adaptive," in *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, 2015, pp. 904–913. [Online]. Available: https://doi.org/10.1109/IPDPS.2015.21

[114] M. Ben-Or, "Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract)," in *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*, 1983, pp. 27–30. [Online]. Available: http://doi.acm.org/10.1145/800221.806707

[115] A. Haeberlen, P. Kouznetsov, and P. Druschel, "PeerReview: practical accountability for distributed systems," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, 2007, pp. 175–188. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294279

[116] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo, "Efficient byzantine fault-tolerance," *IEEE Trans. Computers*, vol. 62, no. 1, pp. 16–30, 2013. [Online]. Available: https://doi.org/10.1109/TC.2011.221

[117] A. Baliga, "Understanding blockchain consensus models," Persistent Systems Ltd, Tech. Rep, Tech. Rep., 2017.

[118] M. Vukolic, "The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication," in *Open Problems in Network Security - IFIP WG 11.4 International Workshop, iNetSec 2015, Zurich, Switzerland, October 29, 2015, Revised Selected Papers*, 2015, pp. 112–125. [Online]. Available: https://doi.org/10.1007/978-3-319-39028-4_9

[119] C. Cachin, "Architecture of the hyperledger blockchain fabric," in *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2016.

[120] A. Daliot and D. Dolev, "Self-stabilizing byzantine agreement," in *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing, PODC 2006, Denver, CO, USA, July 23-26, 2006*, 2006, pp. 143–152. [Online]. Available: http://doi.acm.org/10.1145/1146381.1146405

[121] G. V. Chockler, R. Guerraoui, I. Keidar, and M. Vukolic, "Reliable distributed storage," *IEEE Computer*, vol. 42, no. 4, pp. 60–67, 2009. [Online]. Available: https://doi.org/10.1109/MC.2009.126

[122] D. Dolev, E. N. Hoch, and R. van Renesse, "Self-stabilizing and byzantine-tolerant overlay network," in *Principles of Distributed Systems, 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings*, 2007, pp. 343–357. [Online]. Available: https://doi.org/10.1007/978-3-540-77096-1_25

[123] S. Dolev, O. Liba, and E. M. Schiller, "Self-stabilizing byzantine resilient topology discovery and message delivery - (extended abstract)," in *Networked Systems - First International Conference, NETYS 2013, Marrakech, Morocco, May 2-4, 2013, Revised Selected Papers*, 2013, pp. 42–57. [Online]. Available: https://doi.org/10.1007/978-3-642-40148-0_4

[124] B. Awerbuch, B. Patt-Shamir, and G. Varghese, "Self-stabilizing end-to-end communication," *J. High Speed Networks*, vol. 5, no. 4, pp. 365–381, 1996. [Online]. Available: https://doi.org/10.3233/JHS-1996-5404

[125] Y. Afek and G. M. Brown, "Self-stabilization over unreliable communication media," *Distributed Computing*, vol. 7, no. 1, pp. 27–34, 1993. [Online]. Available: http://dx.doi.org/10.1007/BF02278853

[126] A. M. Costello and G. Varghese, "Self-stabilization by window washing," in *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, 1996, pp. 35–44. [Online]. Available: http://doi.acm.org/10.1145/248052.248056

[127] S. Dolev, S. Dubois, M. Potop-Butucaru, and S. Tixeuil, "Stabilizing data-link over non-fifo channels with optimal fault-resilience," *Inf. Process. Lett.*, vol. 111, no. 18, pp. 912–920, 2011. [Online]. Available: http://dx.doi.org/10.1016/j.ipl.2011.06.010

[128] S. Dolev, A. Hanemann, E. M. Schiller, and S. Sharma, "Self-stabilizing end-to-end communication in (bounded capacity, omitting, duplicating and non-fifo) dynamic networks - (extended abstract)," in *Proceedings of the 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'12)*, 2012, pp. 133–147. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33536-5_14

[129] F. Bonnet and M. Raynal, "Early consensus in message-passing systems enriched with a perfect failure detector and its application in the theta model," in *Eighth European Dependable Computing Conference, EDCC-8 2010, Valencia, Spain, 28-30 April 2010*, 2010, pp. 107–116. [Online]. Available: https://doi.org/10.1109/EDCC.2010.22

[130] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg, "The weakest failure detectors to solve certain fundamental problems in distributed computing," in *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, 2004, pp. 338–346. [Online]. Available: http://doi.acm.org/10.1145/1011767.1011818

[131] L. Alvisi, D. Malkhi, E. T. Pierce, and M. K., "Fault detection for byzantine quorum systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 9, pp. 996–1007, 2001. [Online]. Available: https://doi.org/10.1109/71.954640

[132] A. Arora, S. S. Kulkarni, and M. Demirbas, "Resettable vector clocks," *J. Parallel Distrib. Comput.*, vol. 66, no. 2, pp. 221–237, 2006. [Online]. Available: https://doi.org/10.1016/j.jpdc.2005.07.001

[133] R. Khazan, A. Fekete, and N. A. Lynch, "Multicast group communication as a base for a load-balancing replicated data service," in *DISC*, ser. Lecture Notes in Computer Science, vol. 1499. Springer, 1998, pp. 258–272.

[134] S. Dolev and N. Tzachar, "Empire of colonies: Self-stabilizing and self-organizing distributed algorithm," *Theor. Comput. Sci.*, vol. 410, no. 6-7, pp. 514–532, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2008.10.006

[135] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper, "Muteness failure detectors: Specification and implementation," in *Dependable Computing - EDCC-3, Third European Dependable Computing Conference, Prague, Czech Republic, September 15-17, 1999, Proceedings*, 1999, pp. 71–87. [Online]. Available: https://doi.org/10.1007/3-540-48254-7_7

[136] A. Mostéfaoui, E. Mourgaya, and M. Raynal, "Asynchronous implementation of failure detectors," in *2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, Proceedings*, 2003, pp. 351–360. [Online]. Available: https://doi.org/10.1109/DSN.2003.1209946

[137] M. Demirbas, A. Charapko, and A. Ailijiang, "Does the cloud need stabilizing?" *CoRR*, vol. abs/1806.03210, 2018. [Online]. Available: http://arxiv.org/abs/1806.03210