Robustness of Distributed Systems Inspired by Biological Processes

Thesis for the degree of

DOCTOR of PHILOSOPHY

by

Ariel Daliot

SUBMITTED TO THE SENATE OF
THE HEBREW UNIVERSITY OF JERUSALEM

November 2006

This work was carried out under the supervision of
Prof. Hanna Parnas and Prof. Danny Dolev

Acknowledgements

Should I list all the people that have to some extent or the other affected (or been affected..) by my thesis then it would have to comprise a chapter of its own. I will keep it short. I first wish to thank my family for being so patient and for not renouncing me altogether. I wish to thank all the good and smart people that have contributed ideas, corrections, moral support, time and belief. Among them are Oren Schuldiner, Rami Tzafriri, Noam Shoresh, Erez Ezrachi, Yonathan Livny. I wish to thank my advisors. Prof. Hanna Parnas for always believing in this work, for always being there when needed, for saving me much work during the years through her sharp intuitions and for seeing the good of the student before anything else. I wish to thank Prof. Danny Dolev for teaching me that if it looks hopelessly complicated and almost intractable that's when it starts to get interesting and for teaching me that you always carry on you whatever you need to get out of any hopelessly complicated and almost intractable situation. Last but certainly not least I wish to thank all the great friends from the Parnasia lab and the DANSS lab whom without, doing this thesis would have been much more difficult and definitely less fun.

Abstract

Synchronization in distributed computer networks has been extensively studied in the last decades. Nevertheless, robustness and fault tolerance still remain major challenges. Biological systems have evolved to operate successfully in exceptionally noisy and fault-prone environments. The subject of this work is robustness of distributed computer systems and specifically the potential of biological synchronization models to inspire novel synchronization algorithms in distributed computer system. Prior to this work, only two algorithms existed in the field of distributed computing that were both self-stabilizing and resilient to permanent Byzantine failures, both with exponential convergence times. Several biological models seem to operate in extreme fault models. In this thesis, we pursued to algorithmically imitate such a model and derived the first practical self-stabilizing and Byzantine algorithm .

A self-stabilizing and arbitrary-fault tolerant (Byzantine) algorithm for synchronization of pulses is derived from generalizing a model of the cardiac pacemaker in lobsters. It assumes no prior synchronization besides bounded message delay and it converges in linear time. The synchronization achieved is very tight. Pulse synchronization, which resembles clock synchronization, has not previously been formally defined in the settings of distributed computer systems.

Executing on top of pulse synchronization, an algorithm to transform general Byzantine algorithms into their self-stabilizing counterparts is developed, with only a linear-time additional overhead. Then, an optimized scheme for stabilizing a subset of the class of general Byzantine algorithms is suggested. Following the lines of this scheme, an efficient linear-time self-stabilizing Byzantine solution for the fundamental clock synchronization problem is presented, that has an exceptionally low (constant) overhead during steady-state. An efficient linear-time self-stabilizing Byzantine solution for token circulation and for graph coloring and general resource allocation problems is also shown.

A self-stabilizing Byzantine agreement algorithm is presented. It does not assume synchronized pulses, only bounded message delay and it converges in linear time. The lack of synchrony is overcome, by developing a self-stabilizing Byzantine version of the reliable broadcast primitive.

A second pulse synchronization algorithm is then developed that executes on top of the self-stabilizing Byzantine agreement algorithm. It is fundamentally different from the biologically-inspired one. It converges faster and also achieves very tight synchronization.

All the algorithms need to address the particular lack of any synchronization assumptions as is posed by the confluence of the self-stabilization and Byzantine fault models. They face the problem of creating synchrony and agreement among nodes from a point of no synchrony and with malicious nodes that incessantly hamper stabilization. The timing model is that of semi-synchronous networks.

All but one of the results are the first solutions for their respective problem in the combined self-stabilizing and Byzantine fault domains. The algorithm for self-stabilizing Byzantine clock synchronization, is not the first of its kind but the only one that is linear-time and not exponential.

Contents

Chapter 1

Introduction

This thesis addresses the problem of synchronizing the elements that comprise a distributed system in the face of extreme failures. The only synchrony assumed is that of eventual bounded-delay message delivery. The best way to give some initial intuition to the difficulties of this problem, as well as the motivation behind our solution, is through an illustrative example: It is a well known phenomenon that after a good performance the clapping by the audience is frequently followed by distinct synchronized clapping by most of the people. This synchronized clapping occurs with no prior coordination. Should a conductor initially instruct the audience when to clap following which they only need to maintain the rhythm, then this would intuitively be a much simpler task. In distributed computer systems, the lack of prior coordination is addressed by the self-stabilization fault model. Distinct synchronized clapping is typically also achieved in-spite of a certain fraction of the audience that are reluctant to cooperate and even intentionally clap in their own pace. Intuitively, it would be a much simpler task to attain synchronized clapping without this fraction of sceptics. In distributed computer systems, a bounded fraction of ill-behaved or even malicious behavior is addressed by the Byzantine fault model. This thesis deals with attaining synchronization in the confluence of both the Byzantine and self-stabilization fault models. It is notoriously difficult to achieve results in this fault model, as can be indicated by the surprisingly few prior algorithms developed in this model. Within the current Introduction chapter, we postulate and elaborate on the reasons for this difficulty.

Our work begins and is motivated by the fact that in biology very robust synchronization is frequently achieved surprisingly fast and efficient, while attaining very robust synchronization in distributed computer systems is surprisingly slow and inefficient, if it exists at all.

The phenomenon of robust synchronization is displayed by many biological systems [72] and presumably plays an important role in these systems. For example, the heart of the lobster is regularly activated by the synchronized firing of four neurons in the cardiac pacemaker network [40,41]. It has been concluded in [70] that the organism cannot survive if all four interneurons fire out of synchrony for prolonged times. That system inspired the present work. Other examples of biological synchronization include the *malaccae* fireflies in Southeast Asia, where thousands of male fireflies congregate in mangrove trees, flashing in synchrony [15]; oscillations of the neurons in the circadian pacemaker, determining the day-night rhythm; crickets that chirp in unison [74]; coordinated mass spawning in corals and of course an audience clapping together after a "good" performance [63]. Synchronization in these systems is typically attained despite the inherent variations among the participating elements, failures, or the presence of noise from external sources or

from participating elements. A generic mathematical model for synchronous firing of biological oscillators based on a model of the human cardiac pacemaker is given in [61]. This model does not account for noise or for the inherent differences among biological elements.

In computer science, synchronization is both a goal by itself and an essential building block for algorithms that solve other problems.

In general, it is desired for algorithms to guarantee correct behavior of the system in face of faults or failing elements, without strong assumptions on the initial state of the system. It has been suggested in [70] that similar fault considerations may have been involved in the evolution of distributed biological systems. In the example of the cardiac pacemaker network of the lobster, it was concluded that at least four neurons are needed in order to overcome the presence of one faulty neuron, though supposedly one neuron suffices to activate the heart. The cardiac pacemaker network must be able to adjust the pace of the synchronized firing according to the required heartbeat, up to a certain bound, without losing the synchrony (e.g. while escaping a predator a higher heartbeat is required - though not too high). Due to the vitality of this network, it is presumably optimized for fault tolerance, self-stabilization, tight synchronization and for fast re-synchronization.

The apparent resemblance of the synchronization and fault tolerance requirements of biological networks and distributed computer networks makes it very appealing to infer from models of biological systems onto the design of distributed algorithms in computer science. Especially when assuming that distributed biological networks have evolved over time to particularly tolerate inherent heterogeneity of the cells, noise and cell death. In the current thesis, we show that in spite of obvious differences, a biological fault tolerant synchronization model ([70]) can inspire a novel solution to an apparently similar problem in computer science.

1.1 Background and Related Work

1.1.1 Timing Models

This subsection introduces the timing models. A comprehensive description of the model and the definitions that are used throughout the thesis is given in Subsection 1.2.

A “synchronous” timing model (the one with the least uncertainty) is one in which the system works in lock-step. All nodes communicate and compute in synchronous rounds. A rounds starts and ends at exactly the same moment at all nodes. Between non-faulty nodes, messages are sent and received at the same moments respectively.

In the “asynchronous” timing model there is no bound on message arrival and processing time. The nodes interleave their steps in arbitrary order.

In a “bounded-delay” timing model the processing times and message delivery times are between some upper and lower bounds.

The more realistic “partially synchronous” or “semi-synchronous” timing model lies between the synchronous and asynchronous models. It is a bounded-delay timing model and in addition it is frequently assumed that nodes have some knowledge of time, for example access to approximate real-time or some type of facility to measure the progress of time intervals.

See [57] for a comprehensive description of these timing models.

Coordination and synchronization are among the most fundamental elements of a distributed task, and are particularly pivotal in order to withstand severe faults. The details of the timing model

assumed is crucial for being able to achieve coordination and synchronization. In the classic asynchronous network model, although nothing is assumed on the time taken for message delivery, it is typically assumed that nodes have a controlled and common initialization phase [57]. Nodes typically infer about the state of the other correct nodes from their own internal states. Thus it is assumed that the global state is at least partially consistent so that correct nodes have a common notion as to when the system last initialized. This greatly facilitates the progression of the algorithm in "asynchronous rounds" in which a node knows that if it has commenced some specific round r then all other correct nodes have progressed to at least some lesser round. This is a "state-machine replication" approach as a general framework to address the consistency in a distributed environment (see [69]). The asynchronous model prohibits dealing deterministically with most node failures [38], as it might not be possible to distinguish between a late or lost message and a faulty sender. Thus for fault tolerant deterministic computing at least bounded message delivery must be assumed [12]. The minimal extent of synchrony or consistency required for fault tolerance is also discussed in [24]. However, randomized fault-tolerant algorithms in the asynchronous model do exist [13].

In the partially synchronous network model, nodes typically assume bounded time for message delivery in addition to assuming that nodes have a common initialization phase. These two assumptions allow nodes to use timing criteria to deduce whether certain actions should already have taken place. This allows for resilience to permanent faults and plays a pivotal role in the ability to tolerate Byzantine nodes. Synchronization enables correct nodes to determine whether a certain message received at a certain time or with a certain value at this certain time does not agree with the node's perception of the global progress of the algorithm. In order for all correct nodes to view symmetrically whether a node does not behave according to the protocol, it is required to assume that nodes have similar perceptions of the progress of the algorithm.

Since partially synchronous systems have less uncertainty than asynchronous systems, it is tempting to think that they are easier to program. However, there are extra complications that arise from the timing assumptions, for example, algorithms are often designed so that their correctness depends crucially on the timings. Thus algorithms and proofs in the partially synchronous setting are often more complicated than those for the asynchronous and synchronous models.

All the algorithms in this thesis operate in the (eventual) bounded-delay timing model and the nodes only possess a hardware facility to measure the progress of time intervals, denoted as the *physical timer* of the node, but not necessarily any access to real-time. Thus we only assume that eventually the network satisfies the minimal amount of consistency required for fault-tolerance.

1.1.2 Byzantine Fault Tolerance

Being immune to arbitrary/malicious failures (Byzantine faults) may seem like an overly pessimistic requirement. In reality, this is the most practical methodology to overcome a bounded number of faults whose nature is not predictable in advance or a way to seal off unexpected behavior. Bugs in the code, network failures, arbitrary initial values held by a bounded fraction of the nodes in the system and malicious behavior all fall naturally into this category. The "optimistic" approach is to specify what exactly can go wrong and design algorithms that are immune specifically to these problems. This approach is prone to yield unreliable and unsafe algorithms, as in practice it can become very tricky to correctly characterize the faults to be faced with. Moreover, it is frequently the rare "unanticipated" faults that tend to be the catastrophic ones. A "Byzantine

fault" is one that might be arbitrary, malicious or represent an adversary. Subject to the limitation that it cannot corrupt portions of the system to which it has no access, such as changing the code of a non-faulty node. It can display two-faced behavior, i.e. send different messages to different nodes. Byzantine nodes may cooperate with each other.

The "Byzantine Agreement" (Byzantine Generals) problem was first introduced in 1980 by Pease, Shostak and Lamport [68], and is now considered as one of the most fundamental problems in fault-tolerant distributed computing. The task is to reach agreement in a network of n nodes in which up-to f nodes may be faulty. A distinguished node (*the General* or *the initiator*) broadcasts a value *m*, following which all nodes exchange messages until the non-faulty nodes agree upon the same value. If the initiator is non-faulty then all non-faulty nodes are required to agree on the same value that the initiator sent. In the related problem of "Byzantine Consensus", all nodes already initialize with value to be agreed upon. This is essentially equivalent to Byzantine agreement, as the latter may be reduced to the stage at which all nodes received the initiator's value (or did not receive, in case the initiator is faulty).

Traditionally, a Byzantine algorithm focuses on preventing Byzantine faults from shifting the state of the system away from a correct state and thus implicitly assumes that the system initializes in a correct and coherent state (cf. the very first polynomial Byzantine agreement algorithm [28] through many others like [73]). As an example, a traditional algorithm for Byzantine clock synchronization will typically assume that all non-faulty nodes already have all their clocks synchronized [6, 26, 75]; the task is then to prevent the Byzantine nodes from de-synchronizing the clocks of the non-faulty nodes. This assumption enables algorithms to execute in synchronized "rounds", in which the nodes have more or less the same notion as to when a certain round begins and ends. Generally speaking, we can say that Byzantine algorithms assume the existence of at least some sort of basic synchronization or coordination, such as rounds, and then focus on either maintaining or attaining some sort of higher synchronization, such as clock synchronization, agreement, firing squad, pulse synchronization, etc. We elaborate on these types of synchronizations further on.

Toueg, Perry, and Srikanth introduced the "Reliable Broadcast" primitive for solving Byzantine agreement [73]. It encapsulates the task of broadcasting a message to a set of receiving nodes in the presence of faults. In particular, the sender and any other node might fail at any time. A reliable broadcast protocol typically organizes the system into a sending node and a set of receiving nodes, which may include the sender itself. A node is called "correct" if it does not fail at any point during its execution. The goal of the protocol is to transfer any message from a correct sender to the set of receiving correct nodes within a time bound. Moreover, the protocol must guarantee that if a correct node receives a message then all other correct nodes receive the exact same message as well, and only that message (in case a faulty node sends a message to some correct node only or ambivalent messages). Reliable broadcast is used in the following manner as a building block for Byzantine agreement: The initiator of Byzantine agreement broadcasts it's value using reliable broadcast at round(0). Recall that it is guaranteed that if a single correct node receives the initiator's value then all correct nodes receive it within bounded time. Every correct node then re-broadcasts the initiator's value, again using reliable broadcast. Thus, within bounded time, if at least one correct node received the initiator's value then all correct nodes re-broadcast it and every correct node will receive at least $n - f$ such re-broadcasts. This is a sufficient condition for accepting the initiator's value. In case the initiator did not send any value to any correct node then within a specific number of rounds all correct nodes may conclude that the initiator is faulty.

A well-known result for Byzantine agreement algorithms that are tolerant of f concurrent

Byzantine faults is the requirement of at least $f + 1$ rounds of execution [36]. With respect to the bounds on the number of correct nodes, the Byzantine agreement problem has been shown to have no deterministic solution (without authentication) if $n \leq 3f$, where n is the number of nodes in the network [54, 68]. See [37] for impossibility results on many consensus related problems such as clock synchronization, Byzantine agreement, etc. When using authentication, this ratio is not necessarily required, see [26] that in a network of n nodes can tolerate $n - 1$ Byzantine faults. Authentication on the other hand does not change the requirement of at least $f + 1$ rounds of execution for Byzantine agreement [29].

Another relevant problem with a Byzantine tolerant solution is the Byzantine Firing Squad, which is described in [18] (although it was proposed already in about 1957 by John Myhill). If one or more nodes receive a command to start a firing squad synchronization then at some future time (or round) all correct nodes must "fire" (i.e. enter a special state) at exactly the same round.

1.1.3 Self-stabilization

A system may face transient failures that throw the system out of the assumption boundaries. For example, resulting in more than one third of the nodes being Byzantine or messages of non-faulty nodes getting lost or altered or that messages arrive within unbounded time. This will render the whole system practically unworkable. Eventually the system is assumed to experience a tolerable level of permanent faults for a sufficiently long period of time. Otherwise it would remain unworkable forever. When the system eventually returns to behave according to the presumed assumptions, each node may be in an arbitrary state. It makes sense to require a system to resume operation after such a major failure without the need for an outside intervention to restart the whole system from scratch or to correct it. A self-stabilizing algorithm overcomes this limitation by converging within finite time to a correct state from any initial state. Self-stabilizing problems are usually stated in the structure of Convergence, the property that the solution reaches the goal (correct state) within finite time from any initial state, and Closure, the property that the state of the system does not stray away from the goal (remains in a correct state) once this has been realized. Self-stabilizing algorithms typically have a high cost in the convergence time. It is very favorable for a self-stabilizing algorithm to have a lower time and message complexity cost for maintaining the correct state during steady-state (Closure). For a short survey of self-stabilization see [14], for an extensive study see [31]. The field was sparked by Dijkstra's seminal paper on mutual exclusion which is invariant to the initial state of the nodes [23].

A typical self-stabilizing algorithm makes almost no assumptions on the character, number and extent of the faults but assumes that following a finite period of time the system is in a state with no faults whatsoever. This state is arbitrary with respect to the state of the memory and registers (besides a minimal amount of incorruptible code [39]).

Note that in order to be self-stabilizing an algorithm cannot suffer from communication deadlock, as can happen in message-driven algorithms [14]. The algorithms in this thesis overcome this as the nodes have a time-dependent state change,

1.1.4 Self-stabilization with Byzantine Fault Tolerance

Awareness of the need for robustness in distributed systems increases as distributed systems become integral parts of day-to-day systems. Self-stabilizing while tolerating ongoing Byzantine

faults are wishful properties of a distributed system as such a system will be reliable in spite of a constant presence of arbitrary faults and in spite of temporary catastrophic events. This can be underlined by the recent interest in this combined, extreme fault model, see for example [58]. Many distributed tasks (e.g. clock synchronization) have numerous efficient non-stabilizing solutions tolerating Byzantine faults or conversely non-Byzantine but self-stabilizing solutions. In contrast, solutions combining self-stabilization and Byzantine fault tolerance are rare and pose a special challenge. The difficulty stems from the apparent cyclic paradox of the pivotal role of synchronization for containing the faulty nodes combined with the fact that a self-stabilizing algorithm cannot assume any sort of synchronization or inference of the global state from the local state. This difficulty may be indicated by the remarkably few algorithms that are resilient to both fault models. Observe that assuming a fully synchronous model in which nodes progress in perfect lock-step does not ease this problem (cf. [34]).

In this combined fault model correct nodes cannot assume a common reference to time or even to any common anchor in time and they cannot assume that any procedure or primitive has initialized concurrently at all nodes (such as reliable broadcast for example). This is the result of the possible loss of synchronization following transient faults that might corrupt any agreement or coordination among the correct nodes and alter their internal states (such as the notion as to how long ago the system or algorithm was initialized or a common notion of rounds). Thus even basic synchronization or coordination must be restored from an arbitrary state while facing on-going Byzantine failures. This is a very tricky task considering that all classic tools for containing Byzantine failures, such as [18, 73], assume that synchronization already exists and are thus preempted for use. The protocols in this thesis achieve self-stabilizing Byzantine synchronization without the assumption of any existing synchrony besides eventual bounded message delivery. In [12] it is proven to be impossible to combine self-stabilization with even crash faults without the assumption of bounded message delivery.

In the case of self-stabilizing Byzantine agreement the problem is not relaxed even in the case of a one-shot agreement, i.e. in case that it is known that the General will initiate agreement only once throughout the life of the system. Even if the General is correct and even if agreement is initiated after the system has returned to its coherent behavior following transient failures, then the correct nodes might hold corrupted variable values that might prevent the possibility to reach agreement. The nodes have no knowledge as to when the system returns to coherent behavior or when the General will initiate agreement and thus cannot target to reset their memory exactly at this critical time period. Recurrent agreement initialization by the General allows for recurrent reset of memory with the assumption that eventually all correct nodes reset their memory in a coherent state of the system and before the General initializes agreement. This introduces the problem of how nodes can know when to reset their memory in case of many ongoing concurrent invocations of the algorithm, such as in the case of a faulty General disseminating several values all the time. In such a case correct nodes might hold different sets of messages that were sent by other correct nodes as they might reset their memory at different times.

In [35] consensus is reached assuming eventual synchrony. Following an unstable period with unbounded failures and message delays, eventually no node fails and messages are delivered within bounded time. At this point there is no synchrony among the correct nodes and they might hold copies of obsolete messages. This is seemingly similar to our model but the solution is not truly self-stabilizing since the nodes do not initialize with arbitrary values. Furthermore, the solution only tolerates stopping failures and no new nodes fail subsequent to stabilization. That paper also

argues that in their model, although with Byzantine failures, consensus cannot be reached within less than $O(f')$ time, where f' is the actual number of faults. This is essentially identical to the time complexity of our self-stabilizing Byzantine agreement. Our solution operates in a more severe fault model and thus converges with optimal time complexity.

There are very few specific protocols that tolerate both transient failures as well as permanent Byzantine faults. In this section we survey most of them.

The concept of *multitolerance* is coined by Kulkarni and Arora [9, 51] to describe the property of a system to tolerate multiple fault-classes. They present a component based method for designing multitolerant programs. It is shown how to step-wise add tolerance to the different fault-classes separately. They design as an example a repetitive agreement protocol tolerant to Byzantine failures and to transient failures. Similarly, mutual exclusions for transient and permanent (non Byzantine) faults is designed. In [52] a multitolerant program for distributed reset is designed that tolerates transient and permanent crash failures. It is not shown how the method can be utilized for designing arbitrary algorithms, rather, particular problems are addressed and protocols are specifically designed for these problems using the method.

Nesterenko and Arora [65] define and formalize the notion of *local tolerance* in a multitolerant fault model of unbounded Byzantine faults that eventually comply with the $3f < n$ ratio. Local tolerance refers to the property of faults being contained within a certain distance of the faulty nodes so that nodes outside this containment radius are able to eventually attain correct behavior. They present two locally tolerant Byzantine self-stabilizing protocols for the particular problems of graph coloring and the dining philosophers problem.

Another example is the two randomized self-stabilizing Byzantine clock synchronization algorithms by Dolev and Welch [34], both with exponential convergence times. Our deterministic self-stabilizing Byzantine clock synchronization algorithm in Chapter 4 converges in linear time.

1.1.5 Transforming non-stabilizing Algorithms into Stabilizing ones

Many papers present universal techniques for converting an arbitrary asynchronous protocol into a self-stabilizing equivalent. The asynchrony allows very limited tolerance of faults besides the transient faults. The concept of a *self-stabilizing extension* of a non-stabilizing protocol is brought by Katz and Perry [47]. They show how to compile an arbitrary asynchronous protocol into a self-stabilizing equivalent by centralized predicate evaluation. A self-stabilizing version of Chandy-Lamport snapshots that is recurrently executed is developed. The snapshot is evaluated for a global inconsistency and a distributed reset is done if necessary. This is improved by the local checking method of Awerbuch et al. [11]. Kutten and Patt-Shamir [53] present a time-adaptive transformer which stabilizes any non-stabilizing protocol in $O(f')$ time but on the expense of the space and communication complexities. A stabilizer that takes any off-line or on-line algorithm and "compiles" a self-stabilizing version of it is presented by Afek and Dolev [1]. The stabilizer has the advantage of being local, whereby local it is meant that as soon as the system enters a corrupt state, that fact is detected and second that the expected computation time lost in recovering from the corrupted state is proportional to the size of the corrupted part of the network. "Distributed reset" has been suggested in the context of self-stabilization . E.g. in [8] a distributed reset protocol for shared memory is presented which tolerates fail-stop failures. Note that the fail-stop failure assumption (as opposed to the sudden crash faults) makes the protocol non-masking and thus doesn't truly tolerate permanent faults. Moreover it has a relatively costly convergence time.

Gopal and Perry [44] present a framework for unifying process faults and systemic failures, i.e. ongoing faults and self-stabilization. Their scheme works in a fully synchronous system and is a "compiler" that creates a self-stabilizing version of any fault-tolerant fully synchronous algorithm. They assume the non-stabilizing algorithm works in synchronous rounds. Assuming a fully synchronous system is a strong assumption as it obliterates the need to consider the loss of synchronization of the rounds following a transient failure. Their scheme only assumes the loss of agreement on the round number itself. To overcome this following a systemic (transient) failure, at each round some sort of "agreement" is done on the round number. They assume the register holding the round number is unbounded, which is not a realistic assumption. In a self-stabilizing scheme a transient failure can cause the register to reach its upper limit. Thus they do not handle the overflow and wrap-around of the round number which is a major flaw. The permanent faults that the framework tolerates are any corruption of process code. This may seem very similar to Byzantine faults but the difference hinges on a subtle but significant dissimilarity. It is assumed that corruption of process code cannot result in malicious or two-faced behavior whereas Byzantine failures allow for any adversary behavior. This difference results in the impossibility result for Byzantine behavior, in which at least $3f + 1$ nodes are required to mask f failures [54, 68]. Conversely, corruption of process code imposes no such bound on the number of concurrent failures.

Note that being in an illegal global state is a stable predicate of the system state of a non-stabilizing program as otherwise it would either be self-stabilizing or not have the closure property that is required of any "rational" non-stabilizing algorithm (i.e. if in a legal state then stay in a legal state). A more general way of presenting the stabilizer scheme is as a method for detection of stable predicates (in semi-synchronous networks in our case, see [71] for non fault-tolerant predicate detection in semi-synchronous networks). Distributed reset is just one particular action that can be done upon the detection of a certain predicate. Examples of other predicate detection uses are deadlock detection, threshold detection, progress detection, termination detection, state variance detection (e.g. clock synchronization), among others.

1.1.6 Clock Synchronization

Clock synchronization is a very fundamental task in a distributed system. The vast majority of distributed tasks require some sort of synchronization and clock synchronization is a very straightforward tool for supplying this. It thus makes sense to require an underlying clock synchronization mechanism to be highly fault-tolerant. Clock synchronization aims at giving all the network nodes an approximately common clock reading which also targets as being a good estimation of real time. There are several efficient algorithms for self-stabilizing clock synchronization withstanding crash faults (see [33,66,30], for other variants of the problem see [7,45]). There are many efficient classic Byzantine clock synchronization algorithms, for a performance evaluation of clock synchronization algorithms see [6]. However, strong assumptions on the initial state of the nodes are typically made, usually assuming all clocks are initially synchronized ([6, 26, 75]) and thus these are not self-stabilizing solutions. On the other hand, self-stabilizing clock synchronization algorithms allow initialization with arbitrary clock values, but typically have a cost in the convergence times and do not tolerate permanent faults. Evidently, there are very few self-stabilizing solutions facing Byzantine faults ([34]), all with unpractical convergence times. The clock synchronization protocols of Dolev and Welch in [34] are to the best of our knowledge the first self-stabilizing protocols that are tolerant to Byzantine faults. A special challenge in self-stabilizing clock synchronization

is the clock wrap around. In non-stabilizing algorithms having a large enough integer eliminates the problem for any practical concern. In self-stabilizing schemes a transient failure can cause clocks to hold arbitrary large values, surfacing the issue of clock bounds. Hence self-stabilizing clock synchronization has an inherent difficulty in estimating real-time without an external time reference due to the fact that non-faulty nodes may initialize with arbitrary clock values. Thus, self-stabilizing clock synchronization aims at reaching a legal state from which clocks proceed synchronously at the rate of real-time (assuming that nodes have access to a physical timer whose rate is close to real-time) and not necessarily at estimating real-time. Many applications utilizing the synchronization of clocks do not really require the exact real-time notion (see [55]). In such applications, agreeing on a common clock reading is sufficient as long as the clocks progress within a linear envelope of any real-time interval.

1.1.7 Pulse Synchronization

Pulses, pulse-coupling and pulse synchronization are terms that have been used for a long time in the context of neurobiological networks, as well as laser optics and electronics. To the best of our knowledge the problem of "Pulse Synchronization" has not previously been formally defined in the context of distributed computer systems. In [10] an algorithm for self-stabilizing synchronized pulses for asynchronous networks is developed. The idea is to attach a pulse number to asynchronous rounds such that any message sent at local pulse i is received at the other endpoint before local pulse $i+1$. The problem is fundamentally different from our semi-synchronous pulses since it does not target for the co-occurrences of the pulses, rather as an asynchronous phase synchronizer. In [34] pulses are mentioned as some external event that occurs at all nodes.

In Subsection 1.2 we define the "Pulse Synchronization". It mimics the goal of the cardiac pacemaker neuronal network in the lobster, in which the neurons are required to invoke a regular pulse in tight synchrony, but allows to deviate from exact regularity. The analogous problem in the settings of distributed computer systems is for the nodes to perform a regular "task" or "event" synchronously. This target becomes surprisingly subtle and difficult to achieve when facing both transient and permanent failures. In this thesis we present two very different algorithms for pulse synchronization that self-stabilize while at the same time tolerate a permanent presence of Byzantine faults. Transient failures might throw the system into an arbitrary state in which correct nodes have no common notion what-so-ever, such as time or round numbers, and can thus not infer anything from their own local states upon the state of other correct nodes. The problem in general is to return to a consistent global state from a corrupted global state. The problem as stated in terms of pulse synchronization, is to attain a consistent global state with respect to the pulse event only. I.e. that a correct node can infer that other correct nodes will have invoked their pulse within a very small time window of its own pulse invocation. The Byzantine nodes might incessantly try to de-synchronize the correct nodes. The faulty nodes must not be able to ruin an already attained synchronization, during steady-state; in the worst case, they can slow down the convergence towards synchronization. Interestingly enough, this type of synchronization is sufficient for eventually attaining a consistent general global state from any corrupted general global state. See Chapter 3, where it shown how, by assuming synchronized pulses, almost any Byzantine algorithm can be converted to its self-stabilizing Byzantine counterpart. To the best of our knowledge there is sofar no alternative method besides relying on pulse synchronization for this. A large part of

this thesis presents specific self-stabilizing Byzantine solutions to classical problems such as clock synchronization and token circulation, based on self-stabilizing Byzantine pulse synchronization.

It is important to observe that Byzantine (non-stabilizing) pulse synchronization can be trivially derived from Byzantine clock synchronization. Self-stabilizing (non-Byzantine) pulse synchronization can be easily achieved by following any node that invokes a pulse. Self-stabilizing Byzantine pulse synchronization on the other hand is a surprisingly subtle and difficult problem. No practical self-stabilizing Byzantine clock synchronization algorithm that does not assume the existence of synchronized pulses exists. Thus there is no immediate way to yield a practical self-stabilizing Byzantine pulse synchronization algorithm. The lower bound on clock synchronization in completely connected, fault-free networks is $d(1 - 1/n)$ [56], where d is the bound on the message delay, which is thus also a lower bound for self-stabilizing Byzantine pulse synchronization.

Note that pulse synchronization resembles a firing squad [18] in which the nodes are instructed to "fire" regularly. To date there is no self-stabilizing Byzantine firing squad algorithm which could be used for solving self-stabilizing Byzantine pulse synchronization. It is no straightforward task to make existing Byzantine firing squad algorithms be self-stabilizing. Originally, the firing squad problem was designed for synchronous networks and the goal was for the nodes to fire at exactly the same round. Our model is the semi-synchronous network model. Thus the equivalent problem is for the nodes to "fire" within some time interval of each other. This is more suitably seen as pulse synchronization. Our self-stabilizing Byzantine pulse synchronization results can be seen as self-stabilizing Byzantine firing squad algorithms for the semi-synchronous network model.

In [76] it is shown how to initialize Byzantine clock synchronization among correct nodes that boot at different times. Thus eventually they can also produce synchronized Byzantine pulses (by using the synchronized clocks). That solution is not self-stabilizing as nodes are booted and thus do not initialize with arbitrary values in the memory. It has, on the other hand, a constant convergence time with respect to the required rounds of communication, whereas the solution in this thesis has a dependency on f, which is due to the self-stabilization requirement.

To elucidate the difficulties in trying to solve the pulse synchronization problem it may be instructive to outline a flaw[1] in an early attempt to solve this problem [21, 59]: Non-stabilizing Byzantine algorithms assume that all correct nodes have symmetric views on the other correct nodes. E.g. if a node received a message from a correct node then its assumed all correct nodes did so to. Following transient failures though, a node might initialize in a spurious state reflecting some spurious messages from correct nodes. With the pulse synchronization problem, this spurious state may be enough to trigger a pulse at the node. In order to synchronize their pulses nodes need to broadcast that they have invoked a pulse or that they are about to do so. Correct nodes need to observe such messages until a certain threshold for invoking a pulse is reached. When nodes invoke their pulses this threshold will be reached again subsequent to invoking the pulse, causing a correct node to immediately invoke a pulse again and again.

To prevent incessant pulse invocations, a straightforward solution to this problem is to have a large enough period subsequent to the pulse invocation in which a node does not consider received messages towards the threshold. This is exactly where the pitfall lies, since some correct nodes may initialize in a state that causes them to invoke a pulse based on spurious messages from correct nodes. The consequent pulse message might then arrive at other correct nodes that initialize in a period in which they do not consider received messages. Byzantine nodes can, by sending carefully

[1] The flaw was pointed out by Mahyar Malekpour from NASA LaRC and Radu Siminiceanu from NIA, see [59].

timed messages, cause correct nodes to invoke their pulses in perfect anti-synchrony forever. It is no trivial task to circumvent these difficulties.

In Subsection 1.2 we prove that pulse synchronization requires $n > 3f$.

1.1.8 From a Biological Model to a Pulse Synchronization Algorithm

The motivation for this thesis grew from the many spectacular examples of very high robustness displayed by many biological networks, as specified in the beginning of this chapter. The work in [70] provided the novel idea of characterizing and quantifying measures of robustness of a biological network using measures and tools from distributed systems. As an example, in distributed computer networks it is known that the various incarnations of consensus, such as Byzantine agreement, Byzantine clock synchronization, etc. cannot be solved if the number of participating network elements, n, is less than $3f + 1$, if f faults may occur concurrently. In [70] it has been concluded that in the cardiac pacemaker network of the lobster, at least four neurons are needed in order to overcome the presence of one faulty neuron, though supposedly one neuron suffices to activate the heart. Thus for one "biological fault" that may occur, such as neuron death, neuron's arbitrary firing pattern, etc., the number of neurons in the network must be at least four. Specifically, the actual number of neurons in the cardiac ganglion of the lobster is exactly four, which settles with the conjectured minimal redundancy required for fault tolerance in this biological network.

The cardiac pacemaker network must be able to adjust the pace of the synchronized firing according to the required heartbeat, up to a certain bound, without losing the synchrony (e.g. while escaping a predator a higher heartbeat is required – though not too high). Due to the vitality of this network, it is presumably optimized for its task.

A study of the model of the lobster cardiac pacemaker network in [70, 40, 41] showed that neurons might fail and display highly irregular firing patterns or might also cease to function altogether. Not all the neurons are connected directly to each other, thus an arbitrary firing pattern will be perceived differently by the other functioning neurons. An additional interesting characteristic of this neuronal network is the fact that the four neurons that are required to keep tight synchronization may actually de-synchronize into an arbitrary synchronization pattern for a short period of time due to faults, hormones or even temperature change. The neurons can re-synchronize from a pattern where they fire completely out of synchrony of each other or any other intermediate synchronization pattern. It is pretentious to map noise or malfunctioning elements of a biological entity onto well-defined faults in distributed computer systems. Yet it is suggestive to put forward that the cardiac pacemaker network overcomes failures or situations that seem more extensive than what falls into the Byzantine framework, such as tolerating seemingly arbitrary behavior by a single neuron, and at the same time be able to synchronize from an apparently arbitrary firing pattern of the neural network.

The first task of this thesis is engineering a "Pulse Synchronization" algorithm for distributed computer networks, that mimics the behavior of the cardiac pacemaker model, facing faults similar in nature to the faults tolerated by the neural network. The most suggestive fault model to define for this algorithm is a combination of the Byzantine and self-stabilization fault models.

1.2 Model and General Definitions

A description of the timing model is given in Subsection 1.2, due to its importance in understanding the subtleties of fault-tolerance, as discussed throughout the Introduction.

In the current section we give the full description of the model and all the definitions that are used throughout the thesis. Definitions that are specific to a certain chapter are given in the beginning of that chapter. There are constant values that are defined in the current section but hold different values in the different chapters. These values are defined in the current chapter but get their specific value in the respective chapter.

The environment is a network of n nodes, that communicate by exchanging messages, out of which at most f are faulty nodes. The actual number of concurrent faults is denoted $f' \leq f$. We assume that the message passing allows for an authenticated identity of the direct senders only. The communication network does not guarantee any order on messages among different nodes. When the system is not coherent then there can be an unbounded number of concurrent Byzantine faulty nodes, the turnover rate between faulty and non-faulty nodes can be arbitrarily large and the communication network may behave arbitrarily. Eventually the system settles down in a coherent state in which there at most $f < 3n$ permanent Byzantine faulty nodes, which is the minimal requirement of redundancy of algorithms for tolerating Byzantine faults, without authentication. Eventually there is also bounded message delay, which is the minimal timing requirement of a self-stabilizing algorithm for tolerating any permanent fault. Individual nodes have no access to a central clock and there is no external pulse system. The hardware clock (referred to as the *physical timer*) rate of a correct node has a bounded drift, ρ, from real-time rate. We denote *'physical timer'(u,v)* the amount of clock ticks of the physical timer in a real-time interval $[u, v]$.

DEFINITION 1.2.1 *A node is* **non-faulty** *at times that it complies with the following:*

1. (Bounded Drift) *Obeys a global constant* $0 < \rho << 1$ *(typically* $\rho \approx 10^{-6}$*), such that for every real-time interval* $[u, v]$:
$$(1 - \rho)(v - u) \leq \text{ 'physical timer'}(u, v) \leq (1 + \rho)(v - u).$$
2. (Obedience) *Operates according to the protocol.*
3. (Bounded Processing Time) *Processes any message of the protocol within* π *real-time units of arrival time.*

A node is considered **faulty** if it violates any of the above conditions. A Byzantine node may recover from its faulty behavior once it resumes obeying the conditions of a non-faulty node. In order to keep the definitions consistent the "correction" is not immediate but rather takes a certain amount of time during which the non-faulty node is still not counted as a correct node, although it supposedly behaves "correctly"[2]. We later specify the time-length of continuous non-faulty behavior required of a recovering node to be considered **correct**.

DEFINITION 1.2.2 *The communication network is* **non-faulty** *at periods that it complies with the following:*

[2]For example, a node may recover with arbitrary variables, which may violate the validity condition if considered correct immediately.

- (Bounded Transmission Delay) *Any message sent by a non-faulty node will arrive at every non-faulty node within δ real-time units.*

Thus, our communication network model (timing model) is an "eventual bounded-delay" communication network.

Basic definitions and notations:

We use the following notations though nodes do not need to maintain all of them as variables.

- $d \equiv \delta + \pi$. Thus, when the communication network is non-faulty, d is the upper bound on the elapsed real-time from the sending of a message by a non-faulty node until it is received and processed by every correct node.

- A ***pulse*** is an internal event targeted to happen in "tight"[3] synchrony at all correct nodes. A ***Cycle*** is the "ideal" time interval length between two successive pulses that a node invokes, as given by the user. The actual *cycle* length, denoted in regular caption, has upper and lower bounds as a result of faulty nodes and the physical clock skew.

- σ represents the upper bound on the real-time window within which all correct nodes invoke a pulse (*tightness of pulse synchronization*). The solution in Chapter 2 achieves $\sigma = 2d$ ($\sigma = d$ in broadcast networks) whereas the solution in Chapter 7 achieves $\sigma = 3d$. We assume that *Cycle* $\gg \sigma$.

- $\phi_i(t) \in \mathbb{R}^+ \cup \{\infty\}$, $0 \leq i \leq n$, denotes, at real-time t, the elapsed real-time since the last pulse invocation of p_i. It is also denoted as the "ϕ of node p_i". We occasionally omit the reference to the time in case it is clear out of the context. For a node, p_j, that has not fired since initialization of the system, $\phi_j \equiv \infty$.

- $cycle_{min}$ and $cycle_{max}$ are values that define the bounds on the actual *cycle* length during correct behavior. The solution in Chapter 2 achieves

$$cycle_{min} = \frac{n-2f}{n-f} \cdot Cycle \cdot (1-\rho) \leq cycle \leq Cycle \cdot (1+\rho) = cycle_{max} \ ,$$

  whereas the solution in Chapter 7 achieves $cycle_{min} = Cycle - 11d \leq cycle \leq Cycle + 9d = cycle_{max}$.

- *pulse_conv* represents the convergence time of the underlying pulse synchronization module. The pulse procedure in Chapter 7 converges within $6 \cdot cycle$. The pulse procedure in Chapter 2 converges within $O(f) \cdot cycle$.

- *agreement_duration* represents the maximum real-time required for the chosen Byzantine consensus/agreement procedure to terminate[4].

[3] We consider $c \cdot d$, for some small constant c, as tight.

[4] We differentiate between *consensus* on an initial value held by all nodes and *agreement* on an initial value sent by a specific possibly faulty node.

Note that the protocol parameters n, f and *Cycle* (as well as the system characteristics d and ρ) are fixed constants and thus considered part of the incorruptible correct code[5]. Thus we assume that non-faulty nodes do not hold arbitrary values of these constants.

A recovering node should be considered correct only once it has been continuously non-faulty for enough time to enable it to have decayed old messages and to have exchanged information with the other nodes through at least a cycle.

DEFINITION 1.2.3 *The communication network is* **correct** *following* Δ_{net}[6] *real-time of continuous non-faulty behavior.*

DEFINITION 1.2.4 *A node is* **correct** *following* Δ_{node}[7] *real-time of continuous non-faulty behavior during a period that the communication network is correct.*

DEFINITION 1.2.5 (System Coherence) *The system is said to be* **coherent** *at times that it complies with the following:*

1. (Quorum) *There are at least* $n - f$ *correct nodes, where* f *is the upper bound on the number of potentially non-correct nodes, at steady state.*

2. (Network Correctness) *The communication network is correct.*

Hence, if the system is not coherent then there can be an arbitrary number of concurrent faulty nodes; the turnover rate between the faulty and non-faulty nodes can be arbitrarily large and the communication network may deliver messages with unbounded delays, if at all. The system is considered coherent, once the communication network and a sufficient fraction of the nodes have been non-faulty for a sufficiently long time period for the pre-conditions for convergence of the protocol to hold. The assumption in this thesis, as underlies any other self-stabilizing algorithm, is that the system eventually becomes coherent.

All the lemmata, theorems, corollaries and definitions hold as long as the system is coherent.

In the algorithms all the correct nodes execute an identical algorithm.

We now seek to give an accurate and formal definition of the notion of pulse synchronization. We start by defining a subset of the system states, which we call *pulse_states*, that are determined only by the elapsed real-time since each individual node invoked a pulse (the ϕ's). We then identify a subset of the pulse_states in which some set of correct nodes have "tight" or "close" ϕ's. We refer to such a set as a *synchronized* set of nodes. To complete the definition of synchrony there is a need to address the recurring brief time period in which a correct node in a synchronized set of nodes has just fired while others are about to fire. This is addressed by adding to the definition nodes whose ϕ's are almost a *Cycle* apart.

If all correct nodes in the system comprise a synchronized set of nodes then we say that the pulse_state is a *synchronized_pulse_states of the system*. The objective of the algorithm is hence to reach a synchronized_pulse_state of the system and to stay in such a state.

[5] A system cannot self-stabilize if the entire code space can be perturbed, see [39].

[6] This constant is defined specifically in every chapter.

[7] This constant is defined specifically in every chapter.

- The **pulse_state** of the system at real-time t is given by:

$$pulse\_state(t) \equiv (\phi_0(t), \ldots, \phi_{n-1}(t)) \ .$$

- Let G be the set of all possible pulse_states of a system.
- A set of nodes, S, is called **synchronized** at real-time t if $\forall p_i, p_j \in S, \phi_i(t), \phi_j(t) \leq cycle_{max}$, and one of the following is true:
  1. $|\phi_i(t) - \phi_j(t)| \leq \sigma$, or
  2. $cycle_{min} - \sigma \leq |\phi_i(t) - \phi_j(t)| \leq cycle_{max}$ and $|\phi_i(t-\sigma) - \phi_j(t-\sigma)| \leq \sigma$.
- $s \in G$ is a **synchronized_pulse_state** *of the system* at real-time t if the set of correct nodes is synchronized at real-time t.

DEFINITION 1.2.6 **The Self-Stabilizing Pulse Synchronization Problem**

Convergence: *Starting from an arbitrary system state, the system reaches a synchronized_pulse_state after a finite time.*

Closure: *If s is a synchronized_pulse_state of the system at real-time t_0 then $\forall$ real-time $t, t \geq t_0$,*

1. *pulse_state(t) is a synchronized_pulse_state,*
2. *In the real-time interval $[t_0,\ t]$ every correct node will invoke at most a single pulse if $t - t_0 \geq cycle_{min}$ and will invoke at least a single pulse if $t - t_0 \geq cycle_{max}$.*

The second Closure condition intends to tightly bound the effective pulse invocation frequency within a priori bounds. This is in order to defy any trivial solution that could synchronize the nodes, but be completely unusable, such as instructing the nodes to invoke a pulse every σ time units. Note that this is a stronger requirement than the "linear envelope progression rate" typically required by clock synchronization algorithms, in which it is only required that clock time progress as a linear function of real-time.

We now prove the lower bound of $n > 3f$ for the Byzantine pulse synchronization problem.

Theorem 1.2.1 *The "Self-stabilizing Byzantine Pulse Synchronization Problem" requires $n > 3f$.*

Proof: This lower bound is derived by reduction to non-stabilizing Byzantine clock synchronization. In [37] it is proven that $n > 3f$ is required for clock synchronization. The proof is done by reducing a slightly simplified variation of the algorithm in Figure 4.1 to an algorithm for Byzantine clock synchronization, in an environment with no transient failures only Byzantine failures. The reduction is done as follows: A pulse synchronization procedure that does not require $n > 3f$, with some value of *Cycle*, is executed in the background. The clock synchronization algorithm is initialized, at a time that the pulses are already synchronized (there are no transient failures) and by setting $clock = 0$. The only difference from the original protocol in Figure 4.1 is that subsequent to each pulse every node sets the next expected time, $Next\_ET$ as a function of the last expected time at pulse, ET, as follows: $Next\_ET$:= ET + *Cycle*. This is essentially the same algorithm

only without executing Byzantine Consensus on the next expected time, which is not needed since the clock synchronization algorithm is initialized with synchronized clocks. Thus all correct nodes will always set the next expected time to identical values. Following the same arguments as the original protocol it can be easily seen that this results in a clock synchronization algorithm which does not require $n > 3f$. A contradiction to the proven required bound for clock synchronization. □

1.3 Dissertation Overview

A brief description of the algorithms and their relationship is given below. In the subsections that follow we review and elaborate on the main results.

First, a biologically-inspired self-stabilizing Byzantine pulse synchronization algorithm is developed in Chapter 2. It assumes no prior coordination or synchronization besides eventual bounded message delay.

In Chapter 3, it shown how, by assuming synchronized pulses, almost any Byzantine algorithm can be converted to its self-stabilizing Byzantine counterpart. This "Byzantine Stabilizer" adds only a linear time complexity overhead to the Byzantine algorithm to be stabilized.

In Chapter 4 a linear-time self-stabilizing Byzantine clock synchronization algorithm is developed, on top of synchronized pulses. This algorithm is highly optimized so that it is significantly faster than if taking an arbitrary existing non-stabilizing Byzantine clock synchronization algorithm and then use the stabilizer scheme of Chapter 3.

In Chapter 5 we do the same for the token circulation problem. This is the first self-stabilizing Byzantine token circulation algorithm, to the best of our knowledge. Like the clock synchronization algorithm, this algorithm is also highly optimized so that it is significantly faster than if taking an arbitrary existing non-stabilizing Byzantine token circulation algorithm and then use the stabilizer scheme of Chapter 3.

In Chapter 6 we develop a self-stabilizing Byzantine agreement algorithm, which does not execute on top of pulse synchronization. It only assumes eventual bounded message delay. It is also faster than if using the stabilizer scheme of Chapter 3.

In Chapter 7 we present a self-stabilizing Byzantine pulse synchronization algorithm that is executed on top of the self-stabilizing Byzantine agreement algorithm presented in Chapter 6. It is fundamentally different than the biologically-inspired one presented in Chapter 2 but in comparison, is more complicated but faster.

For brevity we will use the following shorthand notations in the titles of the chapters: "*SS.*" will denote "*self-stabilizing*" and "*Byz.*" will denote "*Byzantine*".

1.3.1 SS. Byz. Pulse Synchronization Inspired by Biological Networks

This result is described in Chapter 2 and has been published in [22]. The algorithm we devise to solve the "Pulse Synchronization" problem resembles non-linear physical/biological pulse-coupled synchronization models [61]. It may have some similarities to simulated annealing in its iterative convergence but our algorithms does not do random mutations to achieve a better local minimum rather converges deterministically, whereas annealing models and algorithms converge probabilistically to a minimum.

We denote *Cycle* the targeted time-interval between pulse invocations. The convergence time is $2(2f+1)$ cycles, where each cycle is $O(f^2)$ communication rounds. The synchronization tightness of the pulses is near-optimal to within d real-time, in case of a broadcast network, in which any message from a Byzantine node eventually reaches every node. We show in Subsection 2.2 how the algorithm can be executed in a non-broadcast network, in which Byzantine node may display two-faced behavior, achieving synchronization of the pulses to within $3d$ real-time.

The message complexity is at most n messages per cycle, thus to reach synchronization from any arbitrary state its at most $2n(2f+1)$ messages.

The faulty nodes cannot ruin an already attained synchronization; in the worst case, they can slow down the convergence towards synchronization and speed up the synchronized firing frequency up to twice the original frequency. Thus the synchronization accuracy (deviation from real-time rate), is 0.5 *Cycle*.

In comparison to the pulse synchronization algorithm of Chapter 7, the current algorithm is simpler, uses significantly shorter messages; it has a much smaller message complexity and can utilize broadcast network to obtain near-optimal synchronization tightness. Note that although the current algorithm has worse accuracy than the one of Chapter 7, we show in Chapter 4, a method where this has no significance for the clock synchronization algorithm that executes on top. This is also true for the scheme developed in Chapter 3, as long as the effective cycle is longer than the required minimum. In addition the current algorithm introduces novel and interesting elements to distributed computing.

1.3.2 Stabilization of General Byz. Algorithms using Pulse Synchronization

We present a scheme that takes a general Byzantine algorithm and produces its self-stabilizing Byzantine counterpart. This result is described in Chapter 3 and has been published in [19]. The algorithm operates in the semi-synchronous network model typical of Byzantine protocols, though the scheme will also transform any asynchronous algorithm into its self-stabilizing semi-synchronous counterpart. The algorithm has a relatively low overhead of $O(f')$ communication rounds, in addition to the inherent time complexity of the algorithm to be stabilized. The protocol assumes synchronized pulses that are used as events for initializing (non-stabilizing) Byzantine agreement on every node's local state. This ensures that following some bounded time there is consensus on the local state of every node (inclusive of faulty nodes). All correct nodes then evaluate whether this global application snapshot corresponds to a legal state of the Byzantine algorithm and, if required, collectively reset it to a consistent state at the next pulse. We utilize a non-stabilizing Byzantine consensus protocol that works in a time-driven manner that is presented in Chapter 4, which makes the agreement procedure progress as a function of the actual message transmission times and not the upper bound on the message transmission time. Consequently, the additional overhead can in effect be very low.

We also suggest an alternative scheme which may yield for certain algorithms a minimal $O(1)$ overhead during steady-state. In Chapter 4 and Chapter 5 we develop several algorithms along the lines of this scheme.

1.3.3 SS. Byz. Clock Synchronization using Pulse Synchronization

This result is described in Chapter 4 and has been published in [21]. We present a self-stabilizing Byzantine clock synchronization protocol with the following property: should the system initialize or recover from any transient faults with arbitrary clock values then the clocks of the correct nodes proceed synchronously at real-time rate. Should the clocks of the correct nodes hold values that are close to real-time, then the correct clocks proceed synchronously with high real-time accuracy. The protocol significantly improves upon existing self-stabilizing Byzantine clock synchronization algorithms by reducing the time complexity from expected exponential ([34]) to deterministic $O(f')$. The protocol improves upon existing non-stabilizing Byzantine clock synchronization algorithms by providing self-stabilization while performing with similar complexity.

This deterministic clock synchronization algorithm is based on the observation that all clock synchronization algorithms require events for exchanging clock values and re-synchronizing the clocks to within safe bounds. These events usually need to happen synchronously at the different nodes. In classic Byzantine algorithms this is fulfilled or aided by having the clocks initially close to each other and thus the actual clock values can be used for synchronizing the events. This implies that clock values cannot differ arbitrarily, which necessarily renders these solutions to be non-stabilizing. Our scheme assumes an underlying distributed pulse synchronization module, which is uncorrelated to any clock values. The synchronized pulses are used as the events for re-synchronizing the clock values by triggering a consensus algorithm on the expected clock value at the next pulse. The algorithm is very efficient and attains and maintains high precision of the clocks. It converges with linear time complexity has a strong advantage that after all clocks are synchronized the algorithm's overhead is minimal. This is due to the fortuitous property of the consensus algorithm used (see Subsection 4.4), that terminates in two rounds, despite the presence of f permanent Byzantine nodes, should all correct nodes hold the the same initial consensus value. In comparison, taking the classic non-stabilizing Byzantine clock synchronization algorithm and devising its self-stabilizing counterpart using our Byzantine stabilizer scheme, will yield an algorithm which has $O(f')$ communication rounds overhead during steady-state.

The attained clock precision and accuracy is $11d + O(\rho)$ real-time units, though we present an additional scheme that can attain clock precision and accuracy of $3d + O(\rho)$. The convergence time is $O(f')$ communication rounds and $O(1)$ during steady-state. It is interesting to note that the time complexity, precision and accuracy of this clock synchronization algorithm is as good as the best non-stabilizing Byzantine clock synchronization algorithms [6].

An additional advantage of our algorithm is the use of a Byzantine consensus protocol in Subsection 4.4 that works in a message driven manner. The basic protocol follows closely the early stopping Byzantine agreement protocol of Toueg, Perry and Srikanth [73] but is stated as a consensus algorithm. The main difference is that our protocol rounds progress at the rate of the actual time of information exchange among the correctly operating nodes. This, typically, is much faster than progression with rounds whose time lengths are functions of the upper bound on message delivery time between correct nodes.

1.3.4 SS. Byz. Token Circulation using Pulse Synchronization

This result is described in Chapter 5. Much work has been devoted in the last decades to the token circulation (or leader election) problem. Some papers focus on resilience to permanent faults. To

the best of our knowledge, there is no self-stabilizing token circulation algorithm that tolerates Byzantine faults.

The problem with handling failures in token circulation is how to maintain the circulation of the token, despite the failure of the node holding the token. Token circulation (as well as leader election and mutual exclusion) has been extensively studied in the context of self-stabilization [46,32,3,43]. There are several non-stabilizing protocols that are tolerant to permanent failures, e.g. [42, 2, 62], and very few that self-stabilize while tolerating permanent failures [48,16,9]. Note that the protocol in [16] is a self-stabilizing mutual exclusion algorithm that tolerates permanent arbitrary faults. These may resemble Byzantine faults but it is assumed that correct nodes can identify faulty nodes which thus makes it a much stronger fault model than Byzantine faults. In [5] it is shown that no self-stabilizing Byzantine token circulation protocol can exist in asynchronous systems, even if randomized.

The protocol assumes pulse synchronization. The synchronized pulses are used as events for initializing Byzantine consensus on the proposition of the identity of the next node to hold the token. Once the pulse is invoked, the correct nodes exchange the id of the expected next token holder. If all correct nodes hold the same proposition when invoking the agreement, then the additional overhead to agree on the next token holder is minimal, merely two communication rounds. Otherwise, the nodes initiate consensus on the id of the next token holder and henceforth, as long as the network performs coherently, they will all have the same proposition of the next token holder. Every node holds the token about $1/n^{\text{th}}$ part of the time, where n is the number of nodes in the network. It thus achieves optimal fairness despite the presence of up-to $f < n/3$ Byzantine nodes. Following a transient chaotic situation, the system converges to a desired state within $O(f')$ communication rounds. We also show how to use this scheme to efficiently solve a general resource allocation problem as well as distributed graph coloring.

During steady-state there is only a constant $O(1)$ time complexity overhead. Using the stabilizer scheme in Chapter 3 would yield an algorithm with $O(f')$ communication rounds overhead during steady-state.

1.3.5 SS. Byz. Agreement without using Pulse Synchronization

It is not clear whether agreement is a stronger form of synchronization than pulse synchronization. In practice though, using the building blocks of this thesis, self-stabilizing Byzantine agreement can be realized easily using a pulse synchronization procedure: the pulse invocation can serve as the initialization event for round zero of the agreement protocol. Thus any existing Byzantine agreement protocol may be used, on top of the pulse synchronization procedure, to attain self-stabilizing Byzantine agreement. Such a scheme though has the additional overhead for convergence of a self-stabilizing Byzantine pulse synchronization procedure, in addition to at least $O(f')$ communication rounds of the existing non-stabilizing Byzantine agreement. Chapter 6 presents self-stabilizing Byzantine agreement without assuming synchronized pulses which converges in $O(f')$ communication rounds and is thus much faster. It reaches agreement among the correct nodes in optimal time, and by using only the assumption of bounded message transmission delay. In the process of solving the problem, two additional important and challenging building blocks were developed: a unique self-stabilizing protocol for assigning consistent relative times to protocol initialization and a reliable broadcast primitive that progresses at the speed of actual message delivery time.

The protocol can be executed in a one-shot mode by a single General or by infinite recurrent agreement initializations and by different Generals every time.

For ease of following the arguments and proofs, the structure and logic of our self-stabilizing Byzantine agreement algorithm is modeled on that of [73]. The rounds in that protocol progress following elapsed time. Each round spans a constant predefined time interval. Our protocol, besides being self-stabilizing, has the additional advantage of having a message-driven rounds structure and not time-driven rounds structure. Thus the actual time for terminating the protocol depends on the actual communication network speed and not on the worst possible bound on message delivery time.

1.3.6 SS. Byz. Pulse Synchronization using SS. Byz. Agreement

In Chapter 7 an alternative self-stabilizing pulse synchronization algorithm is presented, which is fundamentally different to that in Chapter 2.

Realizing self-stabilizing Byzantine pulse synchronization through the use of self-stabilizing Byzantine agreement only, seems very complicated. This may indicate that agreement is probably a weaker form of synchronization than pulse synchronization. Intuitively, achieving synchronized pulses on top of a classic (non-stabilizing) Byzantine agreement should supposedly be rather straightforward: Execute non-stabilizing Byzantine agreement on the elapsed time remaining until the next pulse invocation. This scheme requires the correct nodes to terminate agreement within a short time of each other. The major issue is that, unfortunately, when facing transient failures, the system may end up in a state in which any common reference to time or even common anchor in time might be lost. This preempts the use of classic (non-stabilizing) Byzantine agreement and or non-stabilizing reliable broadcast, as these classic tools typically assume initialization with a common reference to time or common reference to a round number. Thus, a common anchor in time is required to execute agreement which aims at attaining and maintaining a common anchor in time, the synchronized pulses. Thus, what is required, is an agreement algorithm that is both self-stabilizing and Byzantine. This apparent "cyclic paradox" is resolved by utilizing the self-stabilizing Byzantine agreement protocol in Chapter 6. Achieving synchronized pulses is still not trivial since merely using self-stabilizing Byzantine agreement to agree on the elapsed time remaining until the next pulse invocation does not prevent the nodes from initializing multiple concurrent agreements, which may result in an arbitrary effective pulse frequency. This is solved by a procedure which aims at first agreeing on a timing event for initialization of agreement on the elapsed time until the next pulse invocation.

The attained pulse synchronization tightness is $3d$. The bound on the effective length of the cycle (the targeted time between successive pulses) attained is within $O(d)$ of the targeted length of *Cycle*. This equals the synchronization accuracy (deviation from real-time rate). The convergence time is 6 cycles, each containing $O(f)$ communication rounds.

In comparison to the pulse synchronization algorithm in Chapter 2, the current solution has a much higher message complexity and much larger message size, and cannot utilize broadcast networks to obtain a near-optimal pulse tightness. The algorithm is also more complicated but converges very fast, whereas that solution converges in about $4f$ cycles with $O(f^2)$ communication rounds each. Moreover the accuracy is much better than the algorithm in Chapter 2 although this has no significance for the algorithms in this thesis that assume an underlying pulse mechanism.

Summary of the Algorithms Relationships and their Complexities

Algorithm Type	Chapter	Synchronization that is Assumed	Precision	Accuracy	Convergence Time (d units)	Messages per cycle
Pulse Synch	Ch. 2	Bounded Delay	d and $3d$	0.5 *Cycle*	$O(f^3)$	n and $O(n^3)$
Pulse Synch	Ch. 7	SS. Byz. Agreement	$3d$	$O(d)$	$6f$	$O(n^3)$
Stabilizer	Ch. 3	Pulse Synch	that of pulse	that of pulse	$O(f')$	$O(f' \cdot n^2)$
Clock	Ch. 4	Pulse Synch	$11d + O(\rho)$	$11d + O(\rho)$	$3(2f' + 5)$	$O(nf'^2)$
Approx Clock	Ch. 4	Pulse Synch	$3d + O(\rho)$	$3d + O(\rho)$	$O(f')$	$O(nf'^2)$
Token	Ch. 5	Pulse Synch	that of pulse	that of pulse	$3(2f' + 5)$	$O(nf'^2)$
Agreement	Ch. 6	Bounded Delay	termination $3d$	NA	$O(f')$	$O(n^3)$

Table 1.1: Comparison of the complexities of the different algorithms developed in the thesis and their relationships. It shows for each algorithm its assumed prior synchronization. E.g. the clock synchronization algorithm assumes that there are synchronized pulses whereas the agreement algorithm assumes nothing beyond the minimal requirement of a bounded delay network. The convergence time and message complexities are not inclusive of the underlying synchronization primitives that are assumed. E.g. the stated message complexity of the clock synchronization does not include the messages of the pulse synchronization as it can use any pulse synchronization procedure. On the other hand, the specific precision and accuracy presented for the clock synchronization is derived from using one of the pulse synchronization algorithms in this thesis.

Chapter 2

Self-stabilizing Byzantine Pulse Synchronization Inspired by Biological Pacemaker Networks

We present a novel algorithm in the settings of self-stabilizing distributed algorithms, instructing the nodes how and when to invoke a pulse in order to meet the synchronization requirements of "Pulse Synchronization". The core elements of the algorithm are analogous to the neurobiological principles of *endogenous* (self generated) *periodic spiking, summation* and *time dependent refractoriness*. The basic algorithm is quite simple: every node invokes a pulse regularly and sends a message upon invoking it (*endogenous periodic spiking*). The node sums messages received from other nodes in some "window of time" (*summation*) and compares this to the continuously decreasing time dependent firing threshold for invoking a new pulse (*time dependent refractory function*). The node fires when the count of the summed messages crosses the current threshold level, and then resets its cycle. For in-depth explanations of these neurobiological terms see [50].

2.1 Specific Definitions

The nodes regularly invoke "pulses", ideally every *Cycle* real-time units. The invocation of the pulse is expressed by sending a message to all the nodes; this is also referred to as **firing**.

Basic definitions and notations:

We use the following notations though nodes do not need to maintain all of them as variables.

- *message_decay* represents the maximal real-time a non-faulty node will keep a message or a reference to it, before deleting it[1].

DEFINITION 2.1.1 *A node is* **correct** *following* $\Delta_{node} = cycle_{\max} + \sigma +$ *message_decay real-time of continuous non-faulty behavior.*

DEFINITION 2.1.2 *The communication network is* **correct** *following* $\Delta_{net} = cycle_{\max} + \sigma +$ *message_decay real-time of continuous non-faulty behavior.*

[1] The exact elapsed time until deleting a messages is specified in the PRUNE procedure in Fig. 2.2.

In accordance with Definition 1.2.2, the network model in this thesis is such that every message sent or received by a non-faulty node arrives within bounded time, δ, at all non-faulty nodes. The algorithm and its respective proofs are specified in a stronger network model in which every message received by a non-faulty node arrives within δ time at all non-faulty nodes. The subtle difference in the latter definition equals the assumption that every message received by a non-faulty node, even a message from a Byzantine node, will eventually reach every non-faulty node. This weakens the possibility for two-faced behavior by Byzantine nodes. The algorithm is able to utilize this fact so that if executed in such a network environment, then it can attain a very tight pulse synchronization of d real-time units. We show in Subsection 2.2 how to execute in the background a self-stabilizing Byzantine reliable-broadcast-like primitive, which executes in the network model of Definition 1.2.2. This primitive effectively relays every message received by a non-faulty node so that the latter network model is satisfied. In such a case the algorithm can be executed in the network model of Definition 1.2.2 and achieves synchronization of the pulses to within $3d$ real-time.

2.2 The "Pulse Synchronization" Algorithm

We now present the BIO-PULSE-SYNCH algorithm that solves the "Pulse Synchronization" problem defined in Definition 1.2.6, inspired by and following a neurobiological analog. The **refractory function** describes the time dependency of the firing threshold. At threshold level 0 the node invokes a pulse (*fires*) **endogenously**. The algorithm uses several sub-procedures. With the help of the **SUMMATION** procedure, each node sums the pulses that it learns about during a recent time window. If this sum (called the *Counter*) crosses the current (time-dependent) threshold for firing, then the node will fire, i.e broadcasts its Counter value at the firing time. The exact properties of the time window for summing messages is determined by the message decay time in the PRUNE procedure (see Fig. 2.2).

We now show in greater detail the elements and procedures described above.

The refractory function

The *Cycle* is the predefined time a correct node will count on its timer before invoking an endogenous pulse. The refractory function, $REF(t) : t \rightarrow \{0..n{+}1\}$, determines at every moment the threshold for invoking a new pulse. The refractory function is determined by the parameters *Cycle* n, f, d and ρ. All correct nodes execute the same protocol with the same parameters and have the same refractory function. The refractory function is shaped as a monotonously decreasing step function comprised of $n+2$ steps, $REF \equiv (R_{n+1}, R_n, ..., R_0)$, where step $R_i \in \mathbb{R}^+$ is the time length on the node's timer of threshold level i. The refractory function REF, starts at threshold level $n+1$ and decreases with time towards threshold level 0. The time length of each threshold step is formulated in Eq. 2.1:

$$R_i = \begin{cases} \frac{\frac{1}{1-\rho}Cycle}{n-f} & i = 1 \dots n-f-1 \\ \frac{R_1 - R_{n+1} - \frac{\rho}{1-\rho}Cycle}{f+1} & i = n-f \dots n \\ 2d(1+\rho) \cdot \frac{(\frac{1+\rho}{1-\rho})^{n+3}-1}{(\frac{1+\rho}{1-\rho})-1} & i = n+1. \end{cases} \tag{2.1}$$

Subsequent to a pulse invocation the refractory function is restarted at $REF = n+1$. The node will then commence threshold level n only after measuring R_{n+1} time units on its timer. Threshold level 0 ($REF = 0$) is reached only if exactly *Cycle* time units have elapsed on a node's timer since the last pulse invocation, following which threshold level $n+1$ is reached immediately. Hence, by definition, $\sum_{i=1}^{n+1} R_i \equiv$ *Cycle*. It is proven later in Lemma 2.3.2 that REF in Eq. 2.1 is consistent with this.

The special step R_{n+1} is called the **absolute refractory period** of the cycle. Following the neurobiological analogue with the same name, this is the first period after a node fires, during which its threshold level is in practice "infinitely high"; thus a node can never fire within its absolute refractory period.

See Fig. 2.7 for a graphical presentation of the refractory function and its role in the main algorithm.

The message sent when firing

The content of a message M_p sent by a node p, is the Counter, which represents the number of messages received within a certain time window (whose exact properties are described in the proof section in Section 2.6) that triggered p to fire. We use the notation $Counter_p$ to mark the local Counter at node p and $Counter_{M_p}$ to mark the Counter contained in a received message M_p sent by node p.

The SUMMATION procedure

A full account of the proof of correctness of the SUMMATION procedure is provided in Section 2.6. The SUMMATION procedure is executed upon the arrival of a new message. Its purpose is to decide whether this message is eligible for being counted. It is comprised of the following sub-procedures:

Upon arrival of the new message, the **TIMELINESS** procedure determines if the Counter contained in the message seems "plausible" (*timely*) with respect to the number of other messages received recently (it also waits a short time for such messages to possibly arrive). The bound on message transmission and processing time among correct nodes allows a node to estimate whether the content of a message it receives is plausible and therefore timely. For example, it does not make sense to consider an arrived message that states that it was sent as a result of receiving $2f$ messages, if less than f messages have been received during a recent time window. Such a message is clearly seen as a faulty node by all correct nodes. On the other, a message that states that it was sent as a result of receiving $2f$ messages, when $2f-1$ messages have been received during a recent time window does not bear enough information to decide whether it is faulty or not, as other correct nodes may have decided that this message is timely, due to receiving a faulty message. Such a message needs to be temporarily tabled so that it can be reconsidered for being counted in case some correct node sends a message within a short time, and which has counted that faulty message. Thus, intuitively, a message will be timely if the Counter in that messages is less or equal to the

total number of tabled or timely messages that were received within a short recent time window. The exact length of the "recent" time window is a crucial factor in the algorithm. There is no fixed time after which a message is too old to be timely. The time for message exchange between correct nodes is never delayed beyond the network and processing delay. Thus, the fire of a correct node, as a consequence of a message that it received, adds a bounded amount of relay time. This is the basis for the time window within which a specific Counter of a message is checked for plausibility. Hence, a particular Counter of a message is plausible only if there is a sufficient number of other messages (tabled or not) that were received within a sufficiently small time window to have been relayed from one to the other within the bound on relaying between correct nodes. As an example, consider that the bound on the allowed relay interval of messages is taken to be $2d$ time units. Suppose that a correct node receives a message with Counter that equals k. That message will only be considered as timely if there are at least $k + 1$ messages that were received (including the last one) in the last $k \cdot 2d$ time window. This is the main criterion for being timely. On termination of the procedure the message is said to have been **assessed**.

If a message is assessed as timely then the **MAKE-ACCOUNTABLE** procedure determines by how much to increment the Counter. It does so by considering the minimal number of recently tabled messages that were needed in order to assess the message as timely. This number is the amount by which the Counter is incremented by. A tabled message is marked as "uncounted" because the node's Counter does not reflect this message. Tabled messages that are used for assessing a message as timely become marked as "counted" because the node's Counter now reflect these message as if they were initially timely. A node's Counter at every moment is exactly the number of messages that are marked as "counted" at that moment.

The **PRUNE** procedure is responsible for the tabling of messages. A correct node wishes to mark as counted, only those messages which considering the elapsed time since their arrival, will together pass the criterion for being timely at any correct nodes receiving the consequent Counter to be sent. Thus, messages that were initially assessed as timely are tabled after a short while. This is what causes the Counter to dissipate. After a certain time messages are deleted altogether (*decayed*).

```
SUMMATION(a new message M_p arrived at time t_arr)   /* at node q */

if (TIMELINESS(M_p, t_arr) == "M_p is timely") then
  MAKE-ACCOUNTABLE(M_p); /* possibly increment Counter_q */
  PRUNE(t);
```

Figure 2.1: The SUMMATION procedure

The target of the SUMMATION procedure is formulated in the following two properties:

Summation Properties: Following the arrival of a message from a correct node:

P1: The message is assessed within d real-time units.

P2: Following assessment of the message the receiving node's Counter is incremented to hold a value greater than the Counter in the message.

The SUMMATION procedure satisfies the Summation Properties by the following heuristics:

- When the Counter crosses the threshold level, either due to a sufficient counter increment or a threshold decrement, then the node sends a message (fires). The message sent holds the value of Counter at sending time.
- The TIMELINESS procedure is employed at the receiving node to assess the credibility (timeliness) of the value of the Counter contained in this message. This procedure ensures that messages sent by correct nodes with Counter less than n will always be assessed as timely by other correct nodes receiving this message.
- When a received message is declared timely and therefore accounted for it is stored in a "counted" message buffer ("Counted Set"). The receiving node's Counter is then updated to hold a value greater than the Counter in the message by the MAKE-ACCOUNTABLE procedure.
- If a message received is declared untimely then it is temporarily stored in an "uncounted" message buffer ("Uncounted Set") and will not be accounted for at this stage. Over time, the timeliness test of previously stored timely messages may not hold any more. In this case, such messages will be moved from the Counted Set to the Uncounted Set by the PRUNE procedure.
- All messages are deleted after a certain time-period (message decay time) by the PRUNE procedure.

Definitions and state variables:

Counter: an integer representing the node's estimation of the number of timely firing events received from distinct nodes within a certain time window. Counter is updated upon receiving a timely message. The node's Counter is checked against the refractory function whenever one of them changes. The value of Counter is bounded and changes non-monotonously; the arrival of timely events may increase it and the decay/untimeliness of old events may decrease it.

Stored message: is a basic data structure represented as (S_p, t_{arr}) and created upon arrival of a message M_p. S_p is the id (or signature) of the sending node p and t_{arr} is the local arrival time of the message. We say that two stored messages, (S_p, t_1) and (S_q, t_2), are **distinct** if $p \neq q$.

Counted Set (CS): is a set of distinct stored messages that determine the current value of Counter. The Counter reflects the number of stored messages in the Counted Set. A stored message is **accounted for** in Counter, if it was in CS when the current value of Counter was determined.

Uncounted Set (UCS): is a set of stored messages, not necessarily distinct, that have not been accounted for in the current value of Counter and that are not yet due to decay. A stored message is placed (tabled) in the UCS when its message clearly reflects a faulty sending node (such as when multiple messages from the same node are received) or because it is not timely anymore.

Retired UCS (RUCS): is a set of distinct stored messages not accounted for in the current value of Counter due to the elapsed local time since their arrival. These stored messages are awaiting

deletion (decaying).

The CS and UCS are mutually exclusive and together reflect the messages received from other nodes in the preceding time window. Their union is denoted the node's ***Message_Pool.***

$\boldsymbol{t_{send\,M_p}}$: denotes the local-time at which a node p sent a message M_p. An equivalent definition of $t_{send\,M_p}$ is the local-time at which a receiving node p is ready to assess whether to send a message consequent to the arrival and processing of some other message.

$\boldsymbol{MessageAge(t,q,p)}$: is the elapsed time, at time t, on a node q's clock since the most recent arrival of a message from node p, which arrived at local-time t_{arr}. Thus, its value at node q at current local-time t is given by $t - t_{arr}$, where M_p is the most recent message that arrived from p. If no stored message is held at q for p then $MessageAge(t,q,p) = \infty$.

$\boldsymbol{CSAge(t)}$: denotes, at local-time t, the largest $MessageAge(t,q,\ldots)$ among the stored messages in CS of node q.

$\boldsymbol{\tau}$: denotes the function $\tau(k) \equiv 2d(1+\rho)\frac{(\frac{1+\rho}{1-\rho})^{k+1}-1}{(\frac{1+\rho}{1-\rho})-1}$.

The set of procedures used by the SUMMATION **procedure (at node q):**

```
The following procedure moves and deletes obsolete stored
messages. It prunes the CS to hold only stored messages such
that a message sent holding the resultant Counter will be
assessed as timely at any correct receiving node.

PRUNE (t)                                        /* at node q  */
```

- Delete from RUCS all entries (S_p,t) whose $MessageAge(t,q,p) > \tau(n+2)$;
- Move to RUCS, from the *Message_Pool*, all stored messages (S_p,t) whose $MessageAge(t,q,p) > \tau(n+1)$;
- Move to UCS, from CS, stored messages, beginning with the oldest, until: $CSAge(t) \leq \tau(k-1)$, where $k = \max[1,\ \|CS\|]$;
- Set $Counter := \|CS\|$;

Figure 2.2: The PRUNE procedure

We say that M_p has been **assessed** by q, once the following procedure is completed. A message M_p, is timely at local-time t_{arr} at node q once it is declared timely by the procedure, i.e. 1: whether the Counter in the message is within its valid range; 2: whether the sending node has recently sent a message, in which only the latest is considered; 3: whether enough messages have been received recently to support the credibility of the Counter in the message.

TIMELINESS (M_p, t_{arr}) /* at node q */

/* check if Counter is valid */

Timeliness Condition 1:

If $(0 \leq Counter_{M_p} \leq n-1)$ Then

Create a new stored message (S_p, t_{arr}) and insert it into UCS;

Else

return "M_p is not timely";

/* if an older message from same node already exists then must be a faulty node. Delete all its entries but the latest. */

Timeliness Condition 2:

If $(\exists (S_p, t)$, s.t. $t \neq t_{arr}$, in *Message_Pool* $\cup$ RUCS) Then[a]

delete from *Message_Pool* all (S_p, t'), where $t' \neq t_{arr}$;

return "M_p is not timely";

/* check if $Counter_{M_p}$ seems credible with respect to the *Message_Pool* */

Timeliness Condition 3:

Let k denote $Counter_{M_p}$.

If (at some local-time t in the interval $[t_{arr},\ t_{arr} + d(1+\rho)]$: $\|\{(S_r, t') | (S_r, t') \in Message\_Pool, MessageAge(t, q, r) \leq \tau(k+1)\}\| \geq k+1)$ Then[b]

return "M_p is timely";

Else

return "M_p is not timely";

[a] We assume no concomitant messages are stamped with the exact same arrival times at a correct node. We assume that one can uniquely identify messages.

[b] We assume the implementation can assess these conditions within the time window.

Figure 2.3: The TIMELINESS procedure

This procedure moves stored messages from UCS into CS and updates the value of Counter. This is done in case the arrival of a new timely message M_p, has made previously uncounted stored messages eligible for being counted.

MAKE-ACCOUNTABLE (M_p) /* at node q */

- Move the $\max[1,\ (Counter_{M_p} - Counter_q + 1)]$ most recent distinct stored messages from UCS to CS;
- Set $Counter := \|CS\|$;

Figure 2.4: The MAKE-ACCOUNTABLE procedure

```
This procedure causes the effective cycle of the node to
be reset, meaning that the REF function starts the cycle
from the highest threshold level again and down to threshold
level 0.

CYCLE-RESET ()                                    /* at node q  */
    • Restart REF at REF := n+1;
```

Figure 2.5: The CYCLE-RESET procedure

We now cite the main theorems of the SUMMATION procedure. The proofs are given in Section 2.6.

Theorem 2.2.1 *Any message, M_p, sent by a correct node p will be assessed as timely by every correct node q.*

Lemma 2.2.1 *Following the arrival of a timely message M_p, at a node q, then at time $t_{send\,M_q}$, $Counter_q > Counter_{M_p}$.*

Theorem 2.2.2 *The* SUMMATION *procedure satisfies the Summation Properties.*

Proof: Let p denote a correct node that sends M_p. Theorem 2.2.1 ensures that M_p is assessed as timely at every correct node. Lemma 2.2.1 ensures that the value of $Counter$ will not decrease below $Counter_{M_p} + 1$ until local-time $t_{send\,M_p}$, thereby satisfying the Summation Properties. □

The event driven "pulse synchronization" algorithm

Fig. 2.6 shows the main algorithm. Fig. 2.7 illustrates the mode of operation of the main algorithm.

```
BIO-PULSE-SYNCH(n, f, Cycle)                    /* at node q */

• It is assumed that all the parameters and variables
are verified to be within their range of validity.
• t is the local-time at the moment of executing the
respective statement.

if (a new message M_p arrives at time t_arr) then
  SUMMATION ((M_p, t_arr));
  if (Counter_q ≥ REF(t)) then
    Broadcast Counter_q to all nodes; /* invocation of the
Pulse */
    CYCLE-RESET();

if (change in threshold level according to REF) then
  PRUNE (t);
  if (Counter_q ≥ REF(t)) then
    Broadcast Counter_q to all nodes; /* invocation of the
Pulse */
    CYCLE-RESET();
```

Figure 2.6: The event driven BIO-PULSE-SYNCH algorithm

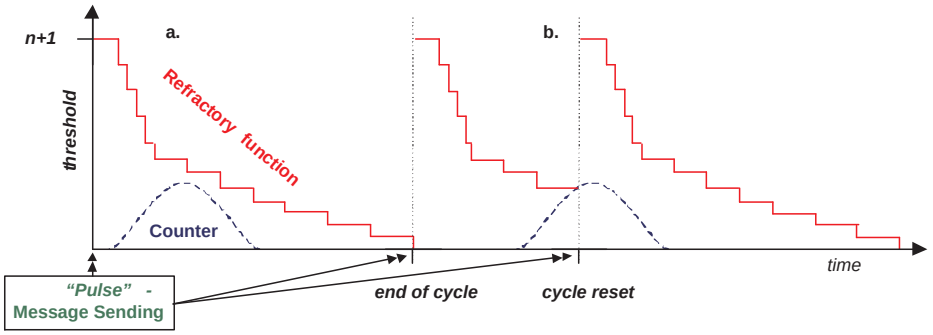

Figure 2.7: Schematic example of the mode of operation of BIO-PULSE-SYNCH: **(a.)** The node's Counter (the summed messages) does not cross the threshold during the cycle, letting the refractory function reach zero and consequently the node fires endogenously. **(b.)** Sufficient messages from other nodes are received in time window for the Counter to surpass the current threshold, consequently the node fires early and resets its cycle.

A Reliable-Broadcast Primitive

In the current subsection we show that the BIO-PULSE-SYNCH algorithm can also operate in networks in which Byzantine nodes may exhibit true two-faced behavior. This is done by executing in the background a self-stabilizing Byzantine reliable-broadcast-like primitive, which assumes no

synchronicity whatsoever among the nodes. It has the property of relaying any message received by a correct node. Hence, this primitive satisfies the broadcast assumption of Definition 1.2.2 by supplying a property similar to the relay property of the reliable-broadcast primitive in [73]. That latter primitive assumes a synchronous initialization and can thus not be used as a building block for a self-stabilizing algorithm.

In Chapter 6 we present the INITIATOR-ACCEPT primitive. We say that a node does an **I-accept** of a message m sent by some node p (denoted $\langle p, m\rangle$) if it accepts that this message was sent by node p.

The INITIATOR-ACCEPT primitive essentially satisfies the following two properties (rephrased for our purposes):

IA-1A (*Correctness*) If all correct nodes invoke INITIATOR-ACCEPT $\langle p, m\rangle$ within d real-time of each other then all correct nodes I-accept$\langle p, m\rangle$ within $2d$ real-time units of the time the last correct node invokes the primitive INITIATOR-ACCEPT $\langle p, m\rangle$.

IA-3A (*Relay*) If a correct node q I-accepts $\langle p, m\rangle$ at real-time t, then every correct node q' I-accepts $\langle p, m\rangle$, at some real-time t', with $|t - t'| \leq 2d$.

The INITIATOR-ACCEPT primitive requires a correct node not to send two successive messages within less than $6d$ real-time of each other. Following the BIO-PULSE-SYNCH algorithm (see Timeliness Condition 2, in the TIMELINESS procedure), non-faulty nodes cannot fire more than once in every $2d(1 + \rho) \cdot n > 6d$ real-time interval even if the system is not coherent, which thus satisfies this requirement.

The use of the INITIATOR-ACCEPT primitive in our algorithm is by executing it in the background. When a correct node wishes to send a message it does so through the primitive, which has certain conditions for I-accepting a message. Nodes may also I-accept messages that where not sent or received through the primitive, if the conditions are satisfied. In our algorithm nodes will deliver messages only after they have been I-accepted (also for the node's own message). From [IA-1A] we get that all messages from correct nodes are delivered within $3d$ real-time units subsequent to sending. From [IA-3A] we have that all messages are delivered within $2d$ real-time units of each other at all correct nodes, even if the sender is faulty. Thus, we get that the new network delay $\tilde{d} = 3d$. Hence, the cost of using the INITIATOR-ACCEPT primitive is an added $2d$ real-time units to the achieved pulse synchronization tightness which hence becomes $\sigma = \tilde{d} = 3d$.

2.3 Proof of Correctness of BIO-PULSE-SYNCH

In this section we prove Closure and Convergence of the BIO-PULSE-SYNCH algorithm. The proof that BIO-PULSE-SYNCH satisfies the pulse synchronization problem follows the steps below:

Subsection 2.3.1.1 introduces some notations and procedures that are for proof purposes only. One such procedure partitions the correct nodes into disjoint sets of synchronized nodes ("synchronized clusters").

In Subsection 2.3.1.2 (Lemma 2.3.4), we prove that "synchronized clusters" once formed stay as synchronized sets of nodes, this implies that once the system is in a synchronized_pulse_state it remains as such (*Closure*).

In Subsection 2.3.1.3 (Theorem 2.3.3), we prove that within a finite number of cycles, the synchronized clusters repeatedly absorb to form ever larger synchronized sets of nodes, until a synchronized_pulse_state of the system is reached (*Convergence*).

Note that the the synchronization tightness, σ, of our algorithm, equals d.

It may ease following the proofs by thinking of the algorithm in the terms of non-liner dynamics, though this is not necessary for the understanding of any part of the protocol or its proofs. We show that the state space can be divided into a small number of stable fixed points ("synchronized sets") such that the state of each individual node is attracted to one of the stable fixed points. We show that there are always at least two of these fixed points that are situated in the basins of attraction ("absorbance distance") of each other. Following the dynamics of these attractors, we show that eventually the states of all nodes settle in a limit cycle in the basin of one attractor.

2.3.1.1 Notations, procedures and properties used in the proofs

First node in a synchronized set of nodes S, is a node of the subset of nodes that "fire first" in S that satisfies:

$$\text{"First node in } S\text{"} = \begin{cases} \min\{i | i \in \max\{\phi_i(t) | \text{node } i \in S, \phi_i(t) \leq \sigma\}\} & \exists i \in S \text{ s.t. } \phi_i(t) \leq \sigma \\ \min\{i | i \in \max\{\phi_i(t) | \text{node } i \in S\}\} & \text{otherwise.} \end{cases}$$

Equivalently, we define **last node**:

$$\text{"Last node in } S\text{"} = \begin{cases} \max\{i | i \in \min\{\phi_i(t) | \text{node } i \in S, \phi_i(t) > \sigma\}\} & \exists i \in S \text{ s.t. } \phi_i(t) > \sigma \\ \max\{i | i \in \min\{\phi_i(t) | \text{node } i \in S\}\} & \text{otherwise.} \end{cases}$$

The second cases in both definitions serve to identify the First and Last nodes in case t falls in-between the fire of the nodes of the set.

Synchronized Clusters

At a given time t the nodes are divided into disjoint **synchronized clusters** in the following way:

1. Assign the maximal synchronized set of nodes at time t as a synchronized cluster. In case there are several maximal sets choose the set that is harboring the first node of the unified set of all these maximal sets.

2. Assign the second maximal synchronized set of nodes that are not part of the first synchronized cluster as a synchronized cluster.

3. Continue until all nodes are exclusively assigned to a synchronized cluster.

The synchronized cluster harboring the node with the largest (necessarily finite) ϕ among all the nodes is designated C_1. The rest of the synchronized clusters are enumerated inversely to the ϕ of their first node, thus if there are m synchronized clusters then C_m is the synchronized cluster whose first node has the lowest ϕ (besides perhaps C_1). Note that at most one synchronized cluster may have nodes whose actual ϕ differences is larger than σ, as it can contain nodes that have just fired and nodes just about to fire. The definition of C_1 implies that at the time the nodes are partitioned into synchronized clusters (time t above) it may be the only synchronized cluster in such a state.

The clustering is an external observation done only for illustrative purposes of the proof. It does not actually affect the protocol or the behavior of the nodes. In the proof we "assign" the nodes to synchronized clusters at some time t. From that time on we consider the synchronized clusters as a constant partitioning of the nodes into disjoint synchronized sets of nodes and then observe how these clusters evolve as the system moves from one state to the other. Thus, once a node is exclusively assigned to some synchronized cluster it will stay a member of that synchronized cluster. We aim at showing that eventually all synchronized clusters become one synchronized set of nodes. Once such a clustering is fixated we ignore nodes that happen to fail and forthcoming recovering nodes. Our proof is based on the observation that eventually we reach a time window within which the permanent number of non-correct nodes at every time is bounded by f and during that window the whole system converges.

OBSERVATION 2.3.1 *The synchronized clustering procedure assigns every correct node to exactly one synchronized set of nodes.*

OBSERVATION 2.3.2 *Immediately following the synchronized clustering procedure no two distinct synchronized clusters comprise one synchronized set of nodes.*

We use the following definitions and notations:

- C_i – synchronized cluster number i.
- n_i – cardinality of C_i (i.e. number of correct nodes associated with synchronized cluster C_i).
- c – current number of synchronized clusters in the current state; $c \geq 1$.
- $dist(a, b, t) \equiv |\phi_a(t) - \phi_b(t)|$ is the **distance** (ϕ difference) between nodes a and b at real-time t.
- $\phi_{c_i}(t)$ – is the $\phi(t)$ of the first node in synchronized cluster C_i.
- $dist(C_i, C_j, t) \equiv dist(\phi_{c_i}(t),\ \phi_{c_j}(t),\ t)$ at real-time t.

If at real-time t there exists no other synchronized cluster C_r, such that $\phi_{c_i}(t) \geq \phi_{c_r}(t) \geq \phi_{c_j}(t)$, then we say that the synchronized clusters C_i and C_j are **adjacent** at real-time t.

We say that two synchronized clusters, C_i and C_j, have **absorbed** if their union comprises a synchronized set of nodes. If a node in C_j fires due to a message received from a node in C_i, then, as will be shown in Lemma 2.3.7, the inevitable result is that their two synchronized clusters absorb. The course of action from the arrival of the message at a node in C_j until C_j has absorbed with C_i is referred to as the **absorbance** of C_j by C_i.

We refer throughout the chapter to the **fire of a synchronized cluster** instead of referring to the sum of the fires of the individual nodes in the synchronized cluster. In Lemma 2.6.8 we prove that these two notations are equivalent.

In Theorem 2.3.1 we show that we can explicitly determine a threshold value, $\mathbf{ad(C_i)}$, that has the property that if for two synchronized clusters C_i and C_j, $dist(C_i, C_j, t) \leq ad(C_i)$ then C_i **absorbs** C_j. We will call that value the "**absorbance distance**" of C_i.

DEFINITION 2.3.1 *The absorbance distance, $ad(C_i)$, of a synchronized cluster C_i, is*

$$ad(C_i) \equiv \sum_{g=f+1}^{f+n_i} R_g$$

real-time units.

Properties used for the proofs

We identify and prove several properties; one property of the SUMMATION procedure (Property 1) and several properties of REF (Properties 2-7). These are later used to prove the correctness of the algorithm.

Property 1: See the Summation Properties in Subsection 2.2.
Property 2: R_i is a monotonic decreasing function of i, $R_i \geq R_{i+1}$, for $i = 1 \ldots n-1$.

Property 3: $R_i > 3d + \frac{2\rho}{1-\rho^2} \sum_{j=1}^{n+1} R_j$, for $i = 1 \ldots n-f-1$.

Property 4: $R_i > \sigma(1-\rho) + \frac{2\rho}{1+\rho} \sum_{j=1}^{n+1} R_j$, for $i = 1 \ldots n$.

Property 5: $R_{n+1} \geq 2d(1+\rho)^{\frac{(\frac{1+\rho}{1-\rho})^{n+3}-1}{(\frac{1+\rho}{1-\rho})-1}}$.

Property 6: $R_1 + \cdots + R_{n+1} = \mathit{Cycle}$.

Consider any clustering of $n-f$ correct nodes into $c > 1$ synchronized clusters, in which j' denotes the largest synchronized. Thus $n_{j'}$ is the number of nodes in the largest synchronized cluster and is less or equal to $n-f-1$. The number of nodes in the second largest cluster is less or equal to $\lfloor \frac{(n-f)}{2} \rfloor$.

Property 7:

$$\sum_{j=1, j\neq j'}^{c} \sum_{g=f+1}^{f+n_j} R_g + \sum_{g=1}^{n_{j'}} R_g \geq \frac{1}{1-\rho} \mathit{Cycle} \text{ , where } \sum_{j=1}^{c} n_j = n-f \text{ .} \tag{2.2}$$

We require the following restriction on the relationship between *Cycle*, d, n and f in order to prove that Properties 3-4 hold:

Restriction 1:

$$\mathit{Cycle} > d \cdot \frac{(1-\rho^2)[(1-\rho)(f+1) + 2(1+\rho) \cdot \frac{(\frac{1+\rho}{1-\rho})^{n+3}-1}{(\frac{1+\rho}{1-\rho})-1}]}{\frac{1-\rho}{n-f} - 3\rho + \rho^2} \text{ .} \tag{2.3}$$

We now prove that Properties 2-7 are properties of REF:

Lemma 2.3.1 *Properties 2-5 are properties of REF under Restriction 1.*

Proof: The proof for Properties 2 and 5 follows immediately from the definition of REF in Eq. 2.1.

Note that $R_i > R_j$, for $1 \leq i \leq n-f-1$ and $n-f \leq j \leq n$. Moreover, for $\sigma = d$, Property 4 is more restrictive than Property 3. Hence, for showing that Properties 3 and 4 are properties of REF it is sufficient to show that R_j (where $n-f \leq j \leq n$) satisfies Property 4:

$$R_j = \frac{R_1 - R_{n+1} - \frac{\rho}{1-\rho}Cycle}{f+1} > \sigma(1-\rho) + \frac{2\rho}{1+\rho}\sum_{j=1}^{n+1} R_j \Rightarrow$$

$$\frac{\frac{1}{1-\rho}\mathit{Cycle}}{n-f} - 2d(1+\rho)\cdot\frac{(\frac{1+\rho}{1-\rho})^{n+3}-1}{(\frac{1+\rho}{1-\rho})-1} - \frac{\rho}{1-\rho}Cycle > [d(1-\rho) + \frac{2\rho}{1+\rho}\mathit{Cycle}](f+1) \Rightarrow$$

$$\frac{1}{1-\rho}\mathit{Cycle} - \frac{\rho}{1-\rho}(n-f)\mathit{Cycle} - \frac{2\rho}{1+\rho}(n-f)\mathit{Cycle}$$
$$> [d(1-\rho)(f+1) + 2d(1+\rho)\cdot\frac{(\frac{1+\rho}{1-\rho})^{n+3}-1}{(\frac{1+\rho}{1-\rho})-1}](n-f) \Rightarrow$$

$$[\frac{1-\rho(n-f)}{1-\rho} - \frac{2\rho}{1+\rho}(n-f)]\mathit{Cycle}$$
$$> d[(1-\rho)(f+1) + 2(1+\rho)\cdot\frac{(\frac{1+\rho}{1-\rho})^{n+3}-1}{(\frac{1+\rho}{1-\rho})-1}](n-f) \Rightarrow$$

$$[\frac{(\frac{1}{n-f}-\rho)(1+\rho) - 2\rho(1-\rho)}{1-\rho^2}]\mathit{Cycle} > d[(1-\rho)(f+1) + 2(1+\rho)\cdot\frac{(\frac{1+\rho}{1-\rho})^{n+3}-1}{(\frac{1+\rho}{1-\rho})-1}] \Rightarrow$$

$$\frac{\frac{1-\rho}{n-f} - 3\rho + \rho^2}{1-\rho^2}\mathit{Cycle} > d[(1-\rho)(f+1) + 2(1+\rho)\cdot\frac{(\frac{1+\rho}{1-\rho})^{n+3}-1}{(\frac{1+\rho}{1-\rho})-1}] \Rightarrow$$

$$\mathit{Cycle} > d\cdot\frac{(1-\rho^2)[(1-\rho)(f+1) + 2(1+\rho)\cdot\frac{(\frac{1+\rho}{1-\rho})^{n+3}-1}{(\frac{1+\rho}{1-\rho})-1}]}{\frac{1-\rho}{n-f} - 3\rho + \rho^2} \quad . \tag{2.4}$$

This inequality is exactly satisfied by Restriction 1 and thus Eq. 2.1 satisfies Properties 3 and 4.

Note that for $\rho = 0$, the inequality becomes $\mathit{Cycle} > d\cdot(f+1)(n-f)$. □

Lemma 2.3.2 *Property 6 is a property of* REF.

Proof:

$$\begin{aligned} R_1 + \cdots + R_{n+1} &= (R_1 + \cdots + R_{n-f-1}) + (R_{n-f} + \cdots + R_n) + R_{n+1} \\ &= (n-f-1)\cdot\frac{\frac{1}{1-\rho}\mathit{Cycle}}{n-f} + (f+1)\cdot\frac{R_1 - R_{n+1} - \frac{\rho}{1-\rho}\mathit{Cycle}}{f+1} + R_{n+1} \\ &= \frac{1}{1-\rho}\mathit{Cycle} - \frac{\frac{1}{1-\rho}\mathit{Cycle}}{n-f} + R_1 - R_{n+1} - \frac{\rho}{1-\rho}\mathit{Cycle} + R_{n+1} = \mathit{Cycle} \ . \end{aligned}$$

□

Lemma 2.3.3 *Property 7 is a property of* REF.

Proof: We will prove that the constraint in Eq. 2.2 is always satisfied by the refractory function in Eq. 2.1.

Note that Eq. 2.2 is a linear equation of the R_i values of REF. We denoted $n_{j'}$ to be the number of nodes in the largest synchronized cluster, following some partitioning of the correct nodes into synchronized clusters. We want to find what is the largest value of i such that R_i is a value with a non-zero coefficient in the linear equation Eq. 2.2. This value is determined by either the largest possible cluster, which may be of size $n-f-1$ (in case all but one of the correct nodes are in one synchronized cluster[2]), or by the second-largest possible cluster, which may be of size $\lfloor\frac{(n-f)}{2}\rfloor$ (in case all correct nodes are in two possibly equally sized synchronized clusters). Thus the largest value of i such that R_i is a value with a non-zero coefficient equals $\max[f+\lfloor\frac{(n-f)}{2}\rfloor,\ n-f-1] = n-f-1$, for $n \geq 3f+1$.

Thus, following Eq. 2.1, each of these R_i values equals $\frac{\frac{1}{1-\rho}Cycle}{n-f}$. There are exactly $n-f$ (not necessarily different) R_i values in Eq. 2.2. Hence, incorporating Eq. 2.1 into Eq. 2.2 reduces Eq. 2.2 to the linear equation: $(n-f)\cdot R_i \geq \frac{1}{1-\rho}Cycle$, where $1 \leq i \leq n-f-1$. It remains to show that Eq. 2.1 satisfies this constraint:

$$(n-f)\cdot R_i = (n-f)\cdot\frac{\frac{1}{1-\rho}Cycle}{n-f} = \frac{1}{1-\rho}Cycle.$$

□

2.3.1.2 Proving the Closure

We now show that a synchronized set of nodes stays synchronized. This also implies that the constituent nodes of a synchronized clusters stay as a synchronized set of nodes, as a synchronized cluster is in particular a synchronized set of nodes. This proves the first Closure requirement of the "Pulse Synchronization" problem in Definition 1.2.6.

Lemma 2.3.4 *A set of correct nodes that is a synchronized set at real-time t', remains synchronized $\forall t,\ t \geq t'$.*

Proof: Let there be a synchronized set of nodes at real-time t'. From the definition of a synchronized set of nodes, this set of nodes will stay synchronized as long as no node in the set fires. This is because the ϕ difference between nodes (in real-time units) does not change as long as none of them fires. We therefore turn our attention to the first occasion after t' at which a node from the set fires. Let us examine the extreme case of a synchronized set consisting of at least two nodes at the maximal allowed ϕ difference; that is to say that at time t', $dist(first\_node, last\_node, t') = \sigma$. Further assume that the first node in the set fires with a Counter=k, $(0 \leq k \leq n-1)$, at some time $t \geq t'$ at the very beginning of its threshold level k, and without loss of generality is also the first node in the set to fire after time t'. We will show that the rest of the nodes in the set will fire within the interval $[t, t+\sigma]$ and thus remains a synchronized set.

Property 1 ensures that the last node's Counter will read at least $k+1$ subsequently to the arrival and assessment of the first node's fire, since its Counter should be at least the first node's Counter

[2]The case in which the $n-f$ correct nodes are in one synchronized cluster implies the objective has been reached.

plus 1. The proof of the lemma will be done by showing that right after the assessment of the first node's fire, the last node cannot be at a threshold higher than $k+1$ and thus will necessarily fire.

The proof is divided into the following steps:

1. Show that when the first node is at threshold level k then the last node is at threshold level $k+1$ or lower.

2. Show that if the first node fires with a Counter=k then due to Property 1 and Step 1 the last node will fire consequently.

3. Show that the last node fires within a d real-time window of the first node, and as a result, the new distance between the first and last node is less than or equal to σ.

Observe that the extreme case considered is a worst case since if the largest ϕ difference in the set is less than σ then the threshold level of the last node may only be lower. The same argument also holds if the first node fires after its beginning of its threshold level k. Thus the steps of the proofs also apply to any intermediate node in the synchronized set and thus remains as a synchronized set of nodes.

Step 1: In this step we aim at calculating the amount of time on the last node's clock remaining until it commences its threshold level k, counting from the event in which the first nodes commences its threshold level k. By showing that this remaining time is less than the length of threshold level $k+1$, as counted on the clock of the last and slowest node we conclude that this node must be at most at threshold level $k+1$. The calculations are done on the slow node's clock.

Assume the first node to be the fastest permissible node and the last one the slowest. Hence, when the first node's threshold level k commences,

$$\frac{1}{1+\rho}\sum_{i=k+1}^{n+1} R_i \tag{2.5}$$

real-time units actually passed since it last fired. The last node "counted" this period as:

$$\frac{1-\rho}{1+\rho}\sum_{i=k+1}^{n+1} R_i \ . \tag{2.6}$$

The last node has to count on its clock, from the time that the first node fired, at most $\sigma(1-\rho)$ local-time units (max. ϕ difference of correct nodes in a synchronized set as counted by the slowest node), and

$$\sum_{i=k+1}^{n+1} R_i \tag{2.7}$$

in order to reach its own threshold level k. As a result, the maximum local-time difference between the time the first node starts its threshold level k till the last node starts its own threshold level k as counted by the last node is therefore $\sigma(1+\rho)$ plus the difference Eq. 2.7 – Eq. 2.6, which yields

$$\sigma(1-\rho)+\frac{1+\rho}{1+\rho}\sum_{i=k+1}^{n+1}R_i-\frac{1-\rho}{1+\rho}\sum_{i=k+1}^{n+1}R_i=\sigma(1-\rho)+\frac{2\rho}{1+\rho}\sum_{i=k+1}^{n+1}R_i\ . \tag{2.8}$$

Property 4 ensures that R_{k+1} is greater than Eq. 2.8 for $0 \leq k \leq n-1$; thus when the first node commences threshold level k the last node must be at a threshold level that is less or equal to $k+1$.

Step 2: Let the first node fire as a result of its Counter equalling k at time t at threshold level k. In case that the last node receives almost immediately the first node's fire (and thus increments its Counter to at least $k+1$ following Property 1), it must be at a threshold level that is less or equal to $k+1$ (following Step 1) and will therefore fire. All the more so if the first node's fire is received later, since the threshold level can only decrease in time before a node fires.

Step 3: We now need to estimate the new distance between the first and last node in order to show that they still comprise a synchronized set. The last node assesses the first node's fire within d real-time units after the first node sent its message (per definition of d). This yields a distance of $d(1-\rho)$ as seen by the last node, which equals the maximal allowed real-time distance, $d\ (=\sigma)$, between correct nodes in a synchronized set at real-time t', and thus they stay a synchronized set at time t'. □

Corollary 2.3.5 *(Closure 1) Lemma 2.3.4 implies the first Closure condition.*

Lemma 2.3.6 *(Closure 2) As long as the system state is in a synchronized_pulse_state then the second Closure condition holds.*

Proof: Due to Lemma 2.3.4 the first node to fire in the synchronized set following its previous pulse, may do so only if it receives the fire from faulty nodes or if it fires endogenously. This may happen the earliest if it receives the fire from exactly f distinct faulty nodes. Thus following Eq. 2.1 its cycle might have been shortened by at most $f \cdot \frac{Cycle}{n-f}$ real-time units. Hence, in case the first node to fire is also a fast node, it follows that $cycle_{min} = Cycle \cdot (1-\rho) - \frac{f}{n-f} \cdot Cycle \cdot (1-\rho) = \frac{n-2f}{n-f} \cdot Cycle \cdot (1-\rho)$ real-time units. A node may fire at the latest if it fires endogenously. If in addition it is a slow node then it follows that $cycle_{max} = Cycle \cdot (1+\rho)$ real-time units.

Thus in any real-time interval that is less or equal to $cycle_{min}$ any correct node will fire at most once. In any real-time interval that is greater or equal to $cycle_{max}$ any correct node will fire at least once. This concludes the second closure condition. □

2.3.1.3 Proving the Convergence

The proof of Convergence is done through several lemmata. We begin by presenting sufficient conditions for two synchronized clusters to absorb. In Subsection 2.3.1.4, we show that the refractory function REF ensures the continuous existence of a pair of synchronized clusters whose unified set of nodes is not synchronized, but are within an absorbance distance and hence absorb. Thus, iteratively, all synchronized clusters will eventually absorb to form a unified synchronized set of nodes.

Lemma 2.3.7 *(Conditions for Absorbance) Given two synchronized clusters, C_i preceding C_j, if:*

1. *C_i fires with Counter=k, at real-time $t_{c_i\_fires}$, where $0 \leq k \leq f$*

2. *$dist(C_i, C_j, t_{c_i\_fires}) \leq \frac{1}{1-\rho} \sum_{g=k+1}^{k+n_i} R_g - \frac{2\rho}{1-\rho^2} \sum_{g=k+1}^{n+1} R_g$*

then C_i will absorb C_j.

Proof: The proof is divided into the following steps:

1. (a) If C_i fires before C_j, then C_j consequently fires.

   (b) Subsequent to the previous step: $dist(C_i, C_j, ..) \leq 3d$.

2. Following the previous step, within one cycle the constituent nodes of the two synchronized clusters comprise a synchronized set of nodes.

Step 1a: Let us examine the case in which C_i fires first at some real-time denoted $t_{c_i\_fires}$, and in the worst case that C_j doesn't fire before it receives all of C_i's fire. All the calculations assume that at $t_{c_i\_fires}$, $\phi_{c_i}(t_{c_i\_fires})$ has still not been reset to 0. Specifically, assume that the first node in C_i fired due to incrementing its Counter to k $(0 \leq k \leq f)$ at the beginning of its threshold level k. Following Property 1 and Lemma 2.6.8 the nodes of C_j increment their Counters to $k + n_i$ after receiving the fire of C_i. Additionally, in the worst case, assume that the first node in C_j receives the fire of C_i almost immediately. We will now show that this fire is received at a threshold level $\leq k + n_i$.

We will calculate the upper-bound on the ϕ of the first node in C_j at real-time $t_{c_i\_fires}$, and hence deduce the upper-bound on its threshold level. Assume the nodes of C_i are fast and the nodes of C_j are slow. Should the nodes of C_j be faster, then the threshold level may only be lower.

$$
\begin{aligned}
&\phi_{c_j}(t_{c_i\_fires}) = \\
&= \phi_{c_i}(t_{c_i\_fires}) - [\frac{1}{1-\rho}\sum_{g=k+1}^{k+n_i} R_g - \frac{2\rho}{1-\rho^2}\sum_{g=k+1}^{n+1} R_g] \\
&= \frac{1}{1+\rho}\sum_{g=k+1}^{n+1} R_g - [\frac{1}{1-\rho}\sum_{g=k+1}^{k+n_i} R_g - \frac{2\rho}{1-\rho^2}\sum_{g=k+1}^{n+1} R_g] \\
&= \frac{1}{1+\rho}\sum_{g=k+1}^{n+1} R_g - [\frac{1}{1-\rho}\sum_{g=k+1}^{k+n_i} R_g + (\frac{1}{1+\rho} - \frac{1}{1-\rho})\sum_{g=k+1}^{n+1} R_g] \\
&= \frac{1}{1+\rho}\sum_{g=k+1}^{n+1} R_g - [(\frac{1}{1+\rho} - (\frac{1}{1+\rho} - \frac{1}{1-\rho}))\sum_{g=k+1}^{k+n_i} R_g + (\frac{1}{1+\rho} - \frac{1}{1-\rho})\sum_{g=k+1}^{n+1} R_g] \\
&= \frac{1}{1+\rho}\sum_{g=k+1}^{n+1} R_g - [\frac{1}{1+\rho}\sum_{g=k+1}^{k+n_i} R_g + (\frac{1}{1+\rho} - \frac{1}{1-\rho})\sum_{g=k+1+n_i}^{n+1} R_g] \\
&= \frac{1}{1+\rho}\sum_{g=k+1}^{n+1} R_g - [\frac{1}{1+\rho}\sum_{g=k+1}^{n+1} R_g - \frac{1}{1-\rho}\sum_{g=k+1+n_i}^{n+1} R_g] \\
&= \frac{1}{1-\rho}\sum_{g=k+1+n_i}^{n+1} R_g \ .
\end{aligned}
\tag{2.9}
$$

We now seek to deduce the bound on C_j's threshold level at the time of C_i's fire. Thus, following Eq. 2.9, at real-time $t_{c_i\_fires}$ the ϕ of the first node in C_j is at most $\frac{1}{1-\rho}\sum_{g=k+1+n_i}^{n+1} R_g$. We assumed the worst case in which the constituent correct nodes of C_j are slow, thus these nodes have counted on their timers at least $(1-\rho)\cdot\frac{1}{1-\rho}\sum_{g=k+1+n_i}^{n+1} R_g = \sum_{g=k+1+n_i}^{n+1} R_g$ time units since their last pulse. Hence, the correct nodes of C_j are at real-time $t_{c_i\_fires}$ at most in threshold level $k+n_i$. Should $k<f$ or the fire of C_i be received at a delay, then this may only cause the threshold level at time of assessment of the fire from C_i to be equal or even smaller than $k+n_i$. Thus, Lemma 2.3.4 and Property 1 guarantee that the first node in C_j will thus fire and that the rest of the nodes in both synchronized clusters will follow their respective first ones within σ real-time units.

Step 1b: We seek to estimate the maximum distance between the two synchronized clusters following the fire of C_j. The first node in C_j will fire at the latest upon receiving and assessing the message of the last node in C_i. More precisely, fire at the latest d real-time units following the fire of the last node in C_i, yielding a new $dist(C_i, C_j, ..)$ of at most $2d$ real-time units regardless of the previous $dist(C_i, C_j, ..)$, n_i, k and n_j. The last node of C_j is at most at a distance of d from the first node of C_j therefore making the maximal distance between the first node of C_i and the last node of C_j, at the moment it fires, equal $3d$ real-time units.

Step 2: We will complete the proof by showing that after C_i causes C_j to fire, the two synchronized clusters actually absorb. We need to show that in the cycle subsequent to Step 1, the nodes that constituted C_i and C_j become a synchronized set. Examine the case in which follow-

ing Step 1, either one of the two synchronized clusters increment its Counter to k' and fires at the beginning of threshold level k'. We will observe the ϕ of the first node to fire, denoted by $\phi_{\text{first\_node}-2nd-cycle}$. Following the same arguments as in Step 1, all other nodes increment their Counters to $k'+1$ after receiving this node's fire. Consider that this happens at the moment that this first node incremented its Counter to k' and fired, denoted $t_{2nd-cycle-fire}$. Below we compute, using Property 3, the lower bound on the ϕ of the rest of the nodes at real-time $t_{2nd-cycle-fire}$, denoted $\phi_{other-nodes}(t_{2nd-cycle-fire})$.

$$\begin{aligned}\phi_{other-nodes}(t_{2nd-cycle-fire}) &\geq \phi_{\text{first\_node}-2nd-cycle}(t_{2nd-cycle-fire}) - 3d \\ &= \frac{1}{1+\rho}\sum_{g=k'+1}^{n+1} R_g - 3d = \frac{1}{1+\rho}\sum_{g=k'+2}^{n+1} R_g + R_{k'+1} - 3d \\ &> \frac{1}{1+\rho}\sum_{g=k'+2}^{n+1} R_g + \frac{2\rho}{1-\rho^2}\sum_{g=1}^{n+1} R_g \ . \end{aligned} \tag{2.10}$$

In the worst case, the rest of the constituent nodes that were in C_i and C_j are slow nodes and thus, at real-time $t_{2nd-cycle-fire}$, counted:

$$\begin{aligned}(1-\rho)\cdot\left(\frac{1}{1+\rho}\sum_{g=k'+2}^{n+1} R_g + \frac{2\rho}{1-\rho^2}\sum_{g=1}^{n+1} R_g\right) &= \frac{1-\rho}{1+\rho}\sum_{g=k'+2}^{n+1} R_g + \frac{2\rho}{1+\rho}\sum_{g=1}^{n+1} R_g \\ &= \frac{1-\rho}{1+\rho}\sum_{g=k'+2}^{n+1} R_g + \frac{2\rho}{1+\rho}\sum_{g=k'+2}^{n+1} R_g + \frac{2\rho}{1+\rho}\sum_{g=1}^{k'+1} R_g \\ &= \sum_{g=k'+2}^{n+1} R_g + \frac{2\rho}{1+\rho}\sum_{g=1}^{k'+1} R_g > \sum_{g=k'+2}^{n+1} R_g \ . \end{aligned} \tag{2.11}$$

time units since their last pulse. Due to Property 3 all these correct nodes receive the fire and increment their Counters to $k'+1$ in a threshold level which is less or equal to $k'+1$ and will fire as well within d real-time units of the first node in the second cycle. □

Theorem 2.3.1 *(Conditions for Absorbance) Given two synchronized clusters, C_i preceding C_j, if:*

1. *C_i fires with Counter=k, at real-time $t_{c_i\_fires}$, where $0 \leq k \leq f$, and*
2. *$\exists\, t,\ t_{prev\_c_j\_fired} \leq t \leq t_{c_i\_fires}$, for which $dist(C_i, C_j, t) \leq ad(C_i)$*

then C_i will absorb C_j.

Proof: Denote $t_{prev\_c_j\_fired}$ the real-time at which C_j previously fired before time $t_{c_i\_fires}$. Given that at some time t, where $t_{prev\_c_j\_fired} \leq t \leq t_{c_i\_fires}$, $dist(C_i, C_j, t) \leq ad(C_i)$, we wish to calculate the maximal possible distance between the two synchronized clusters at real-time $t_{c_i\_fires}$, the time at which C_i fires with Counter=k, where $0 \leq k \leq f$.

Under the above assumptions, the maximal possible distance at real-time $t_{c_i\_fires}$ is obtained when $k = f$ and when at time $t_{prev\_c_j\_fired}$ the distance between C_i and C_j was exactly $ad(C_i)$, i.e $dist(C_i, C_j, t_{prev\_c_j\_fired}) = ad(C_i)$. The upper bound on $dist(C_i, C_j, t_{c_i\_fires})$ takes into account that from $C_i's$ previous real-time firing time, $t_{prev\_c_i\_fired}$, and until real-time $t_{c_i\_fires}$, the nodes of C_i were fast and that from real-time $t_{prev\_c_j\_fired}$ and until $t_{c_i\_fires}$, the nodes of C_j were slow. Thus the bound on $dist(C_i, C_j, t_{c_i\_fires})$ becomes the real-time difference between these:

$$
\begin{aligned}
&dist(C_i, C_j, t_{c_i\_fires}) = \phi_{c_i}(t_{c_i\_fires}) - \phi_{c_j}(t_{c_i\_fires}) = \\
&\frac{1}{1+\rho}\sum_{g=k+1}^{n+1} R_g - \frac{1}{1-\rho}\sum_{g=k+1+n_i}^{n+1} R_g = \frac{1}{1+\rho}\sum_{g=k+1}^{k+n_i} R_g + (\frac{1}{1+\rho} - \frac{1}{1-\rho})\sum_{g=k+1+n_i}^{n+1} R_g = \\
&(\frac{1}{1+\rho} - (\frac{1}{1+\rho} - \frac{1}{1-\rho}))\sum_{g=k+1}^{k+n_i} R_g + (\frac{1}{1+\rho} - \frac{1}{1-\rho})\sum_{g=k+1}^{n+1} R_g = \\
&\frac{1}{1-\rho}\sum_{g=k+1}^{k+n_i} R_g + (\frac{1}{1+\rho} - \frac{1}{1-\rho})\sum_{g=k+1}^{n+1} R_g = \\
&\frac{1}{1-\rho}\sum_{g=k+1}^{k+n_i} R_g - \frac{2\rho}{1-\rho^2}\sum_{g=k+1}^{n+1} R_g \ . \qquad (2.12)
\end{aligned}
$$

Eq. 2.12 is the upper bound on the distance between the two synchronized clusters at real-time $t_{c_i\_fires}$, thus following Lemma 2.3.7, the two synchronized clusters absorb. □

2.3.1.4 Convergence of the Synchronized Clusters

In the coming subsection we look at the correct nodes as partitioned into synchronized clusters (at some specific time). Observation 2.3.2 ensures that no two of these synchronized clusters comprise one synchronized set of nodes. The objective of Theorem 2.3.2 is to show that within finite time, at least two of these synchronized clusters will comprise one synchronized set of nodes. Specifically, we show that in any state that is not a synchronized_pulse_state of the system, there are at least two synchronized clusters whose unified set of nodes is not a synchronized set but that are within absorbance distance of each other, and consequently they absorb. Thus, eventually all synchronized clusters will comprise a synchronized set of nodes.

We claim that if the following relationship between REF and *Cycle* is satisfied, then absorbance (of two synchronized clusters whose unified set is not a synchronized set), is ensured irrespective of the states of the synchronized clusters. Let $C_{j'}$ denote the largest synchronized cluster. The theorem below, Theorem 2.3.2, shows that for a given clustering of $n - f$ correct nodes into $c > 1$ synchronized clusters and for n, f, *Cycle* and REF that satisfy

$$
\sum_{j=1, j\neq j'}^{c} ad(C_j) + \frac{1}{1-\rho}\sum_{g=1}^{n_{j'}} R_g \geq \frac{1}{1-\rho} \cdot \mathit{Cycle} \qquad (2.13)
$$

there exist at least two synchronized clusters, whose unified set is not a synchronized set of nodes, that will eventually undergo absorbance.

Note that Eq. 2.13 is derived from Property 7 (Eq. 2.2):
Eq. 2.2 derives the following equation (since the R_g values are non-negative),

$$\sum_{j=1,j\neq j'}^{c}\sum_{g=f+1}^{f+n_j} R_g + \frac{1}{1-\rho}\sum_{g=1}^{n_{j'}} R_g \geq \frac{1}{1-\rho}\cdot \mathit{Cycle}\ . \tag{2.14}$$

Incorporating the absorbance distance of Definition 2.3.1 into Eq. 2.14 yields exactly Eq. 2.13. We use Eq. 2.13 in Theorem 2.3.2 instead of Eq. 2.2 (Property 7) for readability of the proof.

Theorem 2.3.2 *(Absorbance) Assume a clustering of $n-f$ correct nodes into $c>1$ synchronized clusters at real-time t_0. Further assume that Eq. 2.13 holds for the resulting clustering. Then there will be at least one synchronized cluster that will absorb some other synchronized cluster by real-time $t_0 + 2\cdot$ cycle.*

Proof: Note that following the synchronized cluster procedure, the unified set of the two synchronized clusters that will be shown to absorb, are not necessarily a synchronized set of nodes at time t_0. Assume without loss of generality that $C_{j'}$ is the synchronized cluster with the largest number of nodes, consequent to running the clustering procedure. Exactly one out of the following two possibilities takes place at t_0:

1. $\exists i\ (1\leq i\leq c)$, such that $dist(C_i, C_{(i+1)(\mathrm{mod}\ c)}, t_0) \leq ad(C_i)$.
2. $\forall i\ (1\leq i\leq c, i\neq j'), dist(C_i, C_{(i+1)(\mathrm{mod}\ c)}, t_0) > ad(C_i)$.

Consider case 1. Following the protocol, C_i must fire within *Cycle* local-time units of t_0. Observe the first real-time, denoted t_i, at which C_i fires subsequent to real-time t_0. Assume that $k\geq 0$ is the number of distinct inputs that causes the Counter of at least one node in C_i to reach the threshold and fire (not counting the fire from nodes in C_i itself). If $k>f$ then at least one correct node outside of C_i caused some node in C_i to fire. This correct node must belong to some synchronized cluster which is not C_i. We denote this synchronized cluster C_x as its identity is irrelevant for the sake of the argument. We assumed that at least one node in C_i fired due to a node in C_x. Following Lemma 2.3.4 the rest of the nodes in C_i. will follow as well, as a synchronized cluster is in particular a synchronized set of nodes. This yields a new $dist(C_x, C_i, \ ..)$ of at most $3d$. Following the same arguments as in Step 2 of Lemma 2.3.7, C_x and C_i hence absorb. Therefore the objective is reached. Hence assume that $k\leq f$ and that C_i did not absorb with any preceding synchronized cluster. Thus, the last real-time that $C_{(i+1)(\mathrm{mod}\ c)}$ fired, denoted $t_{C_{i+1}-fired}$, was before or equal to real-time t_0, i.e. $t_{C_{i+1}-fired}\leq t_0\leq t_i$ and $dist(C_i, C_{(i+1)(\mathrm{mod}\ c)}, t_0)\leq ad(C_i)$. By Theorem 2.3.1, C_i will absorb $C_{(i+1)(\mathrm{mod}\ c)}$.

Consider case 2. We do not assume that $dist(C_{j'}, C_{(j'+1)(\mathrm{mod}\ c)}, t_0) > ad(C_{j'})$. Assume that there is no absorbance until $C_{j'}$ fires (otherwise the claim is proven). Let $t_{j'}$ denote the real-time at which the first node in $C_{j'}$ fires, at which $\phi_{c_{j'}}(t_{j'})=0$. There are two possibilities at $t_{j'}$:

2a. $\exists i(1\leq i\leq c)$, such that at $t_{j'}, dist(C_i, C_{(i+1)(\mathrm{mod}\ c)}, t_{j'})\leq ad(C_i)$.

2b. $\forall i(1\leq i\leq c, i\neq j'), dist(C_i, C_{(i+1)(\mathrm{mod}\ c)}, t_{j'}) > ad(C_i)$.

Consider case 2a. This case is equivalent to case 1. The last real-time that $C_{(i+1)(\mathrm{mod}\ c)}$ fired, denoted $t_{C_{i+1}-fired}$, was before or equal to real-time $t_{j'}$. Denote t_i the real-time at which the first node of C_i fires. Thus, $t_{C_{i+1}-fired} \leq t_{j'} \leq t_i$ and $dist(C_i, C_{(i+1)(\mathrm{mod}\ c)}, t_0) \leq ad(C_i)$. By Theorem 2.3.1, C_i will absorb $C_{(i+1)(\mathrm{mod}\ c)}$.

Consider case 2b. We wish to calculate $\phi_{c_{j'+1}}(t_{j'})$ and from this deduce the upper bound on the threshold level of the first node in $C_{(j'+1)(\mathrm{mod}\ c)}$ at real-time $t_{j'}$. We first want to point out that

$$\phi_{c_{j'+1}}(t_{j'}) > \sum_{j=1, j\neq j'}^{c} ad(C_j)\ . \tag{2.15}$$

This stems from the fact that $C_{j'}$ has just fired and that $C_{j'}$ and $C_{(j'+1)(\mathrm{mod}\ c)}$ are adjacent synchronized clusters which implies that

$$\forall i (1 \leq i \leq c, i \neq j'+1), \phi_{c_{j'+1}}(t_{j'}) > \phi_{c_i}(t_{j'}).$$

Recall that $\phi_{c_{j'}}(t_{j'}) = 0$. From the case considered in 2b we have that

$$\forall i (1 \leq i \leq c, i \neq j'),\ dist(C_i,\ C_{(i+1)\ (\mathrm{mod}\ c)}, t_{j'}) > ad(C_i).$$

Thus Eq. 2.15 follows. Following Eq. 2.13 and Eq. 2.15 we get:

$$\phi_{c_{j'+1}}(t_{j'}) > \sum_{j=1, j\neq j'}^{c} ad(C_j) \geq \frac{1}{1-\rho} \cdot \mathit{Cycle}\ - \frac{1}{1-\rho} \sum_{g=1}^{n_{j'}} R_g\ . \tag{2.16}$$

In the worst case the nodes of $C_{(j'+1)(\mathrm{mod}\ c)}$ are slow. Thus at real-time $t_{j'}$ they have measured, from their last pulse, at least $(1-\rho)\cdot\phi_{c_{j'+1}}(t_{j'}) = (1-\rho)\cdot[\frac{1}{1-\rho}\cdot \mathit{Cycle}\ - \frac{1}{1-\rho}\sum_{g=1}^{n_{j'}} R_g] = \sum_{g=n_{j'+1}}^{n+1} R_g$ local-time units. Thus, following Property 1, the first node in $C_{(j'+1)(\mathrm{mod}\ c)}$ receives the fire from $C_{j'}$ and increment its Counter to at least $n_{j'}$ in a threshold level which is less or equal to $n_{j'}$ and will thus fire as well. Following Lemma 2.3.4 the rest of the synchronized cluster will follow as well. This yields a new $dist(C_{j'}, C_{(j'+1)(\mathrm{mod}\ c)},\ ..)$ of at most $3d$. Following the same arguments as in Step 2 of Lemma 2.3.7, $C_{j'}$ and $C_{(j'+1)(\mathrm{mod}\ c)}$ hence absorb.

Thus at least two synchronized clusters will absorb within $2 \cdot \mathit{cycle}$ of t_0 which concludes the proof. □

The following theorem assumes the worst case of $n = 3f + 1$.

Theorem 2.3.3 *(Convergence) Within at most* $2(2f+1)\cdot$ *cycle real-time units the system reaches a synchronized_pulse_state.*

Proof: Assume that $n = 3f + 1$. Thus, the maximal number of synchronized clusters is $2f + 1$, and since following Theorem 2.3.2 at least two synchronized clusters absorb in every two cycles we obtain the bound. □

2.4 Analysis of the Algorithms

The protocol operates in two epochs: In the first epoch there is no limitations on the number of failures and faulty nodes. In this epoch the system might be in any state. In the second epoch there are at most f nodes that may behave arbitrarily at the same time, from which the protocol may start to converge. Nodes may fail and recover and nodes that have just recovered need time to synchronize. Therefore, we assume that eventually we have a window of time within which the turnover between faulty and non-faulty nodes is sufficiently low and within which the system inevitably converges (Theorem 2.3.2).

Authentication and fault ratio: The algorithm does not require the power of unforgeable signatures, only an equivalence to an authenticated channel is required. Note that the shared memory model [34] has an implicit assumption that is equivalent to an authenticated channel, since a node "knows" the identity of the node that wrote to the memory it reads from. A similar assumption is also implicit in many message passing models by assuming a direct link among neighbors, and as a result, a node "knows" the identity of the sender of a message it receives.

Many fundamental problems in distributed networks have been proven to require $3f + 1$ nodes to overcome f concurrent Byzantine faults in order to reach a deterministic solution without authentication [37, 57, 26, 25]. We have not shown this relationship to be a necessary requirement for solving the "Pulse Synchronization" problem but the results for related problems lead us to believe that a similar result should exist for the "Pulse Synchronization" problem.

There are algorithms that have no lower bound on the number of nodes required to handle f Byzantine faults, but unforgeable signatures are required as all the signatures in the message are validated by the receiver [26]. This is costly time-wise, it increases the message size, and it introduces other limitations, which our algorithm does not have. Moreover, within the self-stabilizing paradigm, using digital signatures to counter Byzantine nodes exposes the protocols to "replay-attack" which might empty its usefulness.

Convergence time: We have shown in Chapter 1 that self-stabilizing Byzantine clock synchronization and self-stabilizing Byzantine pulse synchronization are supposedly equally hard. The only self-stabilizing Byzantine clock synchronization algorithms besides the one in Chapter 4 are found in [34]. The randomized self-stabilizing Byzantine clock synchronization algorithm published there synchronizes in $M \cdot 2^{2(n-f)}$ steps, where M is the upper bound on the clock values held by individual processors. The algorithm uses message passing, it allows transient and permanent faults during convergence, requires at least $3f+1$ processors, but utilizes a global pulse system. An additional algorithm in [34], does not use a global pulse system and is thus partially synchronous similar to our model. The convergence time of the latter algorithm is $O((n-f)n^{6(n-f)})$. This is drastically higher than our result, which has a cycle length of $O(f^2) \cdot d$ time units and converges within $2(2f+1)$ cycles. The convergence time of the only other correct self-stabilizing Byzantine pulse synchronization algorithm (Chapter 7) has a cycle length of $O(f) \cdot d$ time units and converges within 6 cycles.

Message and space complexity: The size of each message is $O(logn)$ bits. Each correct node multicasts exactly one message per cycle. This yields a message complexity of at most n messages per cycle. The system's message complexity to reach synchronization from any arbitrary state is at most $2n(2f+1)$ messages per synchronization from any arbitrary initial state. The faulty nodes cannot cause the correct nodes to fire more messages during a cycle. Comparatively, the

self-stabilizing clock synchronization algorithm in [34] sends n messages during a pulse and thus has a message complexity of $O(n(n-f)n^{6(n-f)})$. This is significantly larger than our message complexity irrespective of the time interval between the pulses. The message complexity of the self-stabilizing Byzantine pulse synchronization in equals $O(n^3)$ per cycle.

The space complexity is $O(n)$ since the variables maintained by the processors keep only a linear number of messages recently received and various other small range variables. The number of possible states of a node is linear in n and the node does not need to keep a configuration table.

The message broadcast assumptions, in which every message, even from a faulty node, eventually arrives at all correct nodes, still leaves the faulty nodes with certain powers of multifaced behavior since we assume nothing on the order of arrival of the messages. Consecutive messages received from the same source within a short time window are ignored, thus, a faulty node can send two concomitant messages with differing values such that two correct nodes might receive and relate to different values from the same faulty node.

Tightness of synchronization: In the presented algorithm, the invocation of the pulses of the nodes will be synchronized to within the bound on the relay time of messages sent and received by correct nodes. In the broadcast version, this bound on the relay time equals d real-time units. Note that the lower bound on clock synchronization in completely connected, fault-free networks [56] is $d(1-1/n)$. We have shown in Section 2.2 how the algorithm can be executed in non-broadcast networks to achieve a synchronization tightness of $\sigma = 3d$. Comparatively, the clock synchronization algorithm of [26] reaches a synchronization tightness typical of clock synchronization algorithms of $d(1+\rho)+2\rho(1+\rho)\cdot R$, where R is the time between re-synchronizations. The second Byzantine clock synchronization algorithm in [34] reaches a synchronization tightness which is in the magnitude of $(n-f)\cdot d(1+\rho)$. This is significantly less tight than our result. The tightness of the self-stabilizing Byzantine pulse synchronization in Chapter 7 equals $3d$ real-time units.

Firing frequency bound: The firing frequency upper bound during normal steady-state behavior is around twice that of the endogenous firing frequency of the nodes. This is because $cycle_{min} \geq \frac{Cycle}{2}$. This bound is influenced by the fraction of faulty nodes (the sum of the first f threshold steps relative to *Cycle*). For $n = 3f+1$ this translates to $\approx \frac{1}{2}Cycle$. Thus, if required, the firing frequency bound can be closer to the endogenous firing frequency of $1 \cdot Cycle$ if the fraction of faulty nodes is assumed to be lower. For example, for a fraction of fault nodes of $f = \frac{n}{10}$, the lower bound on the cycle length, $cycle_{min}$, becomes approximately $8/9$ that of the endogenous cycle length. $cycle_{max} = Cycle \cdot (1+\rho)$ real-time units.

2.5 Discussion

We developed and presented the "Pulse Synchronization" problem in general, and an efficient linear-time self-stabilizing Byzantine pulse synchronization algorithm, BIO-PULSE-SYNCH, as a solution in particular. The pulse synchronization problem poses the nodes with the challenge of invoking regular events synchronously. The system may be in an arbitrary state in which there can be an unbounded number of Byzantine faults. The problem requires the pulses to eventually synchronize from any initial state once the bound on the permanent number of Byzantine failures is less than a third of the network. The problem resembles the clock synchronization problem though there is no "value" (e.g. clock time) to agree on, rather an event in time. Furthermore,

to the best of our knowledge, the only efficient self-stabilizing Byzantine clock synchronization algorithm assumes a background pulse synchronization module.

The algorithm developed is inspired by and shares properties with the lobster cardiac pacemaker network; the network elements (the neurons) fire in tight synchrony within each other, whereas the synchronized firing pace can vary, up to a certain extent, within a linear envelope of a completely regular firing pattern.

The neural network simulator SONN ([67]) was used in early stages of developing the algorithm for verification of the protocol in the face of probabilistic faults and random initial states. A natural next step should be to undergo simulation and mechanical verification of the current protocol that can mimic a true distributed system facing transient and Byzantine faults.

2.6 Proofs

Proof of correctness of the SUMMATION **procedure:**

Lemma 2.6.1 *For* $k \in \mathbb{N},\ k \geq 0$,

$$\tau(k) \cdot \frac{1+\rho}{1-\rho} + 2d(1+\rho) = \tau(k+1) \ .$$

Proof:

$$\tau(k) \cdot \frac{1+\rho}{1-\rho} + 2d(1+\rho) = [2d(1+\rho)\frac{(\frac{1+\rho}{1-\rho})^{k+1} - 1}{(\frac{1+\rho}{1-\rho}) - 1}] \cdot \frac{1+\rho}{1-\rho} + 2d(1+\rho)$$

$$= [2d(1+\rho)\sum_{i=0}^{k}(\frac{1+\rho}{1-\rho})^i] \cdot \frac{1+\rho}{1-\rho} + 2d(1+\rho) = [2d(1+\rho)\sum_{i=1}^{k+1}(\frac{1+\rho}{1-\rho})^i] + 2d(1+\rho)$$

$$= 2d(1+\rho)\sum_{i=0}^{k+1}(\frac{1+\rho}{1-\rho})^i = 2d(1+\rho)\frac{(\frac{1+\rho}{1-\rho})^{k+2} - 1}{(\frac{1+\rho}{1-\rho}) - 1} = \tau(k+1) \ .$$

□

Lemma 2.6.2 *Let a correct node* q *receive a message* M_p *from a correct node* p *at local-time* t_{arr}. *For every one of* p*'s stored messages* (S_r, t') *that is accounted for in* $Counter_{M_p}$, *then at* q, *from some time* t *in the local-time interval* $[t_{arr}, t_{arr} + d(1+\rho)]$ *and at least until the end of the interval:*

$$MessageAge(t, q, r) \leq \tau(Counter_{M_p} + 1) \ .$$

Proof: Following the PRUNE procedure at p, the oldest of its stored messages accounted for in $Counter_{M_p}$ was at most $\tau(Counter_{M_p})$ time units old on p's clock at the time it sent M_p. This oldest stored message could have arrived at q, $\delta(1+\rho)$ local-time units on q's clock, prior to its arrival at p. Within this time p should also have received all the messages accounted for in M_p. Another $\pi(1+\rho)$ local-time units could then have passed on q's clock until M_p was sent. M_p could

have arrived at q, $\delta(1+\rho)$ time units on q's clock after it was sent by p. By this time q would also have received all the messages that are accounted for in M_p, irrespective if q had previous messages from the same nodes. Another $\pi(1+\rho)$ time units can then pass on q's clock until all messages are processed. Thus, in the worst case that node p is slow and node q is fast and by Lemma 2.6.1, for every stored message accounted for in $Counter_{M_p}$, $\exists t \in [t_{arr} + d(1+\rho)]$, we have:

$$\begin{aligned} MessageAge(t,q,r) &\leq MessageAge(t_{arr} + d(1+\rho), q, r) \\ &\leq \tau(Counter_{M_p}) \cdot \frac{1+\rho}{1-\rho} + \delta(1+\rho) + \pi(1+\rho) + \delta(1+\rho) + \pi(1+\rho) \\ &= \tau(Counter_{M_p}) \cdot \frac{1+\rho}{1-\rho} + 2d(1+\rho) = \tau(Counter_{M_p} + 1) \ . \end{aligned}$$

□

Lemma 2.6.3 *The Counter of a correct node cannot exceed n and a correct node will not send a Counter that exceeds $n-1$.*

Proof: There can be at most n distinct stored messages in the CS of a correct node hereby bounding the Counter by n.

For a correct node to have a Counter that equals exactly n it needs its own stored message to be in its CS, as a consequence of a message it sent. Consider the moment after it sent this message, say before the node's Counter reached n, that is accounted for in its CS. This message was concomitant to its pulse invocation and cycle reset. The node assesses its own message at most $d(1+\rho)$ local-time units after sending it thus, following the PRUNE procedure, its own stored message will decay at most $\tau(n+2) + d(1+\rho) < \tau(n+3)) = R_{n+1}$ local-time units after it was sent. Thus at the moment the node reaches threshold level R_n its own message will already have decayed and the Counter will decrease and will be at most $n-1$, implying that any message sent by the node can carry a Counter of at most $n-1$. □

Lemma 2.6.4 *A stored message, (S_r, t'), that has been moved to the RUCS of a correct node q up to $d(1+\rho)$ local-time units subsequent to the event of sending a message M_p by p, (or was moved at an earlier time) cannot have been accounted for in $Counter_{M_p}$.*

Proof: Assume that the stored message (S_r, t') was moved to the RUCS of node q at a local-time t, $d(1+\rho)$ local-time units subsequent to the event $t_{send\,M_p}$ at node p, (or it was moved at an earlier time). Thus at q at local-time t, $MessageAge(t,q,r) > \tau(n+1)$. Therefore at node p at local-time $t_{send\,M_p}$, $MessageAge(t_{send\,M_p}, p, r) > \tau(n+1) - 2d(1+\rho) > \tau(n)$. This is because p could have received the message M_r up to $d(1+\rho)$ local-time units later than q did, and q could have received M_p up to $d(1+\rho)$ local-time units after it was sent.

Following the PRUNE procedure at p, $(S_r, t")$ would have been accounted for at the sending time of M_p only if $Counter_{M_p} \geq n+1$. Therefore by Lemma 2.6.3 node p did not account for the stored message of r in $Counter_{M_p}$. □

Corollary 2.6.5 *A stored message, (S_r, t'), that has decayed at a correct node q prior to the event of sending a message M_p by p, cannot have been accounted for in $Counter_{M_p}$.*

Proof: Corollary 2.6.5 is an immediate corollary of Lemma 2.6.4. □

Corollary 2.6.6 *Let a correct node q receive a message M_p from a correct node p at local-time t_{arr}. Then, at q, from some time t in the local-time interval $[t_{arr}, t_{arr} + d(1+\rho)]$ and at least until the end of the interval:*

$$\|\textit{Message\_Pool}\| \geq Counter_{M_p} + 1 \ .$$

Proof: Corollary 2.6.6 is an immediate corollary of Lemma 2.6.2 and Lemma 2.6.4. □

Thus, as a consequence to the lemmata, we can say informally, that when the system is coherent all correct nodes relate to the same set of messages sent and received.

Proof of Theorem 2.2.1

Recall the statement of Theorem 2.2.1:

Any message, M_p, sent by a correct node p will be assessed as timely by every correct node q.

Proof: Let M_p be sent by a correct node p, and received by a correct node q at local-time $t_{arr.}$ We show that the timeliness conditions hold:
Timeliness Condition 1: $0 \leq Counter_{M_p} \leq n-1$ as implied by Lemma 2.6.3 and by the fact that the CS cannot hold a negative number of stored messages.

Timeliness Condition 2: Following Lemma 2.6.3 a correct node will not fire during the absolute refractory period. Property 5 therefore implies that a correct node cannot count less than $\tau(n+3)$ local-time units between its consecutive firings. A previous message from a correct node will therefore be at least $\tau(n+2)$ local-time units old at any other correct node before it will receive an additional message from that same node. Following the PRUNE procedure, the former message will therefore have decayed at all correct nodes and therefore cannot be present in the *Message_Pool* at the arrival time of the subsequent message from the same sender.

Timeliness Condition 3: This timeliness condition validates $Counter_{M_p}$. The validation criterion relies on the relation imposed at the sending node by the PRUNE procedure, between the $MessageAge(t, p, ..)$ of its accounted stored messages and its current Counter.

By Lemma 2.6.2, for all stored messages (S_r, t') accounted for in M_p,
$MessageAge(t, q, r) \leq \tau(Counter_{M_p} + 1)$ from some local-time $t \in [t_{arr}, t_{arr} + d(1+\rho)]$ and until the end of the interval.

By Corollary 2.6.6, $\|\textit{Message\_Pool}\| \geq Counter_{M_p}+1$, from some local-time $t'' \in [t_{arr}, t_{arr}+ d(1+\rho)]$ and until the end of the interval.

We therefore proved that Timeliness Condition 3 holds for any $0 \leq k < n$ at the latest at local-time $t_{arr} + d(1+\rho)$.

The message M_p is therefore assessed as timely by q. □

Lemma 2.6.7 *Following the arrival and assessment of a timely message M_p at node q, the subsequent execution of the* MAKE-ACCOUNTABLE *procedure yields $Counter_q > Counter_{M_p}$.*

Proof: We first show that at time t, the time of execution of the MAKE-ACCOUNTABLE procedure, $\max[1,\ (Counter_{M_p} - Counter_q + 1)] \leq \|\text{UCS}\|$, ensuring the existence of a sufficient number of stored messages in UCS to be moved to CS.

M_p is assessed as timely at q, therefore, by Timeliness Condition 3 and Lemma 2.6.4, at time t,

$$\begin{aligned} Counter_{M_p} \ &< \ \|\textit{Message\_Pool}\| = \|\text{CS}\| + \|\text{UCS}\| = Counter_q + \|\text{UCS}\| = \\ &\quad Counter_{M_p} - \max[1,\ (Counter_{M_p} - Counter_q + 1)] + 1 + \|\text{UCS}\| \\ &\Rightarrow \ 0 < -\max[1,\ (Counter_{M_p} - Counter_q + 1)] + 1 + \|\text{UCS}\| \\ &\Rightarrow \ \max[1,\ (Counter_{M_p} - Counter_q + 1)] - 1 < \|\text{UCS}\| \\ &\Rightarrow \ \max[1,\ (Counter_{M_p} - Counter_q + 1)] \leq \|\text{UCS}\| \ . \end{aligned}$$

There are two possibilities at the instant **prior** to the execution of the MAKE-ACCOUNTABLE procedure. At this instant $Counter_q = \|\text{CS}\|$:

1. $Counter_{M_p} \leq Counter_q$, then $\max[1,\ (Counter_{M_p} - Counter_q + 1)] = 1$, meaning $\|\text{CS}\|$ will increase by 1.

2. $Counter_{M_p} > Counter_q$, then $\|\text{CS}\|$ will be $Counter_q + \max[1,\ (Counter_{M_p} - Counter_q + 1)] = Counter_q + Counter_{M_p} - Counter_q + 1 = Counter_{M_p} + 1$.

In either case, immediately subsequent to the execution of the procedure we get: $\|\text{CS}\| > Counter_{M_p}$ and therefore the updated $Counter_q > Counter_{M_p}$. □

Proof of Lemma 2.2.1

Recall the statement of Lemma 2.2.1:

Following the arrival of a timely message M_p, at a node q, then at time $t_{send\,M_q}$, $Counter_q > Counter_{M_p}$.

Proof: Let t_{arr} denote the local-time of arrival of M_p at q. Recall that $t_{send\,M_q}$ is the local-time at which q is ready to assess whether to send a message consequent to the arrival and processing of M_p. In the local-time interval $[t_{arr}, t_{send\,M_q}]$ at least one PRUNE procedure is executed at q, the one which is triggered by the arrival of M_p. Following Lemma 2.6.7, $Counter_q > Counter_{M_p}$ subsequent to the execution of the MAKE-ACCOUNTABLE procedure. Note that $t_{arr} \leq t_{send\ M_q} \leq t_{arr} + d(1+\rho)$. By Lemma 2.6.4 all stored messages accounted for in $Counter_{M_p}$ will not be moved out of the *Message_Pool* by any PRUNE procedure executed up to local-time $t_{send\,M_q}$, thus, $Counter_q$ must stay with a value greater than $Counter_{M_p}$ up to time $t_{send\,M_q}$. □

Lemma 2.6.8

Lemma 2.6.8 *Let $p, q \in C_i$ and $r \in C_j$, denote three correct nodes belonging to two different synchronized clusters. Following the arrival and assessment of p's and q's fires, both will be accounted for in the Counter of r.*

Proof: Without loss of generality, assume that p fires before node q. Following Lemma 2.3.4 node q will fire within σ of p ($d(1+\rho)$ on r's clock). Node r will receive and assess q's fire at a time t_q at most $d(1+\rho)+d(1+\rho) = 2d(1+\rho)$ after p fired. Summation Property [P2] ensures that r will account for each one after their arrival and assessments. Furthermore, $MessageAge(t_q, q, p) \leq 2d(1+\rho) = \tau(0)$ and therefore node r did not decay or move M_p to RUCS by time t_q. Therefore, M_p is still accounted for by node r at time t_q and thus, both p and q are accounted for in $Counter_r$ at time t_q. □

Chapter 3

Stabilization of General Byzantine Algorithms using Pulse Synchronization

3.1 Specific Definitions

DEFINITION 3.1.1 *A node is* **correct** *following* $\Delta_{node} = pulse\_conv + 2 \cdot cycle + \sigma$ *real-time of continuous non-faulty behavior.*

DEFINITION 3.1.2 *The communication network is* **correct** *following* $\Delta_{net} = pulse\_conv + 2 \cdot cycle + \sigma$ *real-time of continuous non-faulty behavior.*

The self-stabilization paradigm assumes that all variables and program registers are volatile and thus prone to corruption or can initialize with arbitrary assignments. Conversely, it assumes that the code (the instructed protocol) is not dynamic and can thus be stored on non-volatile or non-corruptible storage. Furthermore, it is assumed in the paradigm that any access to an external module utilized by the system is eventually restored. E.g., any dependency on continuous time correlated to real-time without access to an external time source, can not be handled in the context of self-stabilization as no algorithm can restore the reference to external time without access to the external time source.

A *local state* of a node is comprised of the program counter and an assignment of values to the local variables. A node switches from one local state to another through a computation step. A *global state* of a system of nodes is the set of local states of its constituents nodes and the contents of the FIFO communication channels. A *local application state* is a subset of the variables of the local state that are relevant for the application. Two local states are said to be *distinct* if they represent local states on different nodes. A *global application state* is a collection of all the distinct constituent local application states at a certain moment. A *global application snapshot* is any collection of distinct local application states. An *execution* of a program P is a possibly infinite sequence of global states in which each element follows from its predecessor by the execution of a single computation step of P. We define E to be the set of all possible execution sequences of a program P.

DEFINITION 3.1.3 *An initial state is said to be* ***normal*** *if the program counter of each correct node is 0 and the communication channels are empty.*

DEFINITION 3.1.4 *A **normal execution** is an execution whose initial state is normal and has entirely occurred while the system is coherent.*

DEFINITION 3.1.5 *A global application state is said to be **legal** if it could occur in a normal execution.*

DEFINITION 3.1.6 *A **legal execution** is an execution that is a non-empty suffix of a normal execution.*

We define NE, $(NE \subset E)$, to be the set of normal executions of P (also denoted $NE(P)$). Equivalently, we define LE, $(LE \subset E)$, to be the set of legal executions of P (denoted $LE(P)$ respectively). The legal global states and the set of legal executions are determined by the particular task in the specific system and its respective normal executions. This cannot be characterized in general terms regardless of the actual problem definition that program P seeks to solve.

The self-stabilization of a system is informally defined by the requirement that every execution in E has a non-empty suffix in LE. We adopt the definitions of a self-stabilizing extension of a non-stabilizing program from [47]:

DEFINITION 3.1.7 *A **projection** of a global state onto a subset of the variables and the messages on the channels is the value of the state for those variables and messages.*

DEFINITION 3.1.8 *Program Q is an **extension** of program P if for each global state in $NE(Q)$ there is a projection onto all variables and messages of P such that the resulting set of sequences is identical to $NE(P)$, up to stuttering*[1].

Note that when one considers only those portions of Q's global state that correspond to P's variables and messages and if repetitions of states are ignored, then the legal executions of P and Q are identical. Thus, a state of Q is a legal state of Q iff the projection onto P is a legal state of P. The program P to be extended is called **the basic program**.

DEFINITION 3.1.9 *Program Q is a **self-stabilizing extension** of a program P if Q is an extension of P and any execution in E(Q) has a non-empty suffix whose projection onto P is in LE(P).*

Thus, informally, if Q is a self-stabilizing extension of P then the projection of Q onto P is self-stabilizing. Therefore we refer to Q as a **stabilizer** of P.

3.2 A Byzantine Stabilizer

Intuitively, the task of stabilizing a program should supposedly be rather straightforward: Every period of time, make all nodes report their internal states, then sift through the collected states and search for a possibly global inconsistency in the algorithm as emerges from the global snapshot. Upon such an inconsistency make all nodes reset to a consistent state. Below we display a conceptual view of the scheme:

[1] When comparing sequences, adjacent identical states are eliminated; this is called the elimination of stuttering in [47].

At "$time - to - exchange - states$" do
1. Send local state to all nodes and Byzantine Agree on every node's state;
2. All correct nodes now see the same global snapshot;
3. Check if global snapshot represents a legal state;
4. If not then reset the basic program;
5. If yes but your state is corrupt then repair state;

This greatly simplified scheme does not address the many subtle problems that surface when facing transient faults and permanent Byzantine faults: How do you synchronize the point in time for reporting the internal states? How do you ensure that the global snapshot is concurrent enough to be meaningful? How do you prevent Byzantine nodes from causing correct nodes to see differing global snapshots? How does the predicate detection mask Byzantine values?

We address the synchronization issue by employing an underlying Byzantine self-stabilizing pulse synchronization procedure. The pulse is essentially used as the event that helps to determine when to report the local state. The "meaningfulness" of the global snapshot is addressed by the observation that many algorithms have identifiable events in their executions. In a semi-synchronous protocol different nodes should execute the same events within a small bounded time of each other. If all correct nodes report their local states and clock time[2] at such an event (denoted *sampling point*) then the combination of clock time and the emergent global snapshot can be used for deducing whether the protocol is in a legal state. As an example, consider that the events are the beginning of a round, in case the basic program works in rounds. Thus all correct nodes should, whenever the system is in a legal state, reach the event of a specific round within bounded clock time of each other. By instructing the nodes to report their state (round number) and clock time at the specific round, it can be deduced whether this event indeed happened within the legal bounded time. If so, then that implies that the global snapshot taken carries meaningful information about the global state of the system. By evaluating this global predicate a decision can be made as of the legality of the global state and a reset can be done, if required. If the reported clock times are "too far" from each other then this is a sufficient indication that the system is not in a legal state and thus should be reset.

The issue of Byzantine nodes and values are tackled by initiating Byzantine agreement on the reported states. This ensures that all correct nodes have identical views of the global snapshot.

Our scheme stabilizes any Byzantine protocol that has such events (sampling points) during the execution, which can be identified by checking the program counter and local state. Otherwise, it is required that the basic program signals when to read and report the local state. We argue that this definition covers an extensive set of protocols. Programs that work in round structure is just a specific and easily identifiable example of such protocols. We assume for simplicity that the sampling points are taken at least 4σ apart on the same node in order to be able to differentiate between adjacent sampling points due to the synchronization uncertainties. It remains open whether this bound is really required. In Section 3.3 we give a detailed example of how to extend a specific clock synchronization algorithm that does not operate in a round structure.

Note that we do not aim at achieving a consistent global snapshot in the Chandy-Lamport sense (see [17]), which is not clearly defined in the Byzantine fault model. For our purposes a projection

[2]Note that the clock time can be the elapsed time on a node's timer since the pulse. The synchronization of the pulses implies synchronization of these clocks.

of the local state to the application state suffices in order to detect states that violate the assumptions of the basic program on its initial states, which rendered it non-stabilizing in the first place.

Generally, the extension of the basic program is established through a user-supplied wrapper function, so called because it "wraps" the basic program and functions as an interface between the basic program and the stabilizer. Note that the wrapper procedures must be supplied by the implementor. This is because it is a semantic matter to determine whether the global application state predicate indicates an illegal state that violates the assumptions of the basic program. For the sake of modularity and readability the wrapper is divided into two distinct modules according to its two main functions. The GETSTATE_WRAPPER() module interprets the local state of the basic program and returns the local state at the sampling points. The EVALSTATE_WRAPPER() module evaluates the agreed global application snapshot and determines whether it is legal with respect to the application. It also instructs a node how to repair its local application state as a function of the global application snapshot, should a node detect that its local application state is inconsistent with the legal global application snapshot.

Restrictions on the basic program:

R1: The basic program at all correct nodes can be initialize within at least σ real-time units apart. The procedure INIT_BASIC_PROGRAM initializes it.

R2: The basic program can tolerate that up to f of the nodes can choose to keep values from previous incarnations of the basic program (e.g. for replay of digital signatures).

R3: Has repeated *sampling points* during execution that can be identified through the local state. The sampling points are such that if all correct nodes report their state at the same corresponding sampling point then the global application snapshot is "meaningful" with respect to the application.

R4: During a legal execution all the correct nodes' sampling points are within Δ real-time units of each other. The background pulse algorithm implies that $\Delta \geq \sigma$, because the pulse skew may cause the nodes to reach the sampling points up to σ real-time units of each other.

R5: There exists a value Σ, such that in every time-window that is at least some Σ real-time units long every correct node has at least one sampling point. This value also covers the initialization period of the basic program.

R6: The set of legal application states of the basic program can be determined by evaluating a predicate on the application state variables. An additional requirement is that if up to f non-faulty nodes detect that their own local state is inconsistent with a legal global application snapshot then it can be repaired without needing a global reset[3].

R7: The basic program has a closure property with regards to the legal global states. I.e. if the system is in a legal state and the system is coherent then it stays in a legal state as long as the system stays coherent.

To formalize the intuition we give a more refined presentation of the algorithm:

[3] A basic program that lacks this property might not converge to a legal state.

At "*pulse*" event Do */* received the internal pulse event */*
1. Revoke possible other instances of the algorithm and clear the data structures;
2. If (*reset*) then Do invoke INIT_BASIC_PROGRAM; */* reset the Basic Program */*

/ Lines 3,4 are executed by the* GETSTATE_WRAPPER() *procedure */*
3. Upon a sampling point Do
4. Set $Timer$:= elapsed time since pulse;
5. Record app_state & invoke BYZ_AGREEMENT on (app_state, $Timer$);

/ Line 6 is executed about agreement_duration time after the* $f{+}1^{st}$ *agreement */*
6. Sift through agreed values for a cluster of $\geq n - f$ values whose $Timer$s within 2Δ of each other, thus comprising a meaningful global application snapshot;
7. If no such cluster exists then Do *reset* := *true*;

/ Lines 8,9,10 are executed by the* EVALSTATE_WRAPPER() *procedure */*
8. Else Do predicate evaluation on the global application snapshot;
9. If global application snapshot is not legal Do *reset*:=*true*;
10. Else If you are not part of the cluster Do Repair your application state;

The complete algorithm, denoted BYZSTABILIZER, is given below in Figure **??**.

The internal pulse event is delivered by the pulse synchronization procedure (presented in Chapter 2 or Chapter 7). The synchronization of the pulses ensures that the BYZSTABILIZER procedure is invoked within σ real-time units of its invocation at all other correct nodes. Note that we do not assume any correlation between the pulse cycle and any internal cycles or rounds of the basic program. Hence at the time of the pulse, the basic program may be in any of its states. The Byzantine agreement procedure used, BYZ_AGREEMENT, is presented in Section 3.5. It is essentially the agreement equivalent of the consensus procedure of Section 4.4 in Chapter 4.

Line 1: Following the pulse any possible on-going invocation of BYZSTABILIZER (and thus any on-going BYZ_AGREEMENT or instance of the wrappers, but not the execution of the basic program) is revoked and all data structures that are not used by the basic program are cleared. The exception is the "reset" variable that is not cleared. Note that the application state, as it belongs to the basic program, remains intact.

Line 2-3: Each node p initializes a $Timer$ that holds the elapsed clock time since the last pulse invocation, before possibly doing a reset of the basic program.

Lines 4-7: When the GETSTATE_WRAPPER() wrapper procedure encounters a sampling point subsequent to the pulse, at elapsed time = $Timer$, then it records the local application state into the $RecState$ variable. Agreement is then invoked on (p, $RecState$, $Timer$). The procedure GETSTATE_WRAPPER() sanity checks the state recorded at line 6, thus if it detects that the local application state is invalid or corrupt it will return $\perp$.

Lines 8-15: Target at identifying the $f + 1^{st}$ (time-wise) distinct node whose value has been agreed upon, denoted the *pivot* node. Note that after a bounded time all correct nodes will identify the same pivot node. The time appearing in the agreed value of the pivot node is denoted T_{pivot}. The variable AS holds the set of agreed states. The variable *Agr_nodes* holds the set of nodes

Algorithm 1 BYZSTABILIZER */* executed at node q */*

At "*pulse*" event Do */* received the internal pulse event */*

Begin

1. Revoke possible other instances of BYZSTABILIZER and clear the data structures;
2. $Timer := 0$; $T_{pivot} := 0$;
3. If (*reset*) then Do invoke INIT_BASIC_PROGRAM; */* reset the Basic Program */*
4. Wait until $Timer = \sigma \cdot (1 + \rho)$ time units;

/ $read\&agree$ state at sampl. point; collect f+1 agreed states in window */*

5. Do
6. Invoke in the background $RecState :=$ GETSTATE_WRAPPER();
7. If $RecState \neq \perp$ then Do invoke BYZ_AGREEMENT(q, $RecState$, $Timer$);
8. $AS := \{(p, S, T) \mid$ BYZ_AGREEMENT returned $S \neq \perp\}$; */* add agreed state */*
9. *Agr_nodes* $:= \{p_i \mid (p_i, \_, T_i) \in AS,\ \sigma + \Delta \leq T_i \leq \Sigma + \Delta\}$; */* minimal T_i */*
10. Until ($\parallel$ *Agr_nodes* $\parallel \geq f + 1$ or $Timer > \Sigma + \Delta +$ *agreement_duration*);

/ collect agreed states, until no more possible states from correct nodes */*

11. Do
12. $AS := \{(p, S, T) \mid$ BYZ_AGREEMENT returned $S \neq \perp\}$; */* add agreed state */*
13. *Agr_nodes* $:= \{p_i \mid (p_i, \_, T_i) \in AS,\ \sigma + \Delta \leq T_i \leq \Sigma + \Delta\}$; */* minimal T_i */*
14. Let $pivot$ be the $f+1^{st}$ node in *Agr_nodes*, in ascending order by their min. T_i;
15. Until $Timer \geq T_{pivot} + (\sigma + \Delta +$ *agreement_duration*$) \cdot (1 + \rho)$ time units;

/ seek cluster of $\geq n-f$ values whose Timers within 2Δ of each other */*

16. $AS' := \{(p, S, T) \in AS \mid \sigma + \Delta \leq T \leq T_{pivot} + \Delta \cdot (1 + \rho)\}$;
17. *Cluster_rep* $:= \{(p_c, S_c, T_c) \in AS' \mid$
$\parallel \{p' \mid (p', S', T') \in AS\ \&\ T_c \leq T' \leq T_c + 2\Delta\ \&\ S_c \sim S'\} \parallel \geq n - f\}$;

/ if no cluster do reset, otherwise evaluate snapshot of earliest cluster */*

18. If $\parallel$ *Cluster_rep* $\parallel\ = 0$ then Do *reset* := true; */* if no n-f sized cluster found */*
19. Else Do (p_c, S_c, T_c):=$\min_T\{(p, S, T) \in$ *Cluster_rep*$\}$; */* else seek earliest cluster */*
20. $globAppSnapshot := \{(p', S', T') \in AS \mid T_c \leq T' \leq T_c + 2\Delta\ \&\ S_c \sim S'\}$;
21. *reset* := EVALSTATE_WRAPPER($globAppSnapshot$); */*reset,repair or nothing*/*

End

whose values have been agreed on.

Lines 16-17: A bounded period of time subsequent to T_{pivot}, all correct nodes must have terminated agreement on all nodes' values. It is then, that a cluster of at least $n - f$ agreed values is searched for, such that their $Timer$s are within 2Δ of each other.

Line 18: Such a cluster, if exists, comprises a meaningful global application snapshot. Otherwise, the global application state must be in an illegal state.

Lines 19-21: If a cluster is detected, then the EVALSTATE_WRAPPER procedure evaluates the global application snapshot. It determines whether the node must repair its local application state; whether a global reset should be scheduled at the next pulse invocation or whether the global

application state is assumed to be legal and thus nothing is done. The $\sim$ notation denotes equality between cluster identifiers.

Lemma 3.2.1 *If the system is in an arbitrary global state then, within finite time, subsequent to line* 17 *of the* BYZSTABILIZER *algorithm there is agreement on the set Cluster_rep.*

Proof: Since the system is coherent, all correct nodes invoked the last pulse synchronously. Following the protocol, every correct node reads and then initiates agreement on its application state upon reaching its first occurring sampling point after the pulse. There are two cases to consider: if the agreement is initiated by a correct node, then this happens at least σ real-time units subsequent to it's pulse. Thus all correct nodes will receive this node's initialization message as all correct nodes will already have invoked their pulse and will thus participate immediately in the agreement. In the other case consider that the agreement is initiated by a faulty node. Assume that a correct node receives the initialization message from the faulty node immediately following its pulse and that the other correct nodes receive it before their pulse and thus clear the message buffers at their pulse. This scenario can only happen if the faulty message is received before $\sigma(1+\rho)$ time units have passed on the correct receiver's timer. Following the protocol in Section 3.5, the receiving node waits until the end of the $\sigma \cdot (1+\rho)$ interval following its pulse, before processing the initialization message. By the time it will send an echo message, following the broadcast protocol (presented in Section 4.4), all correct nodes will have invoked their pulses and will receive every correct node's echo messages. This reduces to the case in which the faulty node sent its initialization message to a subset of the correct nodes, a situation which is handled by-design by the reliable broadcast protocol. Thus all the conditions of reliable broadcast are satisfied. Thus, eventually agreement is reached on every node that initiates agreement on its application state.

The agreement on any node's application state initiated with a certain T_i will terminate at all correct nodes within $T_i + (\mathit{agreement\_duration} + \sigma) \cdot (1+\rho)$ local-time units of their pulse. Thus, subsequent to measuring $T_i + (\mathit{agreement\_duration} + \sigma) \cdot (1+\rho)$ local-time units after their pulse, no correct node will decide on a message with $T_j < T_i$. All correct nodes initialize agreement on their application state within Σ of their pulse or within $(\sigma+\Sigma)\cdot(1+\rho)$ local-time units of the pulse of other correct nodes. Thus, within $(\sigma + \Delta + \Sigma + \mathit{agreement\_duration}) \cdot (1+\rho)$ local-time units of their pulse, no new agreements can terminate for $T_j < \Delta + \Sigma$. Thus, all correct nodes agree on the value T_{pivot}. Among the $f+1$ first terminated agreements there must be at least one correct node, and all correct nodes have initialized agreement on their application state within $\sigma + \Delta$ of T_{pivot}. Thus, subsequent to measuring $T_{pivot} + (\sigma + \Delta + \mathit{agreement\_duration}) \cdot (1+\rho)$ local-time units after the pulse, no correct node will decide on a message with $T_j < \Delta + \Sigma$ and there will be agreement on the application state of every node and hence there will be agreement on the set AS, the global application state whose recording times are less or equal to $\Delta + \sigma$. Hence, any action derived from the set AS, such as in Lines 16 and 17, will have the exact same outcome at all correct nodes. Thus, subsequent to line 17 of the BYZSTABILIZER algorithm there is agreement on the set *Cluster_rep*, which thus completes the proof.

Note that this is irrespective whether a subset of the correct nodes did a reset of the basic program subsequent to the pulse and irrespective of the legality of the state of the basic program. Note that not all correct nodes might necessarily be represented with a valid application state in the agreed set AS. E.g. as a result of the case in which the basic program is in an illegal state such that the sampling points of different nodes are far from each other. Another case occurs if the pulses are

invoked in between correct nodes sampling points such that some correct nodes initiate agreement on their application state and some don't. This case also occurs only if the basic program is in an illegal state so that the sampling points are far from each other.

□

Theorem 3.2.1 BYZSTABILIZER *is a self-stabilizing extension of any algorithm that complies with restrictions R1-R7.*

Proof:
Convergence: Let the system be coherent but in an arbitrary global state, s, with the nodes holding arbitrary local application states. The pulse synchronization procedure is self-stabilizing, thus, independent of the system's initial state within a finite time the pulses are invoked regularly and synchronously with a tightness of σ real-time units. At the pulse invocation all remnants of previously invoked BYZSTABILIZER, inclusive of its sub-procedures such as the agreement and wrappers, are flushed by all the correct nodes. Following Lemma 3.2.1, subsequent to line 17 of BYZSTABILIZER there is consensus on the selected cluster (including of the empty cluster). At line 18 there may be one of two possibilities:

1. $\parallel$ *Cluster_rep* $\parallel = 0$: This necessarily implies the basic program is in an illegal state. In this case all correct nodes will do $reset :=$*true*. At the next pulse all correct nodes will reset the basic program and thus converge to a legal state.

2. *A cluster was detected*: In this case subsequent to line 20 the variable globAppSnapshot, which holds the cluster whose states are the earliest agreed on since the pulse, will be generated at all correct nodes. Again, there are two cases to consider:

   (a) *The sampling points are within Δ real-time of each other*:
   Thus all correct nodes have initiated an agreement on their state within Δ real-time units of time T_{pivot} at the pivot node. Hence all correct nodes are represented in the cluster. The reset variable will be set at line 21 by the EVALSTATE_WRAPPER predicate detection procedure. If the procedure returns that the globAppSnapshot is legal then all correct nodes do nothing. Otherwise all correct nodes will reset the basic program at the next pulse and thus the system converges to a legal global state.

   (b) *The sampling points are not within Δ real-time of each other*: There are two cases to consider:

      i. *All correct nodes are represented in the cluster*:
      Thus the basic program is unsynchronized within the uncertainty window. If the EVALSTATE_WRAPPER procedure detects the illegality of the global state then all correct nodes will reset at next pulse, otherwise the illegality will not be detected and all correct nodes will not reset the basic program at the next pulse.

      ii. *At least one correct nodes is not represented in the cluster*: Again there are two cases:

         A. *The* EVALSTATE_WRAPPER *procedure evaluates in line* 21 *the application snapshot as illegal*: Then all correct nodes reset at the next pulse and the system attains a legal global state.

B. *The* EVALSTATE_WRAPPER *procedure evaluates in line* 21 *the application snapshot as legal*: This is due to faulty nodes that "fill-in" for the lacking correct values, then these correct nodes that are not represented will detect so and must repair their local states. Thus no correct node does a reset at the next pulse. By restriction R6, a repair is done by the EVALSTATE_WRAPPER procedure as a function of the global application snapshot such that the new global state will be legal. □

Closure: Following Lemma 3.2.1 the closure proof reduces to case (2.a.) in the proof of convergence, for the case in which the global state is legal. Thus, following restriction R4 the EVALSTATE_WRAPPER procedure evaluates correctly that the global snapshot is legal and thus all correct nodes do $reset :=$*false*. □

This concludes the proof of the theorem.

□

3.3 Example of Stabilizing a Non-stabilizing Algorithm

To illustrate our method and to elucidate its generality we will provide a specific example of the conversion of a well known non-stabilizing algorithms to its stabilizing counterpart.

To stabilize the protocol using our scheme the following needs to be identified: the application state, the sampling points, the bound Δ on the real-time skew between correct nodes' sampling points in a legal state, the GETSTATE_WRAPPER procedure, the EVALSTATE_WRAPPER procedure and how it characterizes the legal states and how it does a repair, the initialization of the basic program following a global reset, the required minimal length of the cycle.

Consider the Byzantine clock synchronization algorithm in [26]. Informally that algorithm operates as follows: The processes resynchronize their clocks every *PER* time period. A process expects the time at the next resynchronization to equal ET. When a process's local time reaches ET it broadcasts a (signed) message stating "the time is ET". Alternatively, when a process receives such a message from $f + 1$ distinct nodes it knows that at least one correct node advanced its local time to ET and thus it resets its clock to ET. Note that this algorithm does not utilize a rounds structure.

It is interesting to note that the candidate protocol above uses signed messages in a way that does not comply with R2, because replay of signed messages from previous incarnations of the protocol can destroy the synchronization of the clocks of the correct nodes. One can transform the protocol to conform with R2, by using Byzantine Agreement instead of sending signed messages. The difficulty above is inherent in stabilizing protocols that use digital signatures.

- The application state will be comprised of the ET variable only.
- Practically any point throughout the inter-*PER* period avoiding the vicinity of the resynchronization events is safe for sampling. For illustrative purposes we will define a sampling point at every time that equals $ET + PER/2$. It is clear that the ET variable is quiescent around this point when the algorithm is in a legal global state.
- The algorithm can be initialized with the required bound of σ real-time units between the different nodes. This will not affect the precision of the algorithm which will stay d. That

will yield a real-time skew between correct nodes' sampling points in a legal state of $\Delta = d + \mathit{PER} \cdot (1 + \rho)$.

- The sampling point is identified by the GETSTATE_WRAPPER procedure through the local state event of $clocktime = ET + \mathit{PER}/2$, at which the ET value is read into the localAppState variable.

- The EVALSTATE_WRAPPER procedure identifies the legal application states as those in which there are at least $n - f$ identical ET values. A repair is done by a node by setting its ET value to equal the other $n - f$ or more ET values in the application snapshot if it was evaluated as legal.

- Following a reset a node should initialize the algorithm by setting its ET variable to some pre-defined value, e.g. $ET = 0$. As mentioned before, the initial skew of σ will affect the accuracy but not the precision, as early and fast nodes will reach their subsequent ET before the others, but the others late and slow nodes will set their clock accordingly upon receiving $f + 1$ messages which is uncorrelated to the initialization skew.

- The required minimal cycle length equals $\mathit{PER}/2$ in case the pulse correlates with the reading of the sampling point and some correct nodes will have to wait until the next sampling point. The protocol then needs to allow for a full Byzantine agreement to terminate, in addition to a few round-trip rounds. Thus the required minimal cycle length equals $\mathit{PER}/2 + (2f + 3)$ rounds.

3.4 Analysis

We require *cycle* to be chosen s.t. $cycle_{min} > \sigma + \Sigma + \mathit{agreement\_duration}$.

From an arbitrary state in which the system is coherent it can take up to *pulse_conv* real-time until the pulses synchronize. Subsequent to the pulses it can take in the order of $\Sigma +$ *agreement_duration* real-time to reach a decision on a reset. The steady-state time complexity equals the time overhead from the pulse until the EVALSTATE_WRAPPER procedure terminates. Again this equals about $\Sigma +$ *agreement_duration* time. With few faults and/or a fast network this becomes in the order of Σ, which is largely determined by the user and can be as low as $4d$ if the basic program allows for frequent sampling points. The message complexity is expressed in point-to-point messages. The message complexity of the steady state is roughly n^2 messages for the pulse synchronization procedure, and $f' \cdot n^2$ for the agreement algorithm.

Note that the agreement instances initiated by correct nodes will always terminate within 2 communication rounds, this is due to the early stopping property of the consensus algorithm which terminates within 2 rounds if all correct nodes hold the same initial agreement value. Thus the communication complexity is that of the actual number of faulty nodes.

The algorithm is *fault-containing*, in the sense that if faulty nodes behave "correctly" such that a correct node detects that it is not in synch with a legal global snapshot then the node can "repair" itself. Thus even though we present a reset-based protocol, repair is done up to a certain amount of concurrent faults. This is because our protocol is Byzantine resilient, thus a non-Byzantine fault or inconsistency will be masked by the protocol while the affected non-faulty node can perform

a repair. Only if there should be more than f faults and inconsistencies would a system reset be performed.

The algorithm is also time-adaptive, the number of rounds executed in every cycle equals the number of actual faults, f'. This is due to the early-stopping feature of the agreement algorithm which terminates within $f' \leq f$ rounds.

Note that if solving a certain Byzantine problem can be reduced to consensus (or agreement) on the future value of the global state at the next pulse, (e.g. clock synchronization, see Chapter 4), as opposed to reaching agreement on the current value of every node, then the agreement algorithm presented can be used to achieve 2-round early stopping subsequent to every pulse. Thus based on the global application snapshot at the last pulse, it can be calculated what the global state should be at this pulse. Thus if all correct nodes previously agreed on the state of every other node, which comprises the global snapshot, then they can enter agreement with consensus on the expected states for all nodes. The early stopping feature of the consensus algorithm in Chapter 4 ensures that if all correct nodes hold the same initial value to be agreed on then consensus is reached within two rounds. This makes the steady-state case extremely cost-efficient with a minimal overhead of 2 rounds. Only following a transient failure might full agreement be executed on the values of the faulty nodes, since different correct nodes may then hold different values for the same nodes.

3.5 The BYZ_AGREEMENT algorithm

The Byzantine agreement algorithm used here extends the consensus approach taken in Chapter 4, Section 4.4 in using explicit time bounds in order to address the variety of potential problems that may arise when the system is stabilizing.

We assume that timers of correct nodes are always within $\bar{\sigma}$ of each other. More specifically, we assume that nodes have timers that reset periodically, say at intervals $\leq cycle'$. Let $T_p(t)$ be the reading of the timer at node p at real-time t. We thus assume that there exists a bound such that for every real-time t, when the system is coherent,

$$\forall p, q \text{ if } \bar{\sigma} < T_p(t), T_q(t) < cycle' - \bar{\sigma} \text{ then } |T_p(t) - T_q(t)| < \bar{\sigma} \ .$$

The bound $\bar{\sigma}$ includes all drift factors that may occur among the timers of correct nodes during that period. When the timers are reset to zero it might be, that for a short period of time, the timers may be further apart. The pulse synchronization algorithm in Chapter 7 satisfies the above assumptions and implies that $\bar{\sigma} > d$.

We use the following notations in the description of the agreement procedure:

- Let $\bar{d}$ be the duration of time equal to $(\bar{\sigma} + d) \cdot (1 + \rho)$ time units on a correct node's timer. Intuitively, $\bar{d}$ can be assumed to be a duration of a "phase" on a correct node's timer.

The *consensus-broadcast* and the *broadcast* primitives are defined in 4.4. Note that an *accept* is issued within the broadcast primitive.

It is assumed that the BROADCAST and CONSENSUS-BROADCAST primitives are implicitly initiated when a corresponding message arrives. A correct node participates in these primitives only after the $\sigma \cdot (1 + \rho)$ interval following its pulse. Any corresponding message received in this interval is held and processed only at the end of this interval.

The BYZ_AGREEMENT algorithm is presented in a somewhat different style. Each step has a condition attached to it, if the condition holds and the timer value assumption holds, then the step is to be executed. Notice that only the step needs to take place at a specific timer value. It is assumed that the internal procedures invoked as a result of the BYZ_AGREEMENT algorithm are implicitly associated with the agreement procedure.

```
Algorithm BYZ_AGREEMENT on (p, Val, T)     /* invoked at node q */

broadcasters := ∅;  value :=⊥;
if p = q then send (initialize, q, Val, T + d̄, 1) to all;  /* the General */
by time (T + d̄) :
   if received (initialize, p, Val, T + d̄, 1) then
      consensus-broadcast(p, Val, T + d̄, 1);
by time (T + 3d̄) :
   if accepted (p, v, T + d̄, 1) then
         value := v;
by time (T + (2f + 3)d̄) :
   if value ≠⊥ then
         broadcast (q, value, T + d̄, ⌊(T_q − T − d̄)/(2d̄)⌋ + 1);
         stop and return value.
at time (T + (2r + 1)d̄) :
   if (|broadcasters| < r − 1) then
         stop and return value.
by time (T + (2r + 1)d̄) :
   if accepted (p, v′, T + d̄, 1) and r − 1 distinct messages (p_i, v′, T + d̄, i)
            where ∀i, j  2 ≤ i ≤ r, and p_i ≠ p_j ≠ p then
         value := v′;
```

Figure 3.1: The BYZ_AGREEMENT algorithm

The BYZ_AGREEMENT algorithm satisfies the following typical properties:

Termination: The protocol terminates in a finite time;
Agreement: The protocol returns the same value at all correct nodes;
Validity: If the initiator is correct, then the protocol returns the intiator's value;

Nodes stop participating in the BYZ_AGREEMENT protocol when they are instructed to do so. They stop participating in the broadcast primitive $2\bar{d}$ after they terminate BYZ_AGREEMENT.

DEFINITION 3.5.1 *We say:*

A node ***returns*** *a value* m *if it has stopped and returned* $value = m$.

A node p ***decides*** *if it stops at that timer time and returns a* $value \neq \perp$.

A node p ***aborts*** *if it stops and returns* $\perp$.

Theorem 3.5.1 *The* BYZ_AGREEMENT *satisfies the Termination property. When* $n > 3f$*, it also satisfies the Agreement and Validity properties.*

Proof: The proof follows very closely to the proof of the algorithm in 4.4. Notice, that there is a difference of one $\bar{d}$ resulting from the initiation of the protocol by a specific node, followed by a consensus. Another difference is that the General itself is one of the nodes, so if it is faulty there are only $f - 1$ potential faults left.

Lemma 3.5.1 *If a correct node aborts at time $T + (2r + 1)\bar{d}$ on its timer, then no correct node decides at a time $T + (2r' + 1)\bar{d} \geq T + (2r + 1)\bar{d}$ on its timer.*

Lemma 3.5.2 *If a correct node decides by time $T + (2r + 1)\bar{d}$ on its timer, then every correct node decides by time $T + (2r + 3)\bar{d}$ on its timer.*

Termination: Lemma 3.5.2 implies that if any correct node decides, all decide and stop. Assume that no correct node decides. In this case, no correct node ever invokes a broadcast $(p, v, T + \bar{d}, \_)$. By the consensus-broadcast properties in 4.4, no correct node will ever be considered as broadcaster. Therefore, by time $T + (2f + 3)\bar{d}$ on their timers, all correct nodes will have at most f broadcasters and will abort and stop. □

Agreement: If no correct node decides, then all abort, and return to the same value. Otherwise, let q be the first correct node to decide. Therefore, no correct node aborts. The value returned by q is the value v of the accepted $(p, v, T + \bar{d}, 1)$ message. By the consensus-broadcast properties in 4.4, all correct nodes accept $(p, v, T + \bar{d}, 1)$ and no correct node accepts $(p, v', T + \bar{d}, 1)$ for $v \neq v'$. Thus all correct nodes return the same value. □

Validity: If the initiator q is correct, all the correct nodes invoke the consensus-broadcast with the same value v' and invoke the protocol with the same timer time ($T+\bar{d}$). By the consensus-broadcast properties in 4.4, all correct nodes will stop and return v'. □

Thus the proof of the theorem is concluded.

□

Chapter 4

Self-stabilizing Byzantine Clock Synchronization using Pulse Synchronization

4.1 Specific Definitions

Basic notations:

We use the following notations though nodes do not need to maintain all of them as variables.

- $Clock_i$, the clock of node i, is a real value in the range 0 to $M-1$. Thus $M-1$ is the maximal value a clock can hold. Its progression rate is a function of node p_i's physical timer. The clock is incremented every time unit. $Clock_i(t)$ denotes the value of the clock of node p_i at real-time t.
- γ is the target upper bound on the difference of clock readings of any two correct clocks at any real-time. Our protocol achieves $\gamma = 3d + O(\rho)$.
- Let $a, b, g, h \in R^+$ be constants that define the linear envelope bound of the correct clock progression rate during any real-time interval.
- $\Psi_i(t_1, t_2)$ is the amount of clock time elapsed on the clock of node p_i during a real-time interval $[t_1, t_2]$ within which p_i was continuously correct. The value of Ψ is not affected by any wrap around of $clock_i$ during that period.
- *agreement_duration* represents the maximum real-time required to complete the chosen Byzantine consensus procedure used in Section 4.2. We assume $\sigma \leq \sigma + \textit{agreement\_duration} < \textit{cycle} \leq \textit{Cycle} + \textit{agreement\_duration}$. For simplicity of our arguments we also assume that $M > \textit{agreement\_duration}$ but this is not a necessary assumption.

Basic definitions:

DEFINITION 4.1.1 *The communication network is* **correct** *following* $\Delta_{net} = \textit{pulse\_conv} + \textit{agreement\_duration} + \sigma$ *real-time of continuous non-faulty behavior.*

DEFINITION 4.1.2 *A node is* **correct** *following* $\Delta_{node} = pulse\_conv + agreement\_duration + \sigma$ *real-time of continuous non-faulty behavior.*

- The **clock_state** of the system at real-time t is given by:

$$clock\_state(t) \equiv (clock_0(t),\ \ldots,\ clock_{n-1}(t)) \ .$$

- The systems is in a **synchronized clock_state** at real-time t if $\forall\, correct\ p_i, p_j$,

$$(|clock_i(t) - clock_j(t)| \leq \gamma)\ \vee\ (|clock_i(t) - clock_j(t)| \geq M - \gamma)\ .^{1}$$

DEFINITION 4.1.3 **The "Self-stabilizing Byzantine Clock Synchronization Problem"**

Convergence: *Starting from an arbitrary system state, s, the system reaches a synchronized clock_state after a finite time.*

Closure: *If s is a synchronized clock_state of the system at real-time t_0 then $\forall\, real\ time\ t \geq t_0$,*

1. *clock_state(t) is a synchronized clock_state,*
2. *"Linear Envelope": for every correct node, p_i,*

$$a \cdot [t - t_0] + b \leq \Psi_i(t_0, t) \leq g \cdot [t - t_0] + h \ .$$

The second Closure condition intends to bound the effective clock progression rate in order to defy a trivial solution.

4.2 Self-stabilizing Byzantine Clock Synchronization

A major challenge of self-stabilizing clock synchronization is to ensure clock synchronization even when nodes may initialize with arbitrary clock values. This, as mentioned before, requires handling the wrap around of clock values. The algorithm we present employs as a building block an underlying self-stabilizing Byzantine pulse synchronization procedure (e.g. Chapter 7 or Chapter 2). In the pulse synchronization problem nodes invoke pulses regularly, ideally every *Cycle* time units. The goal is for the different correct nodes to do so in tight synchrony of each other. To synchronize their clocks, nodes execute at every pulse Byzantine consensus on the clock value to be associated with the next pulse event[2]. When pulses are synchronized, then the consensus results in synchronized clocks. The basic algorithm uses strong consensus to ensure that once correct clocks are synchronized at a certain pulse, and thus enter the consensus procedure with identical values, then they terminate with the same identical values and keep the progression of clocks continuous and synchronized[3].

[1] The second condition is a result of dealing with bounded clock variables.

[2] It is assumed that the time between successive pulses is sufficient for a Byzantine consensus algorithm to initiate and terminate in between.

[3] The pulse synchronization building block does not use the value of the clock to determine its progress, but rather intervals measured on the physical timer.

The Basic Clock Synchronization Algorithm

The basic clock synchronization algorithm is essentially a self-stabilizing version of the Byzantine clock synchronization algorithm in [26].

We call it PBSS-CLOCK-SYNCH (for *Pulse-based Byzantine Self-stabilizing Clock Synchronization*). The agreed clock time to be associated with the next pulse (next "time for synchronization" in [26]) is denoted by ET (for *Expected Time*, as in [26]). Synchronization of clocks is targeted to happen every *Cycle* time units, unless the pulse is invoked earlier (or later)[4].

```
Algorithm PBSS-CLOCK-SYNCH
at "pulse" event                                   /* received the internal pulse event */
begin
 1. Clock := ET;
 2. Revoke possible other instances of PBSS-CLOCK-SYNCH and
        clear all data structures besides ET and Clock;
 3. Wait until σ(1 + ρ) time units have elapsed since pulse;
 4. Next_ET := BYZ_CONSENSUS((ET + Cycle) mod M, σ);
 5. Clock := (Clock + Next_ET − (ET + Cycle)) mod M;    /* posterior adjust. */
 6. ET := Next_ET;
end
```

Figure 4.1: The self-stabilizing Byzantine clock synchronization algorithm

The internal pulse event is delivered by the pulse synchronization procedure. Any pulse synchronization algorithm that delivers synchronized pulses by solving the "Self-stabilizing Pulse Synchronization Problem", in the presence of at most f Byzantine nodes, where $n \geq 3f + 1$, such as the pulse procedures in Chapter 7 or Chapter 2, can be executed in the background.

The pulse event aborts any possible on-going invocation of PBSS-CLOCK-SYNCH (and thus any on-going instant of BYZ_CONSENSUS) and resets all buffers. The synchronization of the pulses ensures that the PBSS-CLOCK-SYNCH procedure is invoked within σ real-time units of its invocation at all other correct nodes.

Line 1 sets the local clock to the pre-agreed time associated with the current pulse event. Line 3 intends to make sure that all correct nodes invoke BYZ_CONSENSUS only after the pulse has been invoked at all others, without remnants of past invocations, which are revoked at Line 2. Past remnants may exist only during or immediately following periods in which the system is not coherent.

In Line 4 BYZ_CONSENSUS intends to reach consensus on the next value of ET. One can use a synchronous consensus algorithm with rounds of size $(\sigma + d)(1 + 2\rho)$ or asynchronous style consensus in which a node waits to get $n - f$ messages of the previous round before moving to the next round. We assume the use of a Byzantine consensus procedure tolerating f faults when $n \geq 3f + 1$. A correct node joins BYZ_CONSENSUS only concomitant to an internal pulse event, as instructed by the PBSS-CLOCK-SYNCH. This contains the possibility of faulty nodes to initiate consensus at arbitrary times.

Line 5 is a posterior clock adjustment. It increments the clock value with the difference between the agreed time associated with the next pulse and the node's pre-consensus estimate for

[4] *Cycle* has the same function as PER in [26].

the time associated with the next pulse (the value which it entered the consensus with). This is equivalent to incrementing the value of ET that the node was supposed to hold at the pulse according to the agreed $Next\_ET$ with the elapsed time from the pulse and until the termination of BYZ_CONSENSUS. This intends to expedite the time to reach synchronization of the clocks. In case that the clock_state before Line 5 was not a synchronized clock_state then a synchronized clock_state is attained following termination of BYZ_CONSENSUS at all correct nodes, rather than at the next pulse event. Note that in the case that all correct nodes hold the same ET value at the pulse, then the posterior clock adjustment adds a zero increment to the clock value.

Note that when the system is not yet coherent, following a chaotic state, pulses may arrive to different nodes at arbitrary times, and the ET values and the clocks of different nodes may differ arbitrarily. At that time not all correct nodes will join BYZ_CONSENSUS and no consistent resultant value can be guaranteed. Once the pulses synchronize (guaranteed by the pulse synchronization procedure to happen within a single cycle) all correct nodes will join the same instant of BYZ_CONSENSUS and will agree on the clock value associated with the next pulse. From that time on, as long as the system stays coherent the clock_state remains a synchronized clock_state.

The use of Byzantine consensus tackles the clock wrap-around in a trivial manner at all correct nodes.

Note that instead of simply setting the clock value to ET we could use some *Clock-Adjustment* procedure (cf. [26]), which receives a parameter indicating the target value of the clock. The procedure runs in the background, it speeds up or slows down the clock rate to smoothly reach the adjusted value within a specified period of time. This procedure should also handle the clock wrap around.

Theorem 4.2.1 PBSS-CLOCK-SYNCH *solves the "Self-stabilizing Byzantine Clock Synchronization Problem".*

Proof:
Convergence: Let the system be coherent but in an arbitrary state s, with the nodes holding arbitrary clock values. Consider the first correct node that completed line 3 of the PBSS-CLOCK-SYNCH algorithm. Since the system is coherent, all correct nodes invoked the preceding *pulse* within σ of each other. At the last pulse all remnants of previously invoked instances of BYZ_CONSENSUS were flushed by all the correct nodes. A correct node does not initiate or join BYZ_CONSENSUS before waiting $\sigma(1+\rho)$ time units subsequent to the pulse, hence not before all correct nodes have invoked a pulse and subsequently flushed their buffers. Thus all correct nodes will eventually join BYZ_CONSENSUS, thus BYZ_CONSENSUS will initiate and terminate successfully.

At termination of the first instance of BYZ_CONSENSUS following the synchronization of the pulses, all correct nodes agree on the clock value to be associated with the next pulse invocation. Subsequently, all correct nodes adjust their clocks, post factum, according to the agreed ET. Note that this posterior adjustment of the clocks does not affect the time span until the invocation of the next pulse but rather updates the clocks concomitantly to and in accordance with the newly agreed ET. This has an effect only if the correct nodes joined BYZ_CONSENSUS with differing values. Hence if all correct nodes join BYZ_CONSENSUS with the same ET then the adjustment equals zero. Since all correct pulses arrived within σ real-time units of each other, after the posterior clock adjustment of the last correct node, all correct clocks values are within

$$\gamma_1 = \sigma(1+\rho) + (\sigma + \mathit{agreement\_duration}) \cdot 2\rho$$

of each other. The 2ρ is the maximal drift rate between any two correct clocks (whereas ρ is their drift with respect to real-time). Observe that $\gamma_1 \leq \gamma$ and therefore the state of the system is a synchronized clock_state. This concludes the Convergence condition.

□

Closure: Recall that system coherence is defined as a continuous non-faulty behavior of the communication network and a large enough fraction of the nodes for at least some minimal period of time. The proof of the Closure condition assumes the correct nodes have synchronized their ET values, thus setting this minimal time to be at least $cycle_{max}$ + *agreement_duration* time, ensuring synchronization of the variables.

Let the system be in a synchronized clock_state and w.l.o.g. assume all correct nodes hold synchronized and identical ET values. Observe that although the correct nodes have synchronized their ET values this does not necessarily imply all correct nodes hold the same ET value at every point in time. At a brief time subsequent to the termination of BYZ_CONSENSUS, only a part of the correct nodes may have set the ET to the new agreed value while the rest of the correct nodes currently holding the old ET value will set ET to the new value in a brief time. We first prove the first Closure condition (*precision*). In this case, each correct node adjusts its clock immediately subsequent to the pulse, but the posterior clock adjustment has no effect since the consensus value equals the value it joined BYZ_CONSENSUS with. To simplify the discussion assume for now that no wrap around of any correct clock takes place during the time that the pulse arrives at the first correct node and until the pulse is invoked at the last correct node. Immediately after the pulse is invoked at the last correct node and its subsequent clock adjustment, all correct clocks are within $\gamma_0 = \sigma(1 + \rho)$ of each other.

From that point on, clocks of correct nodes drift apart at a rate of 2ρ of each other. As long as no wrap around of the clocks takes place and no pulse arrives at any correct node, the clocks are at most $\gamma_0 + \Delta T \cdot 2\rho$ apart, where ΔT is the real-time elapsed since the invocation of the pulse at the first correct node. To estimate the maximal clock difference, γ, at any time, we will consider the following complementary cases:

P1) Prior to the next pulse event at the first correct node.

P2) When a pulse arrives at some correct node.

P3) Immediately after the last node invokes its next pulse event.

Note that in this case we do not need to consider the posterior adjustment of the clocks at Line 5.

Case P1 cannot last more than $\Delta T = cycle_{max}$, since by the end of that time interval all correct nodes will have invoked the pulse, reducing to case P2 or P3. The discussion above implies $\gamma = \gamma_0 + cycle_{max} \cdot 2\rho$.

Case P3 implies that clock readings are at most γ_0 apart, since all nodes invoke the pulses within σ.

To analyze case P2 consider that the next pulse event has been invoked at some node, p. The following situations may take place:

P2a) Following its clock adjustment, the clock of p holds the maximal clock value among all correct clocks at that moment.

P2b) Following its clock adjustment, the clock of p holds the minimal clock value among all correct clocks at that moment.

P2c) Neither of the above.

In case P2a, since p holds the maximal clock value, we claim that no other clock reading can read less than $ET_{lastpulse} + cycle_{min} \cdot (1 - \rho)$. Assume by contradiction the existence of a correct node q whose clock reading is less than this value. Further assume that node q received the same set of messages from the same sources and at the same time as node p. These events caused node p to invoke its pulse and would necessarily cause node q to also invoke a pulse. The elapsed time on the clock of node q between the current pulse and the previous is thus less than $cycle_{min} \cdot (1 - \rho)$ which is less than $cycle_{min}$ real-time after its previous pulse. A contradiction to the definition of $cycle_{min}$. Node p just adjusted its clock which thus reads $ET = ET_{lastpulse} + \mathit{Cycle}$. Due to the clock skew the clock difference may increase an additional $2\rho\sigma$ until the node invokes its pulse and the case reduces to P3. The discussion above implies $\gamma = (ET_{lastpulse} + \mathit{Cycle}) - (ET_{lastpulse} + cycle_{min} \cdot (1 - \rho)) + 2\rho\sigma = \mathit{Cycle} - cycle_{min} \cdot (1 - \rho) + 2\rho\sigma$.

In case P2b, the clock readings of all other nodes that have invoked a pulse can not be more than γ_0 apart (case P3). The clock reading of any node that has not invoked a pulse yet should be less than $cycle_{max}$ following similar reasoning as in case P2a. Node p just adjusted its clock which thus reads $ET = ET_{lastpulse} + \mathit{Cycle}$. Due to the clock skew the clock difference may increase an additional $2\rho\sigma$ until the node invokes its pulse and the case reduces to P3. The discussion above implies $\gamma = (ET_{lastpulse} + cycle_{max} \cdot (1 + \rho)) - (ET_{lastpulse} + \mathit{Cycle}) + 2\rho\sigma = cycle_{max} \cdot (1 + \rho) - \mathit{Cycle} + 2\rho\sigma$.

For case P2c, if the nodes holding the minimal clock reading and maximal clock reading already invoked pulses, then the clock difference reduces to case P3.

If neither of the nodes holding the minimal and maximal clock values have not invoked their pulses yet, then the clock difference reduces to case P1.

Otherwise, if either the node holding the minimal or the maximal clock value already invoked its pulse then one of the bounds of P2a or P2b hold until the other node invokes its pulse.

We now consider the case that a clock wrap around takes place at some ΔT real-time after the last pulse is invoked in the synchronized cycle. From the discussion earlier we learn that at the moment prior to the first correct clock wraps around, the correct clocks are at most γ apart. Therefore, all correct clocks will wrap around within at most another γ time. During the intermediate time, any two correct clocks, i, j, for which one has wrapped around and the other not, satisfy $|clock_i(t) - clock_j(t)| \geq M - \gamma$. Thus we proved that the maximal clock difference will remain less than γ or greater than $M - \gamma$, which completes the first Closure condition.

Henceforth, the bound on the clock differences of correct nodes will equal the maximal of the three values calculated above. Formally this yields $\gamma = max[cycle_{max} \cdot (1 + \rho) - \mathit{Cycle} + 2\rho\sigma,\ \mathit{Cycle} - cycle_{min} \cdot (1 - \rho) + 2\rho\sigma,\ \sigma(1 + \rho) + cycle_{max} \cdot 2\rho]$. The explicit value is dependent on the relationship between $cycle_{max}$, $cycle_{min}$ and Cycle, which is determined by the pulse synchronization procedure. The explicit value of γ is presented in Section 4.3. This concludes the first Closure condition.

For the second Closure condition, note that Ψ_i, as defined in Section 4.1, represents the actual deviation of an individual correct clock (p_i) from the real-time interval during which it progresses. This is equivalent to the maximal actual difference between the clock value and real-time during

a real-time interval in which real-time and the clock value were equal at the beginning of the interval. The *accuracy* of the clocks is the bound on the actual deviation of correct clocks from any finite real-time interval or rate of deviation from the progression of real-time. Thus it suffices to show that correct clocks progress with an accuracy that is a linear function of every finite real-time interval to satisfy the second Closure condition.

The clock progression has an inherent deviation from any real-time interval due to the physical clock skew. In addition, the clocks are repeatedly adjusted at every pulse in order to tighten the precision, which can further deviate the clocks progression from the progression of the real-time during the interval. In Chapter 7 it is shown that the pulses progress with a linear envelope of any real time interval. The accuracy in a cycle equals the bound on the clock adjustment $|t_{pulse} - ET_{pulse}|$, where t_{pulse} is the clock value at the pulse at the moment prior to the adjustment of the clock to ET_{pulse}. Under perfect conditions, i.e. no clock skew and zero clock adjustment $t_{pulse} = ET_{pulse}$. This would further equal real-time should the clocks have initiated with real-time values. Thus it suffices to show that the adjustment to the clocks at every pulse is a linear function of the length of the cycle. The upper and lower bounds on the value t_{pulse} is determined by the bound on the effective cycle length and accounts for the clock skew and the accuracy of the pulses (bound on the deviation of the pulses from perfect regularity). Let $cycle_{min}$ and $cycle_{max}$ denote the lower bound and upper bound respectively on the cycle length in real-time units. Hence,

$$ET_{prev\text{-}pulse} + cycle_{min} \cdot (1 - \rho) \leq t_{pulse} \leq ET_{prev\text{-}pulse} + cycle_{max} \cdot (1 + \rho) \ .$$

The adjustment to the correct clocks, ADJ, is thus bounded by

$$ET_{pulse} - [ET_{prev\text{-}pulse} + cycle_{max} \cdot (1 + \rho)] \leq 0 \leq ADJ$$

and

$$ADJ \leq 0 \leq ET_{pulse} - [ET_{prev\text{-}pulse} + cycle_{min} \cdot (1 - \rho)] \ ,$$

which translates to

$$ET_{prev\text{-}pulse} + \mathbf{Cycle} - [ET_{prev\text{-}pulse} + cycle_{max} \cdot (1 + \rho)] \leq ADJ$$

and

$$ADJ \leq ET_{prev\text{-}pulse} + \mathbf{Cycle} - [ET_{prev\text{-}pulse} + cycle_{min} \cdot (1 - \rho)] \ ,$$

which translates to

$$\mathbf{Cycle} - cycle_{max} \cdot (1 + \rho) \leq ADJ \leq \mathbf{Cycle} - cycle_{min} \cdot (1 - \rho) \ .$$

As can be seen, the bound on the adjustment to the clock is linear in the effective cycle length. The bounds on the effective cycle length are guaranteed by the pulse synchronization procedure to be linear in the default cycle length. Thus the accuracy of the clocks are within a linear envelope of any real-time interval. The actual values of $cycle_{min}$ and $cycle_{max}$ are determined by the specific pulse synchronization procedure used. This concludes the Closure condition.

□

Thus the algorithm is self-stabilizing and performs correctly with f Byzantine nodes for $n \geq 3f + 1$.

□

A Clock Synchronization Algorithm without Consensus

We suggest a simple additional Byzantine self-stabilizing clock synchronization algorithm using pulse synchronization as a building block that does not use consensus.

Our second algorithm resets the clock at every pulse[5]. This approach has the advantage that the nodes never need to exchange and synchronize their clock values and thus do not need to use consensus. This version is useful for example when M, the upper-bound on the clock value, is relatively small. The algorithm has the disadvantage that for a large value of M, a large *Cycle* value is required. This enhances the effect of the clock skew, thus negatively affecting the precision and the accuracy at the end of the cycle. Note that the precision and accuracy of CYCLE-WRAP-CS equals that of PBSS-CLOCK-SYNCH.

```
Algorithm CYCLE-WRAP-CS
at "pulse" event                          /* received the internal pulse event */
begin
   Clock := 0;
end
```

Figure 4.2: Additional CS algorithm in which the clock wraps-around every cycle

A Clock Synchronization Algorithm using an Approximate Agreement Approach

We suggest an additional self-stabilizing Byzantine clock synchronization algorithm using pulse synchronization as a building block, denoted APPROX-CS.

The algorithm uses an approximate agreement approach in order to get continuous clocks with high precision and accuracy on expense of the message complexities and early-stopping property. The precision and the accuracy are $2\sigma+O(\rho)$ and thus improve on those of PBSS-CLOCK-SYNCH.

```
Algorithm APPROX-CS
at "pulse" event                          /* received the internal pulse event */
begin
 1. Clock-at-pulse := Clock;
 2. Revoke possible other instances of APPROX-CS and
        clear all data structures besides Clock-at-pulse;
 3. Wait until σ(1 + ρ) time units have elapsed since pulse;
 4. Clock_Consensus := APPROX_BYZ_AGREE(Clock-at-pulse);
 5. Clock := (Clock_Consensus + elapsed-time-since-pulse) mod M;
end
```

Figure 4.3: Self-stabilizing Byzantine Approximate Clock Synchronization algorithm

In Line 4 of APPROX-CS the nodes invoke approximate-like agreement on their local clock value at the time of the last pulse, denoted Clock-at-pulse. In case that the system state was a

[5]This approach has been suggested by Shlomi Dolev as well.

synchronized clock_state then the resultant value $Clock_{Consensus}$ is guaranteed by APPROX_BYZ_AGREE to be in the range of the initial clock values of the correct nodes. If the clocks were not synchronized then the resultant agreed value may be in any range. In Line 5 every correct node sets its clock to equal the agreed clock value associated with the last pulse, $Clock_{Consensus}$, incremented with the time that has elapsed on its local timer since the pulse.

Algorithm APPROX_BYZ_AGREE(value)
begin
1. Invoke SS-BYZ-AGREE () on $value$;
2. After termination of all SS-BYZ-AGREE instances (substitute missing values with 0) Do:
3. Find largest set of values within γ+σ of each other (if several, choose set harboring smallest value ≥ 0);
4. Find median of the set, identify its antipode := (median + $\lfloor M/2 \rfloor$) mod M;
5. Discard the f immediate values from each side of the antipode;
6. Return the median of the remaining values;
end

Figure 4.4: Self-stabilizing Byzantine Approximate Agreement

In order to be self-contained we bring the definition of *Approximate Agreement*, defined in [27].

Formally, the goal of ϵ-Approximate Agreement is to reach the following: let there be n processes $p_1, ..., p_n$, each starts with an initial value $v_i \in \mathbb{R}$ and may decide on a value $d_i \in \mathbb{R}$.

1. **(Approximate Agreement)** If p_i and p_j are correct and have decided then $|d_i - d_j| \leq \epsilon$.

2. **(Validity)** If p_i is correct and has decided then there exists two correct nodes p_j, p_k such that $v_j \leq d_i \leq v_k$, (the decision value of every correct node is in the range of the initial values of the correct nodes).

3. **(Termination)** All correct nodes eventually decide.

The approximate agreement protocol in [27] cannot be used as-is in the self-stabilization model as the notions of "highest" value and "lowest" value are not defined when nodes can initialize with values reaching their bounds, M. Faulty nodes can in this case cause different correct nodes to view the extremes of the values as complete opposites. To overcome the lack of total order relation introduced by the self-stabilization model, APPROX_BYZ_AGREE thus combines the approximate agreement algorithm of [27] with Byzantine agreement as follows: run separate Byzantine agreements in parallel on every node's value in order to agree on the value of each node. Thus all correct nodes will hold identical multisets and henceforth the heuristics of [27] will be executed on exactly the same values at all correct nodes. The APPROX_BYZ_AGREE procedure satisfies the conditions for classic approximate agreement, while being self-stabilizing.

The SS-BYZ-AGREE procedure used is the Byzantine agreement of [73], though using our BROADCAST primitive presented in Section 4.4 in order to overcome the lack of any common reference to clock time among the correct nodes.

In Line 1 of APPROX_BYZ_AGREE, every node invokes Byzantine agreement on its value, within σ real-time of each other. Every instance of APPROX_BYZ_AGREE must terminate within some bounded time, thus all correct nodes can calculate a time when all the agreement instances have terminated at all correct nodes. In Line 3, after all the agreement instances have terminated and missing values are substituted with a 0, a set of supposedly synchronized values is searched for. Note that if not all instances of APPROX_BYZ_AGREE have terminated within the pre-calculated time-bound then the system must have been in a non-coherent state. Synchronized clock values can be up-to $\gamma + \sigma$ apart in the values agreed subsequent to Line2, due to the pulse uncertainty. In Line 4 the median of the set is identified, and will serve as an anchor for determining the order relation among the different values. In Line 5, the antipode (in the range $1..M$) of the median is identified; the f first values on each side of this antipode are then discarded. If the system is in a synchronized clock_state then all values that are outside of the values in the set identified earlier are discarded. Thus the median of the remaining values, returned in Line 6, is in the range of the initial values of the correct nodes.

Lemma 4.2.1 *The* APPROX_BYZ_AGREE *procedure satisfies all the conditions for ϵ-Approximate Agreement, for $\epsilon = 0$, when the system is in a synchronized clock_state*[6].

Proof: Note the validity of SS-BYZ-AGREE guarantees that the value decided by all correct nodes for node i is i's actual input value.

1. **Approximate_Agreement**: All correct nodes hold the same multiset of values following all terminations of the instances of SS-BYZ-AGREE , thus they all find the same set in Line 3 and hence do the exact same operations in lines 3-5, and thus return the same value in Line 6.

2. **Validity**: Let the system be in a synchronized clock_state. Thus the agreed clock values for all correct nodes subsequent to executing Line 2 are at most $\gamma + \sigma$ apart. Hence, the largest set found in Line 3 includes at least $n - f$ values. We now seek to prove that the decision value is in the range of the initial values of the correct nodes. Since $f < n/3$ it follows that all values that are not in the range (at most f) of this set are discarded in Line 5. Thus all remaining values must be in the range of the initial values of the correct nodes. In particular, the median of the remaining values is in the range of the initial values. This completes the proof of the validity condition.

3. **Termination**: Follows from the termination of SS-BYZ-AGREE .

□ □

The precision γ, is the bound on the clock differences of all correct nodes at any time.

Lemma 4.2.2 *The precision of* APPROX_BYZ_AGREE *is* $2\sigma + O(\rho)$.

[6]The notion "in the range of" remains undefined if the system is not in a synchronized clock_state. Thus the validity condition remains undefined for this case.

Proof: At the moment after all correct nodes have executed Line 5 in APPROX-CS their clocks differ by at most $\sigma + O(\rho)$, thus the clock differences are at most $\sigma + O(\rho)$ also at the forthcoming pulse invocation. The precision γ, is maximized at the moment that a correct node has set its clock subsequent to its execution of Line 5 in APPROX_BYZ_AGREE, while some other node has yet to execute this line. Following the validity condition, the agreed clock value $Clock_{Consensus}$, is within the initial clock values that was held by the correct nodes at their last pulse. As the system is in a synchronized clock_state thus these initial values were within $2\sigma + O(\rho)$ of each other. Thus the node that has just adjusted its clock, set it to a value that is within $2\sigma + O(\rho)$ of its clock at the moment before the adjustment. In particular this adjusted clock value is also within $2\sigma + O(\rho)$ of the clock value of any other correct node. This observation yields a precision of $\gamma = 2\sigma + O(\rho)$. □

The accuracy equals the maximal clock adjustment which for the same arguments as above yields an accuracy of $2\sigma + O(\rho)$.

A self-stabilizing Byzantine approximate agreement algorithm that knows how to handle bounded, wrapping values and thus does not need to reach exact agreement on every node's value, will supposedly yield a clock synchronization algorithm with time and message complexity comparable to PBSS-CLOCK-SYNCH with precision and accuracy of APPROX-CS. To the best of our knowledge no such approximate agreement algorithm exists.

4.3 Analysis and Comparison to other Clock Synchronization Algorithms

Our clock synchronization algorithm PBSS-CLOCK-SYNCH requires reaching consensus in every cycle. This implies that the cycle should be long enough to allow for the consensus procedure to terminate at all correct nodes. This implies having $cycle_{min} \geq 2\sigma + 3(2f + 4)d$, assuming that the BYZ_CONSENSUS procedure takes $(f+2)$ rounds of $3d$ each. The algorithm has the advantage that it uses the full time to reach consensus only following a catastrophic state in which correct nodes hold differing ET values. Once in a synchronized clock_state, all correct nodes participate in the consensus with the same initial consensus value which thus terminates within 2 communication rounds only, due to its early-stopping property. Hence, during steady state, in which the system is in a legal state, the time and message complexity overhead of PBSS-CLOCK-SYNCH is minimal.

For simplicity we also assume M to be large enough so that it takes at least a cycle for the clocks to wrap around.

Note that Ψ_i, defined in Section 4.1, represents the actual deviation of an individual correct clock, p_i, from a given real-time interval. The *accuracy* of the clocks is the bound on this deviation of correct clocks from any real-time interval. The clocks are repeatedly adjusted in order to minimize the accuracy. Following a synchronization of the clock values, that is targeted to occur once every *Cycle* time units, correct clocks can be adjusted by at most ADJ, where following Theorem 4.2.1,

$$\textit{Cycle} - cycle_{max} \cdot (1 + \rho) \leq ADJ \leq \textit{Cycle} - cycle_{min} \cdot (1 - \rho) \ ,$$

which, following $cycle_{min}$ and $cycle_{max}$ determined by the pulse synchronization procedure of Chapter 7 to equal $\textit{Cycle} - 11d$ and $\textit{Cycle} + 9d$ respectively, translates to

Algorithm	Self-stabilizing /Byzantine	Precision γ	Accuracy	Convergence Time	Messages
PBSS-CLOCK-SYNCH	SS+BYZ	$11d + O(\rho)$	$11d + O(\rho)$	$cycle_{max} + 3(2f+5)d$	$O(nf^2)$
CYCLE-WRAP-CS	SS+BYZ	$11d + O(\rho)$	$11d + O(\rho)$	$cycle_{max}$	$O(n^2)$
APPROX-CS	SS+BYZ	$3d + O(\rho)$	$3d + O(\rho)$	$cycle_{max}$	$O(nf)^2$
DHSS [26]	BYZ	$d + O(\rho)$	$(f+1)d + O(\rho)$	$2(f+1)d$	$O(n^2)$
LL-APPROX [75]	BYZ	$5\epsilon + O(\rho)$	$\epsilon + O(\rho)$	$d + O(\epsilon)$	$O(n^2)$
DW-SYNCH [34]*	SS+BYZ	0	0	$M2^{2(n-f)}$	$n^2 M 2^{2(n-f)}$
DW-BYZ-SS [34]	SS+BYZ	$4(n-f)\epsilon + O(\rho)$	$(n-f)\epsilon + O(\rho)$	$O(n)^{O(n)}$	$O(n)^{O(n)}$
PT-SYNC [66]*	SS	0	0	$4n^2$	$O(n^2)$

Table 4.1: Comparison of clock synchronization algorithms (ϵ is the uncertainty of the message delay). The convergence time is in pulses for the algorithms utilizing a global pulse system and in rounds for the other semi-synchronous protocols. PT-SYNC assumes the use of shared memory and thus the "message complexity" is of the "equivalent messages". The '*' denotes the use of a global pulse or global clock tick system.

$$-9d(1+\rho) - \rho \cdot \mathit{Cycle} \leq ADJ \leq 11d(1-\rho) + \rho \cdot \mathit{Cycle}\ .$$

The accuracy is thus $11d+O(\rho)$ real-time units. Should the initial clock values reflect real-time then this determines the accuracy of the clocks with respect to real-time (and not only with respect to real-time progression rate), as long as the system is coherent and clocks do not wrap around.

Recall that the precision γ, is the bound on the difference between correct clock values at any time. This bound is largely determined by the maximal clock value difference at the time in which a correct node has just set its clock and some other correct node is about to do it in a short time. It is guaranteed by Theorem 4.2.1 and the pulse synchronization tightness $\sigma = 3d$ of Chapter 7, to be:

$$\begin{aligned}
\gamma &= \max[cycle_{max} \cdot (1+\rho) - \mathit{Cycle} + 2\rho\sigma, \\
&\qquad \mathit{Cycle} - cycle_{min} \cdot (1-\rho) + 2\rho\sigma,\ \sigma(1+\rho) + cycle_{max} \cdot 2\rho] \\
&= \max[9d(1+\rho) + \rho \cdot \mathit{Cycle} + 2\rho\sigma,\ 11d(1-\rho) + \rho \cdot \mathit{Cycle} + 2\rho\sigma, \\
&\qquad 3d(1+\rho) + (\mathit{Cycle} + 9d) \cdot 2\rho] \\
&= 11d(1-\rho) + \rho \cdot \mathit{Cycle} + 2\rho\sigma = 11d + O(\rho)\ .
\end{aligned}$$

The bound on the difference between correct clock values immediately after all correct nodes have synchronized their clock value (at Line 1 or Line 5) is σ.

The only self-stabilizing Byzantine clock synchronization algorithms, to the best of our knowledge, are published in [31, 34]. Two randomized self-stabilizing Byzantine clock synchronization algorithms are presented, designed for fully connected communication graphs, use message passing which allow faulty nodes to send differing values to different nodes, allow transient and permanent faults during convergence and require at least $3f+1$ processors. The clocks wrap around,

where M is the upper bound on the clock values held by individual processors. The first algorithm assumes a common global pulse system and synchronizes in expected $M \cdot 2^{2(n-f)}$ global pulses. The second algorithm in [34] does not use a global pulse system and is thus partially synchronous similar to our model. The convergence time of the latter algorithm is in expected $O((n-f)n^{6(n-f)})$ time. Both algorithms thus have drastically higher convergence times than ours.

In Table 1 we compare the parameters of our protocols to previous classic Byzantine clock synchronization algorithms, to non-Byzantine self-stabilizing clock synchronization algorithms and to the prior Byzantine self-stabilizing clock synchronization algorithms. It shows that our algorithm achieves precision, accuracy, message complexity and convergence time similar to non-stabilizing algorithms, while being self-stabilizing.

The message complexity of PBSS-CLOCK-SYNCH is solely based on the underlying Pulse and Consensus procedures. Its inherent convergence time is $cycle_{max}$. The $O(nf^2)$ message complexity as well as the $+3(2f+5)d$ additive in the convergence time come from BYZ_CONSENSUS, the specific Byzantine consensus procedure we use. The pulse synchronization procedure we use from Chapter 7 has a message complexity of $O(n^2)$ and $6 \cdot cycle$ convergence time. Note that BYZ_CONSENSUS has two early-stopping features: It stops in a number of rounds dependent on the actual number of faults and if nodes initiate with the same values (same ET values) then it stops within 2 rounds.

Note that some of the algorithms cited in Table 1 refer to ϵ, the uncertainty in message delivery, rather than d, the end-to-end communication network delay.

The DW-SYNCH and PT-SYNCH algorithms cited in Table 1 make use of global clock ticks (common physical timer). Note that this does not make the clock synchronization problem trivial as such clock ticks can not be used to invoke agreement procedures and the nodes still need to agree on the clock values. The benefit of utilizing a global pulse systems is in the optimal precision and accuracy acquired (see [34]).

4.4 The Consensus and Broadcast Primitives

The BYZ_CONSENSUS Procedure

The BYZ_CONSENSUS procedure can implement many of the classical Byzantine consensus algorithms. It assumes that timers of correct nodes are always within $\bar{\sigma}$ of each other. More specifically, we assume that nodes have timers that reset periodically, say at intervals $\leq cycle'$. Let $T_i(t)$ be the reading of the timer at node p_i at real time t. We thus assume that there exists a bound such that for every time t, when the system is coherent,

$$\forall i, j \text{ if } \bar{\sigma} < T_i(t), T_j(t) < cycle' - \bar{\sigma} \text{ then } |T_i(t) - T_j(t)| < \bar{\sigma} .$$

The bound $\bar{\sigma}$ includes all drift factors that may occur among the timers of correct nodes during that period. When the timers are reset to zero it might be that, for a short period of time, the timers may be further apart. The pulse synchronization algorithm in Chapter 7 satisfies the above assumptions and implies $\bar{\sigma} \geq d$.

The self-stabilization requirement and the deviation that may arise from any synchronization assumption imply that any consensus protocol must be carefully specified. The consensus algorithm will function properly if it is invoked when the timers of correct nodes are within $\bar{\sigma}$ of each

other. The subtle point is to make sure that an arbitrary initialization of the procedure cannot cause the nodes to block or deadlock. Below we show how to update the early stopping Byzantine Agreement algorithm of Toueg, Perry and Srikanth [73] to become self-stabilization and to make it into a general consensus (vs. agreement) procedure.

The procedure does not assume any reference to real-time and no complete synchronization of the rounds, as is assumed in [73]. Rather it resets the local timers of correct nodes at each pulse which thus makes the timers within bounds of each other. The node invokes the procedure with the value to agree on and the local timer value. In the procedure nodes also consider all messages accumulated in their buffers that were accepted prior to the invocation, if they are relevant.

We use the following notations in the description of the consensus procedure:

- Let $\bar{d}$ be the duration of time equal to $(\bar{\sigma} + d) \cdot (1 + \rho)$ time units on a correct node's timer. Intuitively, $\bar{d}$ can be assumed to be a duration of a "phase" on a correct node's timer.
- The BROADCAST primitive is the primitive defined in Section 4.4 and is an adaptation of the one described in [73]. Note that an *accept* is issued within the BROADCAST primitive.

The main differences from the original protocol of [73] are:

- Instead of the General in the original protocol we use a virtual (faulty) "General" notion of a virtual node whose value is the assumed value of all correct nodes at a correct execution. It is the value with which the individual nodes invoke the procedure. Thus, every correct node does a CONSENSUS-BROADCAST of its initial Val in contrast to the original protocol in which only the General does this. If all correct nodes initiate with the same value and at the same timer time this will be the agreed value.
- The CONSENSUS-BROADCAST primitive has been modified by omitting the code dealing with the $init$ messages. All correct nodes send an echo of their initial values as though they previously received the $init$ message from the virtual General.
- It is assumed that the BROADCAST and CONSENSUS-BROADCAST primitives are implicitly initiated when a corresponding message arrives.
- The eventual termination is a function of the actual message exchange time among correct nodes and not the upper possible time a "round" may take.

BYZ_CONSENSUS is presented in a somewhat different style. Each step has a condition attached to it, if the condition holds and the timer value assumption holds, then the step is to be executed. Notice that only the step needs to take place at a specific timer value.

The BYZ_CONSENSUS procedure satisfies the following typical properties:

Termination: The protocol terminates in a finite time;

Agreement: The protocol returns the same value at all correct nodes;

Validity: If all correct nodes invoke the protocol with the same value and time, then the protocol returns that value;

It also satisfies the following **early stopping** properties:

Procedure BYZ_CONSENSUS(Val, T) /* *invoked at p with timer T* */

$broadcasters := \emptyset$; $value = \perp$;

Do CONSENSUS-BROADCAST $(General, Val, T, 1)$;

by time $(T + 2\bar{d})$:
  if accepted $(General, v, T, 1)$ **then**
    $value := v$;

by time $(T + (2f + 4)\bar{d})$:
  if $value \neq \perp$ **then**
    BROADCAST $(p, value, T, \lfloor \frac{T_i - T}{2\bar{d}} \rfloor + 1)$;
    stop and **return** $value$.

at time $(T + 2r\bar{d})$:
  if $(|broadcasters| < r - 1)$ **then**
    stop and **return** $value$.

by time $(T + 2r\bar{d})$:
  if accepted $(General, v', T, 1)$ **and** $r - 1$ **distinct messages** (q_i, v', T, i)
      where $\forall i, j \;\; 2 \leq i \leq r$, **and** $q_i \neq q_j$ **then**
    $value := v'$;

Figure 4.5: The BYZ_CONSENSUS procedure

ES-1 If all correct nodes invoke the protocol with the same consensus value and with the same timer value, then they all stop within two "rounds" of information exchange among correct nodes.

ES-2 If the actual number of faults is $f' \leq f$ then the algorithm terminates by $\min[T + (2f' + 6)\bar{d},\ T + (2f + 4)\bar{d}]$ on the timer of each correct node.

Notice that [ES-1] takes in practice significantly less time than the specified upper bound on the message delivery time.

We first prove the properties of the CONSENSUS-BROADCAST primitive and later we prove the correctness of the BYZ_CONSENSUS procedure.

The CONSENSUS-BROADCAST primitive and the BROADCAST primitive (defined in Section 4.4) satisfy the following [TPS-*] properties of Toueg, Perry and Srikanth [73], which are phrased in our system model.

TPS-1 *(Correctness)* If a correct node p does BROADCAST (p, m, τ, k) by $\tau + (2k - 2)\bar{d}$ on its timer, then every correct node accepts (p, m, τ, k) by $\tau + 2k\bar{d}$ on its timer.

TPS-2 *(Unforgeability)* If no correct node p does a BROADCAST (p, m, τ, k), then no correct node accepts (p, m, τ, k).

TPS-3 *(Relay)* If a correct node accepts (p, m, τ, k) by $\tau + 2r\bar{d}$, for $r \geq k$, on its timer then every other correct node accepts (p, m, τ, k) by $\tau + (2r + 2)\bar{d}$ on its timer.

TPS-4 *(Detection of broadcasters)* If a correct node accepts (p, m, τ, k) by $\tau + 2r\bar{d}$, on its timer then every correct node has $p \in broadcasters$ by $\tau + (2k + 1)\bar{d}$ on its timer. Furthermore,

Procedure CONSENSUS-BROADCAST $(General, v, \tau, 1)$
/ invoking a broadcast simulating the General */*
/ nodes send specific message with the same τ only once */*
/ multiple messages sent by an individual node are ignored*/*

send $(echo, General, v, \tau, 1)$ to all;

by time $(\tau + \bar{d})$:
if received $(echo, General, v, \tau, 1)$ from $\geq n - 2f$ distinct nodes **then**
$broadcasters := broadcasters \bigcup \{General\}$;
if received $(echo, General, v, \tau, 1)$ from $\geq n - f$ distinct nodes q **then**
send $(echo', General, v, \tau, 1)$ to all;

at any time:
if received $(echo', General, v, \tau, 1)$ from $\geq n - 2f$ distinct nodes **then**
send $(echo', General, v, \tau, 1)$ to all;
if received $(echo', General, v, \tau, 1)$ from $\geq n - f$ distinct nodes **then**
accept $(General, v, \tau, 1)$;

Figure 4.6: CONSENSUS-BROADCAST

if a correct node p does not BROADCAST any message, then a correct node can never have $p \in broadcasters$.

Additionally, the CONSENSUS-BROADCAST primitive also satisfies:

TPS-5 *(Uniqueness)* If a correct node accepts $(General, m, \tau, 1)$, then no correct node ever accepts $(General, m', \tau, 1)$ with $m' \neq m$.

Notice the differences from the original properties. The detection property does not require having $r \geq k$. In general, the relay property holds even earlier than $r \geq k$. The condition $r \geq k$ of when the property can be guaranteed is used to simplify the possible cases. At $r < k$, if an accept takes place as a result of getting $n - f$ echo messages, the adversary may cause the relay to take $3\bar{d}$ by rushing messages to one correct node and delay messages to and from others.

Theorem 4.4.1 *The* CONSENSUS-BROADCAST *primitive satisfies the five [TPS-*] properties.*

Proof:
Correctness: If all correct nodes send $(echo, General, v, \tau, 1)$ at time τ on their timers, then by Lemma 4.4.3 every correct node accepts $(General, v, \tau, 1)$ from $n - f$ correct nodes by $\tau + \bar{d}$ on its timer. Thus each correct node sends $(echo, General, v, \tau, 1)$ by that time and will accept $(General, v, \tau, 1)$ by $\tau + 2\bar{d}$ on their timers.

Unforgeability: If all correct nodes hold the same initial value v then no correct node will send $(echo, General, v', 1)$, thus no correct node will receive $n - f$ distinct $(echo, General, v', 1)$ messages. Therefore, no correct node will send $(echo', General, v', 1)$, and no correct node will ever receive $n - 2f$ or $n - f$ distinct $(echo', General, v', 1)$ messages. Thus, no correct node can accept $(General, v', 1)$.

Relay: If a correct node accepts $(General, v, \tau, 1)$ by $\tau + 2r\bar{d}$ on its timer, then it received $n - f$ distinct $(echo', General, v, \tau, 1)$ message by that time. $n - 2f$ of these were sent by correct nodes and by Lemma 4.4.3 all of them will reach all correct nodes by $\tau + (2r + 1)\bar{d}$. As a result, all such correct nodes will send $(echo', General, v, \tau, 1)$, which will be received by all correct nodes. Hence, by $\tau + (2r + 2)\bar{d}$ on their timers, all correct nodes will hold $n - f$ distinct $(echo', General, v, \tau, 1)$ messages and will thus accept $(General, v, \tau, 1)$.

Detection of broadcasters: If a correct node q' accepts $(General, v, \tau, 1)$ by time $\tau + 2r\bar{d}$ on its timer, then node q' should have received at least $n - f$ distinct $(echo', General, v, \tau, 1)$ messages, at least $n - 2f$ of which are from correct nodes. Let q be the first correct node to ever send $(echo', General, v, \tau, 1)$. If q sent it as a result of receiving $n - f$ such messages, then q is not the first to send. Therefore, it should have sent it as a result of receiving $n - f$ $(echo, General, v, \tau, 1)$ messages by time $\tau + \bar{d}$. Thus, at least $n - 2f$ such messages were sent by correct nodes by time τ on their timers and would arrive at all correct nodes by time $\tau + \bar{d}$ on their timers. As a result, all will have $General \in broadcasters$.

Uniqueness: Notice that if a correct node sends $(echo', General, v, \tau, 1)$ by time $\tau + \bar{d}$, then no correct node sends $(echo', General, v', 1)$ at any later time. Otherwise, similarly to the arguments in proving the previous property we get that at least $n - f$ nodes sent $(echo, General, v, \tau, 1)$ and $n - f$ nodes sent $(echo, General, v', 1)$. Since $n > 3f$, this implies that at least one correct node sent both $(echo, General, v, \tau, 1)$ and $(echo, General, v', 1)$, and this is not allowed.

Also note that if a correct node accepts $(General, v, \tau, 1)$, then at least one correct node sends $(echo', General, v, \tau, 1)$, which yields the proof of the *Uniqueness* property. □

Nodes stop participating in BYZ_CONSENSUS when they are instructed to do so. They stop participating in the BROADCAST primitive $2\bar{d}$ after they terminate BYZ_CONSENSUS.

DEFINITION 4.4.1

A node **returned** *a value m if it has stopped and returned $value = m$.*

A node p **decides** *if it stops at that timer time and returns a $value \neq \perp$.*

A node p **aborts** *if it stops and returns $\perp$.*

Theorem 4.4.2 *The* BYZ_CONSENSUS *procedure satisfies the Termination property. When $n > 3f$, it also satisfies Agreement, Validity and the two early stopping conditions.*

Proof: We prove the five properties of the theorem. We build up the proof through the following arguments.

Lemma 4.4.1 *If a correct node aborts at time $T + 2r\bar{d}$ on its timer, then no correct node decides at a time $T + 2r'\bar{d} \geq T + 2r\bar{d}$ on its timer.*

Proof: Let p be a correct node that aborts at time $T + 2r\bar{d}$. In this case it should have identified exactly $r - 2$ broadcasters by that time. By the detection of broadcasters property [TPS-4] no correct node will ever accept $(General, v, T, 1)$ and $r - 2$ distinct messages (q_i, v, T, i) for $2 \leq$

$i \leq r-1$, since that would have caused all correct nodes to hold $r-1$ broadcasters by time $T+(2r-1)\bar{d}$ on their timers. Thus, no correct node can decide at local-time $T+2r'\bar{d} \geq T+2r\bar{d}$. □

Lemma 4.4.2 *If a correct node decides by time $T+2r\bar{d}$ on its timer, then every correct node decides by time $T+2(r+1)\bar{d}$ on its timer.*

Proof:
Let p be a correct node that decides by time $T+2r\bar{d}$ on its timer. We consider the following cases:

1. $r=1$: No correct node can abort by time $T+2\bar{d}$, since the inequality will not hold. Node p must have accepted $(General, v, T, 1)$ by $T+2\bar{d}$. By the relay property [TPS-3] all correct nodes will accept $(General, v, T, 1)$ by $T+4\bar{d}$ on their timers. Moreover, p invokes BROADCAST $(p, v, T, 2)$, by which the correctness property [TPS-1] will be accepted by all correct nodes by time $T+4\bar{d}$ on their timers. Thus, all correct nodes will have $value \neq \perp$ and will BROADCAST and stop by time $T+4\bar{d}$ on their timers.

2. $2 \leq r \leq f+1$. Node p must have accepted $(General, v, T, 1)$ and also accepted $r-1$ distinct (q_i, v, T, i) messages for all $i, 2 \leq i \leq r$, by time $T+2r\bar{d}$ on its timer. By Lemma 4.4.1, no correct aborts by that time. By Relay property [TPS-3] each (q_i, v, T, i) message will be accepted by all correct nodes by time $T+(2r+2)\bar{d}$ on their timers. Node p does BROADCAST $(p, v, T, r+1)$ before stopping. By the correctness property, this message will be accepted by all correct nodes by time $T+(2r+2)\bar{d}$ on their timers. Thus, no correct node will abort by $T+(2r+2)\bar{d}$ and all correct nodes will have $value \neq \perp$ and will decide and stop by that time.

3. $r=f+2$. Node p must have accepted (q_i, v, T, i) messages for all $i, 2 \leq i \leq f+2$, by $T+(2f+4)\bar{d}$ on its timer, where the $f+1$ q_i's are distinct. At least one of these $f+1$ nodes, say q_j, must be correct. By the Unforgeability property [TPS-2] q_j, invoked BROADCAST (q_j, v, T, j) by time $T+(2j)\bar{d}$ on its timer, and decided. Since $j \leq f+1$ the above arguments imply that by $T+(2f+4)\bar{d}$ on their timers all correct will decide.

□

Lemma 4.4.2 implies that if a correct node decides at time $T+2r\bar{d}$ on its timer, then no correct node aborts at round $T+2r'\bar{d}$. Lemma 4.4.1 implies the other direction.

Termination: Lemma 4.4.2 implies that if any correct node decides, all decide and stop. Assume that no correct node decides. In this case, no correct node ever invokes a BROADCAST $(q, v, T, \_)$. By detection of broadcasters property [TPS-4], no correct node will ever be considered as broadcaster. Therefore, by time $T+((2f+4)\bar{d}$ on their timers, all correct nodes will have at most f broadcasters and will abort and stop. □

Agreement: If no correct node decides, then all abort, and return to the same value. Otherwise, let p be the first correct node to decide. Therefore, no correct node aborts. The value returned by p is the value v of the accepted $(General, v, 1)$ message. By Properties [TPS-3] and [TPS-5] all

correct nodes accept $(General, v, T, 1)$ and no correct node accepts $(General, v', T, 1)$ for $v \neq v'$. Thus all correct nodes return the same value. □

Validity: Let all the correct nodes begin with the same value v' and invoke the protocol with the same timer time (T). Then, by time $T + \bar{d}$ on their timers, all correct nodes receive at least $n - 2f$ distinct $(echo, General, v', T, 1)$ messages via the CONSENSUS-BROADCAST primitive and send $(echo', General, v', T, 1)$ messages to all. Hence, all nodes receive at least $n - f$ distinct $(echo', General, v', T, 1)$ messages by $T + 2\bar{d}$ on their timers and thus accept $(General, v', T, 1)$. Hence in the BYZ_CONSENSUS procedure all correct nodes set their value to v'. By $T + 2\bar{d}$ on their timers, all correct nodes will stop and return v'. □

Early-stopping: The first early stopping property [ES-1] is directly implied from the proof of the validity property. Correct nodes proceed once they receive messages from $n - f$ nodes, thus it is enough to receive messages from all correct nodes. The proof of the second early stopping property [ES-2] is identical to the proof of the termination property. By time $T + (2f' + 4)\bar{d}$ all will abort unless any correct node invokes BROADCAST by that time on its timer. This implies that by $T + (2f' + 6)\bar{d}$ on their timers all correct nodes will always terminate, if the actual number of faults f' is less than f. □

Thus the proof of the theorem is concluded. □

The BROADCAST Primitive

This section presents the **Broadcast** (and *accept*) primitive that is used by the BYZ_CONSENSUS procedure presented earlier, in Section 4.4. The primitive follows the primitive of of Toueg, Perry, and Srikanth [73], though here it is presented in a real-time model.

In the original synchronous model, nodes advance according to phases. This intuitive lock-step process clarifies the presentation and simplifies the proofs. In this section, the discussion carefully considers the various time consideration and proves that nodes can rush through the protocol and do not to need to wait for a completion of a "phase" in order to move to the next step of the protocol.

Note that when a node invokes the procedure it evaluates all the messages in its buffer that are relevant to the procedure.

The BROADCAST primitive satisfies the four [TPS-*] properties, under the assumption that $n > 3f$. The proofs below follow closely to the original proofs of [73], in order to make it easier for readers that are familiar with the original proofs.

Lemma 4.4.3 *If a correct node p_i sends a message at timer time $T_i \leq \tau + r\bar{d}$ on p_i's timer it will be recieved by each correct node p_j by timer time $\tau + (r + 1)\bar{d}$ on p_j's timer.*

Proof: Assume that node p_i sends a message at real time t with timer time $T_i(t) \leq \tau + r\bar{d}$. Thus, $T_i(t) \leq \tau + r(\bar{\sigma} + d)(1 + \rho)$. It should arrive at every correct timer p_j within $d(1 + \rho)$ on any correct node's timer. Recall that $|T_i(t) - T_j(t)| < \bar{\sigma}(1 + \rho)$. If $T_j \geq T_i$ we are done. Otherwise,

$$T_j(t) \leq T_i(t) + \bar{\sigma}(1 + \rho) \leq \tau + r(\bar{\sigma} + d)(1 + \rho) + \bar{\sigma}(1 + \rho) .$$

By the time (say t') that the message arrives to p_j we get $T_j(t') \leq \tau + r(\bar{\sigma} + d)(1 + \rho) + \bar{\sigma}(1 + \rho) + d(1 + \rho) \leq \tau + (r + 1)\bar{d}$. □

Procedure BROADCAST (p, m, τ, k)

/ executed per such quadruple */*
/ nodes send specific message with the same τ only once */*
/ multiple messages sent by an individual node are ignored */*

node p sends $(init, p, m, \tau, k)$ to all nodes;

by time $(\tau + (2k-1)\bar{d})$:
  if (received $(init, p, m, \tau, k)$ from p **then**
    send $(echo, p, m, \tau, k)$ to all;

by time $(\tau + 2k\bar{d})$:
  if (received $(echo, p, m, \tau, k)$ from $\geq n - 2f$ distinct nodes q **then**
    send $(init', p, m, \tau, k)$ to all;
  if (received $(echo, p, m, \tau, k)$ msgs from $\geq n - f$ distinct nodes **then**
    accept (p, m, τ, k);

by time $(\tau + (2k+1)\bar{d})$:
  if (received $(init', p, m, \tau, k)$ from $\geq n - 2f$ **then**
    $broadcasters := broadcasters \bigcup \{p\}$;
  if (received $(init', p, m, \tau, k)$ from $\geq n - f$ distinct nodes **then**
    send $(echo', p, m, \tau, k)$ to all;

at any time:
  if (received $(echo', p, m, \tau, k)$ from $\geq n - 2f$ distinct nodes **then**
    send $(echo', p, m, \tau, k)$ to all;
  if (received $(echo', p, m, \tau, k)$ from $\geq n - f$ distinct nodes) **then**
    accept (p, m, τ, k);

end

Figure 4.7: BROADCAST primitive

Lemma 4.4.4 *If a correct node ever sends $(echo', p, m, \tau, k)$ then at least one correct node must have sent $(echo', p, m, \tau, k)$ by timer time $\tau + (2k+1)\bar{d}$.*

Proof: Let t be the earliest timer time by which any correct node q sends the message $(echo', p, m, \tau, k)$. If $t > \tau + (2k+1)\bar{d}$, node q should have received $(echo', p, m, \tau, k)$ from $n - 2f$ distinct nodes, at least one of which from a correct node that was sent prior to timer time $\tau + (2k+1)\bar{d}$. □

Lemma 4.4.5 *If a correct node ever sends $(echo', p, m, \tau, k)$ then p's $(init, p, m, \tau, k)$ must have been received by at least one correct node by time $\tau + (2k-1)\bar{d}$.*

Proof: By Lemma 4.4.4, if a correct node ever sends $(echo', p, m, \tau, k)$, then some correct node q should send it by time timer $\tau + (2k+1)\bar{d}$. By the procedure, q have received $(init', p, m, \tau, k)$ from at least $n - f$ nodes by timer time $\tau + (2k+1)\bar{d}$. At least one of them is correct who have received $n - 2f$ $(echo, p, m, \tau, k)$ by timer time $\tau + 2k\bar{d}$. One of which was sent by correct node that should have received $(init, p, m, \tau, k)$ before sending $(echo, p, m, \tau, k)$ by timer time $\tau + (2k-1)\bar{d}$. □

Theorem 4.4.3 *The* BROADCAST *primitive presented in Figure 4.7 satisfies properties [TPS-1] through [TPS-4].*

Proof:

Correctness: Assume that a correct node p sends (p, m, τ, k) by $\tau + (2k-2)\bar{d}$ on its timer. Every correct node receives $(init, p, m, \tau, k)$ and sends $(echo, p, m, \tau, k)$ by $\tau + (2k-1)\bar{d}$ on its timer. Thus, every correct node receives $n-f$ $(echo, p, m, \tau, k)$ from distinct nodes by $\tau+(2k-1)\bar{d}$ on its timer and accepts (p, m, τ, k).

Unforgeability: If no correct node p does a BROADCAST (p, m, τ, k), it does not send $(init, p, m, \tau, k)$, and no correct node will send $(echo, p, m, \tau, k)$ by $\tau + (2k-1)\bar{d}$ on its timer. Thus, no correct node accepts (p, m, τ, k) by $\tau + 2k\bar{d}$ on its timer. If a correct node would have accepted (p, m, τ, k) at a later time it can be only as a result of receiving $n - f$ $(echo', p, m, \tau, k)$ distinct messages, some of which must be from correct nodes. By Lemma 4.4.5, p should have sent $(init, p, m, \tau, k)$, a contradiction.

Relay: Notice that $r \geq k$, thus even if nodes issue an accept at earlier time, the claim holds for the specified times.

The subtle point is when a correct node issues an accept as a result of getting echo messages. If $r = k$ and the correct node, say q, have received $(echo, p, m, \tau, k)$ from $n - f$ nodes by $\tau + 2k\bar{d}$ on its timer. At least $n-2f$ of them were sent by correct nodes. Since every correct node among these has sent its message by $\tau + (2k-1)\bar{d}$, all those messages should have arrived to every correct node by $\tau + 2k\bar{d}$ on its timer. Thus, every correct node should have sent $(init', p, m, \tau, k)$ by $\tau + 2k\bar{d}$ on its timer. As a result, every correct node will receive $n - f$ such messages by $\tau + (2k+1)\bar{d}$ on its timer and will send $(echo', p, m, \tau, k)$ by that time, which will lead all correct nodes to accept (p, m, τ, k) by $\tau + (2r+2)\bar{d}$ on its timer.

Otherwise, the correct node, say q, accepts (p, m, τ, k) by $\tau + 2r\bar{d}$ on its timer as a result of receiving $n - f$ $(echo', p, m, \tau, k)$ by that time. Since $n - f$ of these are from correct nodes, they should arrive at any correct node by $\tau + (2r+1)\bar{d}$ on their timers. As a result, by $\tau + (2r+1)\bar{d}$, all correct nodes send $(echo', p, m, \tau, k)$ and by $\tau + (2r+2)\bar{d}$ on their timers will accept (p, m, τ, k).

Detection of broadcasters: As in the original proof, we first argue the second part. Assume that a correct node q adds node p to $broadcasters$. It should have received $n-2f$ $(init', p, m, \tau, k)$ messages. Thus, at least one correct node has sent $(init', p, m, \tau, k)$ as a result of receiving $n-2f$ $(echo, p, m, \tau, k)$ messages. One of these should be from a correct node that has received the original BROADCAST message of p.

To prove the first part, we consider two similar cases to support the Relay property. If $r = k$ and the correct node, say q, accepts (p, m, τ, k) as a result of receiving $n - f$ $(echo, p, m, \tau, k)$ by $\tau + 2k\bar{d}$ on its timer. At least $n - 2f$ of them were sent by correct nodes. Since every correct node among these has sent its message by $\tau + (2k-1)\bar{d}$, all those messages should have arrived at every correct node by $\tau + 2k\bar{d}$ on its timer. Thus, every correct node should have sent $(init', p, m, \tau, k)$ by $\tau + 2k\bar{d}$ on its timer. Consequently, all correct nodes will receive $n - f$ such messages by time $\tau + (2k+1)\bar{d}$ and will add p to $broadcasters$.

Otherwise, q accepts (p, m, τ, k) as a result of receiving $(echo', p, m, \tau, k)$ from $n - f$ nodes by $\tau + 2r\bar{d}$ (for $r \geq k$) on its timer. By Lemma 4.4.4 a correct node sent $(echo', p, m, \tau, k)$ by $\tau + (2k+1)\bar{d}$. It should have received $n - f$ $(init', p, m, \tau, k)$ messages by that time. All such messages that were sent by correct nodes were sent by $\tau + 2k\bar{d}$ on their timers and should arrive at every correct node by $\tau + (2k+1)\bar{d}$ on its timer. Since there are at least $n - 2f$ such messages, all will add p to $broadcasters$ by $\tau + (2k+1)\bar{d}$ on their timers. □

This completes the proof of Theorem 4.4.2. □

Chapter 5

Self-stabilizing Byzantine Token Circulation using Pulse Synchronization

5.1 Specific Definitions

Basic notations:

- *agreement_duration* represents the maximum real-time required to complete the chosen Byzantine consensus procedure used by the algorithm 5.1. We assume the use of the consensus procedure presented in Section 4.4. We assume $\sigma \leq \sigma + \textit{agreement\_duration} < \textit{cycle} \leq \textit{Cycle} + \textit{agreement\_duration}$.

Basic definitions:

DEFINITION 5.1.1 *The communication network is* **correct** *following* $\Delta_{net} = \textit{pulse\_conv} + \textit{agreement\_duration} + \sigma$ *real-time of continuous non-faulty behavior.*

DEFINITION 5.1.2 *A node is* **correct** *following* $\Delta_{node} = \textit{pulse\_conv} + \textit{agreement\_duration} + \sigma$ *real-time of continuous non-faulty behavior.*

- $\mathbf{token_q(t)} \in \{\mathbf{p_i}\}_{\mathbf{i=0}}^{\mathbf{n-1}}$ is the node holding the token according to node q's view at real-time t.
- The **token_state** of the system at real-time t is given by: $token\_state(t) \equiv (token_{p_0}(t), \ldots, token_{p_{n-1}}(t))$.
- A system is in an **agreed token_state** at real-time t if $\forall\ correct\ p_i, p_j, \quad token_{p_i}(t) = token_{p_j}(t)$.
- Let G be the set of all possible token_states of a system S.
- $s \in G$ is an **eventually-agreed token_state** of the system at real-time t if the system is in an agreed token_state at some real-time t_{agree} in the interval $[t, t + \hat{\sigma}]$, where $\hat{\sigma}$ is some small constant.

In the context of this chapter we achieve $\hat{\sigma} := \sigma$.

The "Self-stabilizing Byzantine Token Circulation Problem" is defined as follows:

DEFINITION 5.1.3 **The Self-stabilizing Token Circulation Problem**

As long as the systems is coherent:

Convergence: *Starting from an arbitrary state, s, the system reaches an eventually-agreed token_state after a finite time.*

Closure: *If s is an eventually-agreed token_state of the system at real-time t_0 then $\forall$ $real\ time\ t \geq t_0$,*

1. *token_state(t) is an eventually-agreed token_state,*
2. *"Fairness": if $t \longrightarrow \infty$ then $\forall i, p_i$ holds the token an infinite number of times.*

The fairness validity requirement is used as a method for defying triviality such as setting $Token := q$ for all value of time t.

5.2 Self-stabilizing Byzantine Token Circulation

The token circulation concept addresses the problem of determining which node holds the token at any given time. The definition of the token circulation problem in our model addresses the issue of loss of synchronization and the potential existence of faulty nodes. Since there is no assumption of synchronized clocks or any external method to synchronize the nodes, the token circulation needs to be driven by the local timers of the nodes and by the messages they exchange. Since there are limits on how close nodes can be synchronized, and there are uncertainties concerning message delivery time, the definition cannot assume instantaneous transfer of the token responsibility.

Self-stabilizing protocols have inherent recurrence or infinite execution in them. This is due to the fact that there is never an agreed t_0 at which all the correct nodes could check the system state concurrently and decide whether to perform a system-wide operation or a reset. This is because the system could enter a brief illegal state just before time t_0 which could render a disagreement on t_0, thus the system would never be able to exit the illegal state. This is especially severe when facing Byzantine faults as otherwise it could be possible for every node to send a "system-state verification" message every time period which could be marked as some t_0. Byzantine nodes can cause such a protocol to continuously reset itself. The recurrence raises the issue of consistency on who holds the token during the time period when one correct node has decided on the new token while the other correct nodes have not yet done so and still hold the value of the previous tokens. At this brief period there is an alleged violation of agreement. This is addressed by the definition of the eventually-agreed token_state. Whereas a non-stabilizing protocol could stay in an agreed token_state forever, stabilizing protocols will always have to address the issue of agreement state change and the consequent short-lived violation of agreement.

Note that when one considers a different type of fault, the consensus primitive can either be used as is, or can be replaced by a similar one that is better optimized to the specific fault model. The early stopping property can be maintained in other schemes, as well.

Intuitively, once a pulse arrives, the node resets all variables, other than the index to the next token holder. It starts a timer and waits some time to make sure that all nodes have received the

pulse. Notice that the pulse synchronization implies that this takes at most $\sigma(1+\rho)$ time units on its clock. The $(1+\rho)$ factor counts for the drift between the local timer readings and real time.

After that pause the nodes invoke BYZ_CONSENSUSto agree on the index of the token holder in the following round. When the system is coherent, this results in the index each correct node expects. If the resulting index differs from what a node expects it will adjust its index. Note that the BYZ_CONSENSUSprocedure may return $\perp$. We identify that value with the default value, say p_0.

When the system is coherent, the BYZ_CONSENSUScompletes after two rounds of information exchange among the correct nodes, which may be completed extremely fast as the actual time it takes for messages to travel may be much faster than the expected upper bound d. The protocol presents an option to invoke the next pulse after a period of time when the current node holds the token, resulting in a pretty fast token exchange. Otherwise, the token holder changes every *cycle*.

```
Algorithm SS-BYZ-TOKEN
at "pulse" event          /* received the internal pulse event */
begin
 1. Token := p_next;  /* next is the index of the next token holder*/
 2. Revoke possible other instances of SS-BYZ-TOKEN and
      clear all data structures besides Token and p_next;
 3. Wait until σ(1+ρ) time units have elapsed since pulse;
 4. p_next := BYZ_CONSENSUS (p_((next+1) mod n), σ);
 5. Token := p_((next-1) mod n);                  /* posterior adj. */
 6. Option: Wait a predefined time and invoke the next pulse.
end
```

Figure 5.1: The self stabilizing token circulation protocol

Theorem 5.2.1 SS-BYZ-TOKEN *solves the "Self-stabilizing Byzantine Token Circulation Problem".*

Proof: Convergence: Let the system be coherent but in an arbitrary state s, with the nodes holding arbitrary tokens. Consider the first correct node that completed Line 3 of the SS-BYZ-TOKEN algorithm. Since the system is coherent, all correct nodes invoked the preceding *pulse* within σ of each other. At the last pulse all remnants of previously invoked instances of BYZ_CONSENSUS were flushed by all the correct nodes. A correct node does not initiate or join BYZ_CONSENSUS before waiting $\sigma(1+\rho)$ time units subsequent to the pulse, hence not before all correct nodes have invoked a pulse and subsequently flushed their buffers. Thus all correct nodes will eventually join BYZ_CONSENSUS which thus will terminate successfully. Following termination of BYZ_CONSENSUS by which all correct nodes have decided on the same value, the token is adjusted again in order to get immediate token agreement (instead of waiting until the next pulse invocation). This has an effect only if the correct nodes initiated BYZ_CONSENSUS with different token values (can only happen following system initialization or following a catastrophic state). Hence, subsequent to Line 5 all correct nodes hold the same $Token$ and thus the system is in an agreed token_state. Note that by the early stopping property [ES-2] of BYZ_CONSENSUS, the time complexity to reach consensus is that of the actual number of faults.

Closure: If $\hat{\sigma} = \sigma$, then when the first node to execute a pulse sets its new token holder, there is a time period, bounded by σ, until the last node to execute its pulse will also set its token to the same

value. Therefore, for up to σ real-time units, the system will not be in agreed token_state but rather be in an eventually-agreed token_state, following which it will return to an agreed token_state. Note that when all correct nodes enter the BYZ_CONSENSUSalgorithm with identical values then the posterior adjustment makes no change. The faulty nodes cannot affect the choice of the next token holder and, therefore, the fairness also holds. Note that by the early stopping property [ES-1] of BYZ_CONSENSUS, the overhead is only two communication rounds.

□

Complexity Analysis of SS-BYZ-TOKEN

The time complexity is the sum of the convergence time of the pulse synchronization procedure and the time complexity of the BYZ_CONSENSUSprocedure. The pulse tightness σ, is of the order of $3d$ and $\bar{d}$ is of order $4d$. Therefore, the convergence time is always bounded by $O(f') \cdot cycle + 3d + 3(2f'+6)d$ from any arbitrary state, where $f' \leq f$ is the actual number of faulty nodes. The $O(f')$ is contributed by the convergence of the pulse synchronization module. The accuracy, or the time at which correct nodes are not in an agreed token_state, equals $\sigma(= 3d)$. The BYZ_CONSENSUSprocedure has the two early-stopping features presented above. Thus, the time complexity when the network performs correctly is less than $2d$, the upper bound on the time it takes to have two rounds of information exchange among the correct nodes. The message complexity is $O(nf^2)$. Token circulation/leader election algorithms have a proven message complexity, lower bounds, of $\Omega(nlogn)$ and typically have a time complexity of $o(n)$. Self-stabilizing protocols usually have a higher convergence time. SS-BYZ-TOKEN is of comparable complexity to non-stabilizing or non-fault tolerant token circulation/leader election algorithms while being much more robust.

5.3 An Extended Scheme for General Resource Allocation

The presented scheme can be extended to solve a general resource allocation problem in an efficient way that is self-stabilizing and resilient to permanent Byzantine faults.

Let there be k different types of resources; we assume they are numbered and totally ordered. Every process wants to have access to each one of the k resources, one resource at a time. Processes cannot allocate the same type of resource at the same time. Thus there is a finite discrete number of legal states, $\mathcal{L} \subset \mathcal{S}$, where $\mathcal{S}$ is the set of all system states, in which there is exactly one process that is allocated to each one of the resource and the process has no additional resource allocated to it at the same time. We now need to ensure fairness so that in every infinite execution every process gets every type of resource allocated infinitely often.

Let $R_i(j)$ denote that resource i is allocated to node p_j. We define the set of legal allocation states $\mathcal{LA} \subset \mathcal{L}$, as follows:

$$\mathcal{LA} = \{s \in \mathcal{L} | \exists j \in 1..n \; s.t. \; R_1(j), ..., R_k(j+k) \pmod n\}.$$

Thus, informally, the set of legal states are all system states in which the k consecutive resources are allocated to k consecutive processes. We associate each legal allocation state with some unique integer. We number the legal allocation states according to the index of the process that allocates the resource that is numbered as "1". Every node holds a variable number corresponding to its current proposition on the legal allocation state. In every pulse the "next" legal

allocation state is imposed. The table of legal allocation states is precalculated and stored at every process in non-volatile memory.

The algorithm becomes essentially identical to SS-BYZ-TOKEN though instead of agreeing on the index of the next token holder, the agreement is on the index of the next legal allocation state. The Closure and Convergence follows from the proof of SS-BYZ-TOKEN . The fairness follows trivially from the legal allocation state transition rule.

The same scheme also solves the graph coloring problem, and thus the dining philosophers, which are two of the few distributed problems that have self-stabilizing Byzantine solutions (see [65,64,60]). In these problems it is required to alternate between different colorings of the network graph. The dining philosophers problem is a particular case that can alternate between two legal states.

Chapter 6

Self-stabilizing Byzantine Agreement without using Pulse Synchronization

6.1 Specific Definitions

DEFINITION 6.1.1 *The communication network is* **correct** *following* $\Delta_{net} \geq d$ *real-time of continuous non-faulty behavior.*

DEFINITION 6.1.2 *A node is* **correct** *following* $\Delta_{node} \geq 14(2f+3)d+10d$ *real-time of continuous non-faulty behavior during a period that the communication network is correct.*

It is assumed that each node has a local timer that proceeds at the rate of real-time. The actual reading of the various timers may be arbitrarily apart, but their relative rate is bounded in our model. To simplify the presentation we will ignore the drift factor of hardware clocks. Since nodes measure only periods of time that span several d, we will assume that d is an upper bound on the sending time of messages among correct nodes, measured by each local timer. To distinguish between a real-time value and a node's local-time reading we use t for the former and τ for the latter. The function $rt(\tau_p)$ represents the real-time when the timer of p reads τ_p at the current execution.

6.2 The SS-BYZ-AGREE algorithm

We consider the Byzantine agreement problem in which a *General* broadcasts a value and the correct nodes agree on the value broadcasted. In our model any node can be a General. An instance of the protocol is executed per General, and a correct General is expected to send one value at a time[1]. The target is for the correct nodes to associate a local-time with the protocol initiation by the General and to agree on a specific value associated with that initiation, if they agree that such an initiation actually took place. We bound the frequency by which correct Generals may initiate agreements, though Byzantine nodes might trigger agreements on their values as frequent as they wish.

[1] One can expand the protocol to a number of concurrent invocations by using an index to differentiate among the concurrent invocations.

```
Protocol SS-BYZ-AGREE on (G, m)            /* Executed at node q. τ_q is the
local-time at q. */
                    /* Block Q is executed only when (and if) invoked. */
 /* Each block is executed at most once, when the precondition holds. */
 /* Executed as a result of I-accept  at Line R1, or when τ_q^G is defined.
 */
     /* Invoked at node q upon arrival of a message (Initiator, G, m) from
node G. */

Q.  INITIATOR-ACCEPT(G, m).   /* determines τ_q^G and a value m' for node G */

R1.  if I-accept ⟨G, m', τ_q^G⟩ and τ_q − τ_q^G ≤ 4d then
R2.       value := ⟨G, m'⟩;
R3.       MSGD-BROADCAST(q, value, 1);
R4.       stop and return ⟨value, τ_q^G⟩.

S1.   if by τ_q (where τ_q ≤ τ_q^G + (2r+1)·Φ)
        accepted r distinct messages (p_i, ⟨G, m''⟩, τ_i, i)
        where ∀i, j  1 ≤ i ≤ r, and p_i ≠ p_j ≠ G then
S2.       value := ⟨G, m''⟩;
S3.       MSGD-BROADCAST(q, value, r + 1);
S4.       stop and return ⟨value, τ_q^G⟩.

T1.   if by τ_q (where τ_q > τ_q^G + (2r+1)·Φ)  |broadcasters| < r − 1 then
T2.       stop and return ⟨⊥, τ_q^G⟩.

U1.   if τ_q > τ_q^G + (2f+3)·Φ then
U2.       stop and return ⟨⊥, τ_q^G⟩.

cleanup:
  3d after returning a value reset the related INITIATOR-ACCEPT and
MSGD-BROADCAST;
  Remove any value or message older than (2f + 3)·Φ + 3d time units.
```

Figure 6.1: The SS-BYZ-AGREE protocol

The General initiates agreement by disseminating a message $(Initiator, G, m)$ to all nodes. Upon receiving the General's message, each node invokes the SS-BYZ-AGREE protocol, which in turn invokes the INITIATOR-ACCEPT primitive. Alternatively, if a correct node concludes that enough nodes have invoked the protocol (or the primitive) it will participate by executing the various parts of the INITIATOR-ACCEPT primitive, but will not invoke it. If all correct nodes invoke the protocol within a "small" time-window, as will happen if the General is a correct node, then it is ensured that the correct nodes agree on a value for the General. If all correct nodes do not invoke the SS-BYZ-AGREE protocol within a small time-window, as can happen if the General is faulty, then if any correct node accepts a non-null value, all correct nodes will accept and agree on that value.

For ease of following the arguments and the logic of our SS-BYZ-AGREE protocol, we chose to follow the building-block structure of [73]. The equivalent of the broadcast primitive that simulates authentication in [73] is the primitive MSGD-BROADCAST presented in Section 6.4. The main differences between the original synchronous broadcast primitive and MSGD-BROADCAST are two-folds: first, the latter executes rounds that are anchored at some agreed event whose local-time is supplied to the primitive through a parameter; second, the conditions to be satisfied at each round at the latter, need to be satisfied by some time span that is a function of the round number and need

not be executed only during the round itself. This allows nodes to rush through the protocol in the typical case when messages happen to be delivered faster than the worse case round span.

The SS-BYZ-AGREE protocol needs to take into consideration that correct nodes may invoke the agreement procedure at arbitrary times and with no knowledge as to when other correct nodes may have invoked the procedure. A mechanism is thus needed to make all correct nodes attain some common notion as to when and what value the General has sent. The differences of the real-time representations of the different nodes' estimations should be bounded. This mechanism is satisfied by the INITIATOR-ACCEPT primitive defined in Section 6.3.

We use the following notations in the description of the agreement procedure:

- Let Φ be the duration of time equal to $(\tau^G_{skew} + 2d)$ local-time units on a correct node's timer, where $\tau^G_{skew} = 5d$ in the context of this result. Intuitively, Φ is the duration of a "phase" on a correct node's timer.
- Δ will be equal to $(2f + 3) \cdot \Phi$, the upper bound on the time it takes to run the agreement protocol.
- $\perp$ denotes a null value.
- In the INITIATOR-ACCEPT primitive:
  - A I-*accept*[2] is issued on values sent by G.
  - τ^G_q denotes the local-time estimate, at node q, as to when the General have sent a value that has been I-accept in INITIATOR-ACCEPT by node q.

In the context of this chapter we assume that a correct node will not initiate agreement on a new value at least $6d$ time units subsequent to termination of its previous agreement.

DEFINITION 6.2.1 *We say:*

A node p ***decides*** *at time τ if it stops at that local-time and returns $value \neq \perp$.*

A node p ***aborts*** *if it stops and returns $\perp$.*

A node p ***returns*** *a value if it either aborts or decides.*

The SS-BYZ-AGREE protocol is presented (see Figure 6.1) in a somewhat different style than the original protocol in [73]. Each round has a precondition associated with it: if the local timer value associated with the initialization by the General is defined and the precondition holds then the step is to be executed. It is assumed that the primitives instances invoked as a result of the SS-BYZ-AGREE protocol are implicitly associated with the agreement instance that invoked them. A node stops participating in the procedures and the invoked primitives $3d$ time units after it returns a value.

The SS-BYZ-AGREE protocol satisfies the following typical properties:
Agreement: The protocol returns the same value ($\neq\perp$) at all correct nodes;
Validity: If all correct nodes are triggered to invoke the SS-BYZ-AGREE protocol by a value sent by a correct General G, then the all correct nodes return that value;

[2] An *accept* is issued within MSGD-BROADCAST.

Termination: The protocol terminates in a finite time.

It also satisfies the following properties:

Timeliness:

1. (agreement) For every two correct nodes q and q' that decide on (G, m) at τ_q and $\tau_{q'}$, respectively:
   (a) $|rt(\tau_q) - rt(\tau_{q'})| \leq 3d$, and if validity holds, then $|rt(\tau_q) - rt(\tau_{q'})| \leq 2d$.
   (b) $|rt(\tau_q^G) - rt(\tau_{q'}^G)| \leq 5d$.
   (c) $rt(\tau_q^G), rt(\tau_{q'}^G) \in [t_1 - 2d, t_2]$, where $[t_1, t_2]$ is the interval within which all correct nodes that actually invoked the SS-BYZ-AGREE (G, m) did so.
   (d) $rt(\tau_q^G) \leq rt(\tau_q)$ and $rt(\tau_q) - rt(\tau_q^G) \leq \Delta$ for every correct node q.
2. (validity) If all correct nodes invoked the protocol in an interval $[t_0, t_0 + d]$, as a result of some value m sent by a correct General G that spaced the sending by at least $6d$ from the completion of the last agreement on its value, then for every correct node q, the decision time τ_q, satisfies $t_0 - d \leq rt(\tau_q^G) \leq rt(\tau_q) \leq t_0 + 3d$.
3. (termination) The protocol terminates within Δ time units of invocation, and within $\Delta + 7d$ in case it was not invoked explicitly.
4. (separation) Let q be any correct node that decided on any two agreements regarding p, then $t_2 + 5d < \bar{t}_1$ and $rt(\tau_q) + 5d < \bar{t}_1 < rt(\bar{\tau}_q)$, where t_2 is the latest time at which a correct node invoked SS-BYZ-AGREE in the earlier agreement and $\bar{t}_1$ is the earliest SS-BYZ-AGREE invoked by a correct node in the later agreement.

Note that the bounds in the above property is with respect to d, the bound on message transmission time among correct nodes and not the worse case deviation represented by Φ.

Observe that since there is no prior notion of the possibility that a value may be sent, it might be that some nodes associate a $\perp$ with a faulty sending and others may not notice the sending at all.

The proof that the SS-BYZ-AGREE protocol meets its properties appears in Section 6.5.

6.3 The INITIATOR-ACCEPT Primitive

In the protocol in [73] a General that wants to send some value broadcasts it in a specific round (round 0 of the protocol). From the various assumptions on synchrony all correct nodes can check whether a value was indeed sent at the specified round and whether multiple (faulty) values were sent. In the transient fault model no such round number can be automatically adjoined with the broadcast. Thus a faulty General has more power in trying to fool the correct nodes by sending its values at completely different times to whichever nodes it decides.

The INITIATOR-ACCEPT primitive aims at making the correct nodes associate a relative time to the invocation of the protocol by (the possibly faulty) General, and to converge to a single candidate value for the agreement to come. Since the full invocation of the protocol by a faulty

```
Primitive INITIATOR-ACCEPT(G, m)
                          /* Executed at node q. τ_q is the local-time at q. */
                /* Lines L1 and L2 are repeatedly executed until I-accept. */
      /* The rest are executed at most once, when the precondition holds. */
       /* Block K is executed only when (and if) the primitive is explicitly
invoked. */

K1. if τ_q − last_τ_q > 7d and if at τ_q − d initiator[G, _] =⊥ then     /* allow recent
entries */
K2.   send (support, G, m) to all;                          /* for a single m ever */
K3.   set initiator[G, m] := τ_q − d;                            /* recording time */

L1. if received (support, G, m) from ≥ n − 2f distinct nodes
       within a window of α ≤ 4d time units of each other then
L2.   initiator[G, m] := max[initiator[G, m], (τ_q − α − 2d)];     /* recording time */
L3. if received (support, G, m) from ≥ n − f distinct nodes
       within a window of 2d time units of each other then
L4.   send (ready, G, m) to all;

M1. if received (ready, G, m) from ≥ n − 2f distinct nodes then
M2.   send (ready, G, m) to all;
M3. if received (ready, G, m) from ≥ n − f distinct nodes then
M4.   τ_q^G := initiator[G, m]; I-accept ⟨G, m, τ_q^G⟩; last_τ_q := τ_q;

cleanup:
  Remove any value or message older than Δ + 7d time units.
  If last_τ_q > τ_q then last_τ_q :=⊥ .
```

Figure 6.2: The INITIATOR-ACCEPT primitive that yields a common notion of protocol invocation

General might be questionable, there may be cases in which some correct nodes will return a $\perp$ value and others will not identify the invocation as valid. If any correct node returns a value $\neq\perp$, all will return the same value.

Each correct node records the local-time at which it first received messages associated with the invocation of the protocol and produces an estimate to its (relative) local-time at which the protocol may have been invoked. The primitive guarantees that all correct nodes' estimates are within bounded real-time of each other.

We say that a node does an **I-accept** of a value sent by the General if it accepts this value as the General's initial value, and τ_q^G is the estimated local-time at q associated with the invocation of the protocol by the General.

The nodes maintain a vector *initiator*$[G, \_]$ for the possible values sent by the General G, where each non-empty entry is a local-time associated with the entry value. We will consider the data structures of a node *fresh* if up to d units of time ago *initiator*$[G, \_]$ did not contain any value and *latest_accept* was $\perp$.

Nodes decay old messages and reset the data structures shortly after completion of the primitive, as defined below. It is assumed that correct nodes will not invoke the INITIATOR-ACCEPT primitive when the data structures are not fresh.

The INITIATOR-ACCEPT primitive satisfies the following properties:

IA-1 *(Correctness)* If all correct nodes invoke INITIATOR-ACCEPT (G, m), with *fresh* data structures, within some real-time interval $[t_0, t_0 + d]$, then:

1A All correct nodes I-accept $\langle G, m, \tau^G \rangle$ within $2d$ time units of the time the last correct node invokes the primitive INITIATOR-ACCEPT(G, m).

1B All correct nodes I-accept $\langle G, m, \tau^G \rangle$ within $2d$ time units of each other.

1C For every pair of correct nodes q and q' that I-accepts $\langle G, m, \tau_q^G \rangle$ and $\langle G, m, \tau_{q'}^G \rangle$, respectively: $|rt(\tau_{q'}^G) - rt(\tau_q^G)| \leq d$.

1D For each correct node q that I-accepts $\langle G, m, \tau_q^G \rangle$ at τ_q, $t_0 - d \leq rt(\tau_q^G) \leq rt(\tau_q) \leq t_0 + 3d$.

IA-2 *(Unforgeability)* If no correct node invokes INITIATOR-ACCEPT (G, m), then no correct node I-accepts $\langle G, m, \tau^G \rangle$.

IA-3 *(Δ-Relay)* If a correct node q I-accepts $\langle G, m, \tau_q^G \rangle$ at real-time t, such that $0 \leq t - rt(\tau_q^G) \leq \Delta$, then:

3A Every correct node q' I-accepts $\langle G, m, \tau_{q'}^G \rangle$, at some real-time t', with $|t - t'| \leq 2d$ and $|rt(\tau_q^G) - rt(\tau_{q'}^G)| \leq 5d$.

3B Moreover, $rt(\tau_q^G), rt(\tau_{q'}^G) \in [t_1 - 2d, t_2]$, where $[t_1, t_2]$ is the interval within which all correct nodes that actually invoked SS-BYZ-AGREE (G, m) did so.

3C For every correct node q', $rt(\tau_{q'}^G) \leq rt(\tau_{q'})$ and $rt(\tau_{q'}) - rt(\tau_{q'}^G) \leq \Delta + 7d$.

IA-4 *(Uniqueness)* If a correct node q I-accepts $\langle G, m, \tau_q^G \rangle$, then no correct node I-accepts $\langle G, m', \tau_p^G \rangle$ for $m \neq m'$, for $|rt(\tau_q^G) - rt(\tau_p^G)| \leq 5d$.

Each node maintains in addition to *initiator*$[G, \_]$ a data structure in which the latest message from each partner regarding a possible value sent by the General is kept. The data structure records as a time stamp the local-time at which each message is received. If the data structure contains illegal values or future time stamps (due to transient faults) the messages are removed. The protocol also requires the knowledge of the state of the vector *initiator*$[G, \_]$ d time units in the past. It is assumed that the data structure reflects that information.

When the primitive is explicitly invoked the node executes block K. A node may receive messages related to the primitive, even in case that it did not explicitly invoke the primitive. In this case it executes the rest of the blocks of the primitive, if the appropriate preconditions hold. A correct node repeatedly executes Line L1 and Line L2, whenever the precondition holds. The rest are executed at most once, when the precondition holds for the first time.

Following the completion of SS-BYZ-AGREE, the data structures of the related INITIATOR-ACCEPT instance are reset.

The proof that the INITIATOR-ACCEPT primitive satisfies the [IA-*] properties, under the assumption that $n > 3f$, appears in Section 6.5.

6.4 The MSGD-BROADCAST Primitive

This section presents the MSGD-BROADCAST (a message driven broadcast) primitive, which **accept**s messages being **broadcast**ed by executing it. The primitive is invoked by the SS-BYZ-AGREE

```
Primitive MSGD-BROADCAST(p, m, k)
                                    /* Executed per such triplet at node q. */
                                /* Nodes send specific messages only once. */
                        /* Nodes execute the blocks only when τ^G is defined. */
                  /* Nodes log messages until they are able to process them. */
              /* Multiple messages sent by an individual node are ignored. */

    At node q = p:        /* if node q is node p that invoked the primitive */
V.    node p sends (init, p, m, k) to all nodes;

W1. At time τ_q : τ_q ≤ τ_q^G + 2k·Φ
W2.   if received (init, p, m, k) from p then
W3.       send (echo, p, m, k) to all;

X1. At time τ_q : τ_q ≤ τ_q^G + (2k − 1)·Φ
X2.   if received (echo, p, m, k) from ≥ n − 2f distinct nodes then
X3.       send (init', p, m, k) to all;
X4.   if received (echo, p, m, k) messages from ≥ n − f distinct nodes then
X5.       accept (p, m, k);

Y1. At time τ_q : τ_q ≤ τ_q^G + (2k + 2)·Φ
Y2.   if received (init', p, m, k) from ≥ n − 2f then
Y3.       broadcasters := broadcasters ∪ {p};
Y4.   if received (init', p, m, k) from ≥ n − f distinct nodes then
Y5.       send (echo', p, m, k) to all;

Z1. At any time:
Z2.   if received (echo', p, m, k) from ≥ n − 2f distinct nodes then
Z3.       send (echo', p, m, k) to all;
Z4.   if received (echo', p, m, k) from ≥ n − f distinct nodes then
Z5.       accept (p, m, k);                              /* accept only once */

cleanup:
  Remove any value or message older than (2f + 3)·Φ time units.
```

Figure 6.3: The MSGD-BROADCAST primitive with message-driven round structure

protocol presented in Section 6.2. The primitive follows the broadcast primitive of Toueg, Perry, and Srikanth [73]. In the original synchronous model, nodes advance according to rounds that are divided into phases. This intuitive lock-step process clarifies the presentation and simplifies the proofs. The primitive MSGD-BROADCAST is presented without any explicit or implicit reference to time, rather an anchor to the potential initialization point of the protocol is passed as a parameter by the calling procedure. The properties of the INITIATOR-ACCEPT primitive guarantee a bound between the real-time of the anchors of the correct nodes. Thus a general notion of a common round structure can be implemented by measuring the elapsed time units since the local-time represented by the passed anchor.

In the broadcast primitive of [73] messages associated with a certain round must be sent by correct nodes at that round and will be received, the latest, at the end of that round by all correct nodes. In MSGD-BROADCAST, on the other hand, the rounds progress with the arrival of the anticipated messages. Thus for example, if a node receives some required messages before the end of the round it may send next round's messages. The length of a round only imposes an upper bound on the acceptance criteria. Thus the protocol can progress at the speed of message delivery, which may be significantly faster than that of the protocol in [73].

Note that when a node invokes the primitive it evaluates all the messages in its buffer that are relevant to the primitive. The MSGD-BROADCAST primitive is executed in the context of some initiator G that invoked SS-BYZ-AGREE, which makes use of the MSGD-BROADCAST primitive. No correct node will execute the MSGD-BROADCAST primitive without first producing the reference (anchor), τ^G, on its local timer to the time estimate at which G supposedly invoked the original agreement. By IA-3A this happens within $2d$ of the other correct nodes.

The synchronous Reliable Broadcast procedure of [73] assumes a round model in which within each phase all message exchange among correct nodes take place. The equivalent notion of a round in our context will be Φ defined to be: $\Phi := t^G_{skew} + 2d$.

The MSGD-BROADCAST primitive satisfies the following [TPS-*] properties of Toueg, Perry and Srikanth [73], which are phrased in our system model.

TPS-1 (*Correctness*) If a correct node p MSGD-BROADCAST(p, m, k) at τ_p, $\tau_p \leq \tau_p^G + (2k-1)\cdot\Phi$, on its timer, then each correct node q accepts (p, m, k) at some τ_q, $\tau_q \leq \tau_q^G + (2k+1)\cdot\Phi$, on its timer and $|rt(\tau_p) - rt(\tau_q)| \leq 3d$.

TPS-2 (*Unforgeability*) If a correct node p does not MSGD-BROADCAST(p, m, k), then no correct node accepts (p, m, k).

TPS-3 (*Relay*) If a correct node q_1 accepts (p, m, k) at τ_1, $\tau_1 \leq \tau_1^G + r\cdot\Phi$ on its timer then any other correct node q_2 accepts (p, m, k) at some τ_2, $\tau_2 \leq \tau_2^G + (r+2)\cdot\Phi$, on its timer.

TPS-4 (*Detection of broadcasters*) If a correct node accepts (p, m, k) then every correct node q has $p \in broadcasters$ at some τ_q, $\tau_q \leq \tau_q^G + (2k+2)\cdot\Phi$, on its timer. Furthermore, if a correct node p does not MSGD-BROADCAST any message, then a correct node can never have $p \in broadcasters$.

Note that the bounds in [TPS-1] are with respect to d, the bound on message transmission time among correct nodes.

The MSGD-BROADCAST primitive satisfies the [TPS-*] properties, under the assumption that $n > 3f$. The proofs that appear in Section 6.5 follow closely the original proofs of [73], in order to make it easier for readers that are familiar with the original proofs.

6.5 Proofs

Proof of the INITIATOR-ACCEPT Properties

Theorem 6.5.1 *The* INITIATOR-ACCEPT *primitive presented in Figure 6.2 satisfies properties [IA-1] through [IA-4], assuming that a correct node that invokes the primitive invokes it with fresh data structures.*

Proof:
Correctness: Assume that within d of each other all correct nodes invoke INITIATOR-ACCEPT (G, m). Let t_1 be the real-time at which the first correct node invokes the INITIATOR-ACCEPT and t_2 be the time the last one did so. Since all data structures are *fresh*, then no value $\{G, m'\}$ appeared in

broadcasters d time units before that, thus Line K1 will hold for all correct nodes. Therefore, every correct node sends $(support, G, m)$. Each such message reaches all other correct nodes within d. Thus, between t_1 and $t_2 + d$ every correct node receives $(support, G, m)$ from $n - f$ distinct nodes and sends $(ready, G, m)$ and by $t_2 + 2d$ I-accepts $\langle G, m, \tau' \rangle$, for some τ', thus, proving [IA-1A].

To prove [IA-1B], let q be the first to I-accept after executing Line M4. Within d all correct nodes will execute Line M2, and within $2d$ all will I-accept.

Note that for every pair of correct nodes q and q', the associated initial recording times τ and τ' satisfy $|\tau - \tau'| \leq d$. Line K3 implies that the recording times of correct nodes can not be earlier than $t_1 - d$. Some correct node may see $n - 2f$, with the help of faulty nodes as late as $t_2 + 2d$. All such windows should contain a $support$ from a correct node, so should include real-time $t_2 + d$, resulting in a recording time of $t_2 - d$. Recall that $t_2 \leq t_1 + d$, proving [IA-1C].

To prove [IA-1D] notice that the fastest node may set τ' to be $t_1 - d$, but may I-accept only by $t_2 + 2d \leq t_1 + 3d$.

Unforgeability:
If no correct node invokes INITIATOR-ACCEPT and will not send $(support, G, m)$, then no correct node will ever execute L4 and will not send $(ready, G, m)$. Thus, no correct node can accumulate $n - f$ $(ready, G, m)$ messages and therefore will not I-accept $\langle G, m \rangle$.

Δ-Relay:
Let q be a correct node that I-accepts $\langle G, m, \tau_q^G \rangle$ at real-time t, such that $0 \leq t - rt(\tau_q^G) \leq \Delta$. It did so as a result of executing Line M4. Let X be the set of correct nodes whose $(ready, G, m)$ were used by q in executing Line M4. Either there exists in X a correct node sending it as a result of executing Line L4, or at least one of the nodes in X have heard from such a node (otherwise, it heard from other $f + 1$ distinct nodes and there will be at least $n - f + f + 1 > n$ distinct nodes in total, a contradiction).

Let $\bar{q}$ be the first correct node to execute Line L4, and assume it took place at some real-time t''. Note that $t'' \leq t$. Node $\bar{q}$ collected $n - f$ $support$ messages, with at least $n - 2f$ from correct nodes[3]. Let t_1 be the time at which the $(n - 2f)^{\text{th}}$ $support$ message sent by a correct node was received. Since $\bar{q}$ executed Line L4, all these messages should have been received in the interval $[t_1 - 2d, t_1]$. Node $\bar{q}$ should have set a recording time $\tau \geq t_1 - 4d$ as a result of (maybe repeating) the execution of Line L2.

Every other correct node should have received this set of $(n - 2f)$ $support$ messages sent by correct nodes in the interval $[t_1 - 3d, t_1 + d]$ and should have set the recording time after (maybe repeatedly) executing Line L2, since this window satisfies the precondition of Line L1. Thus, eventually all recording times are $\geq t_1 - 5d$.

Some correct node may send a $support$ message, by executing Line K2, at most d time units later (just before receiving these $n-2f$ messages). This can not take place later than t_1+d, resulting in a recording time of t_1, though earlier than its time of sending the $support$ message. This $support$ message (with the possible help of faulty nodes) can cause some correct node to execute Line L2 at some later time. The window within which the $support$ messages at that node are collected should include the real-time $t_1 + 2d$, the latest time any $support$ from any correct node could have been

[3]Ignore for a moment decaying of messages (we will prove below that no correct node decays these messages at that stage).

received. Any such execution will result in a recording time that is $\leq t_1 + 2d - 2d = t_1$. Thus the range of recording times for all correct nodes (including q) are $[t_1 - 5d, t_1]$. Proving the second part of [IA-3A].

Since we assumed that $0 \leq t - rt(\tau_q^G) \leq \Delta$, all messages above are within the decaying window of all correct nodes and none of these messages will be decayed, proving that the result holds. For the same reason, all correct messages collected by q will not be decayed by other correct nodes. By time $t + d$ all correct nodes will be able to execute Line M2, and by $t + 2d$ each correct node q' will execute Line M4 to I-accept $\langle G, m, \tau_{\bar{q}} \rangle$. Proving the first part of [IA-3A].

To prove [IA-3B] notice that any range of values in Line L2 includes a $support$ of a correct node. The resulting recording time will never be later than the sending time of the $support$ message by that correct node, and thus by some correct node. To prove the second part of [IA-3B] consider again node $\bar{q}$ from the proof of the $\Delta - relay$ property. It collected $n - 2f$ $support$ messages from correct nodes in some interval $[\bar{t}_1, t_1]$, where $\bar{t}_1 \geq t_1 - 2d$. These messages, when received by any correct node will be within an interval of $4d$, with the first message in it from a correct node. These messages will trigger a possible update of the recording time in Line L2. Thus, the resulting recording time of any correct node cannot be earlier than some $2d$ of receiving a $support$ message from a correct node, thus not earlier than $2d$ of sending such a message.

The first part of [IA-3C] is immediate from Line L2 and Line K3. For the second part observe that for every other correct node q', $rt(\tau_{q'}) \leq rt(\tau_q) + 2d$ and $rt(\tau_{q'}^G) \geq rt(\tau_q^G) - 5d$. Thus, $rt(\tau_{q'}) - rt(\tau_{q'}^G) \leq rt(\tau_q) - rt(\tau_q^G) + 7d \leq \Delta + 7d$.

To prove [IA-4] observe that each node sent a $support$ for a single m. In order to I-accept, some correct node needs to send $ready$ after receiving $n - f$ $support$ messages. That can happen for at most a single value of m. What is left to prove, is that future invocations of the primitive will not violate [IA-4]. Observe that by [IA-3B], once a correct node sends a $ready$ message, all recording times are within $2d$ of the reception time of some $support$ message from a correct node. Moreover, by [IA-3C], this is always prior to the current time at any node that sets the recording time. Let q be the latest correct node to I-accept, at some time τ_q on its clock with some $last\_\tau_q$ as the returned recording time. Let p be the first correct node to send a $support$ following that, at some local-time $\bar{\tau}_p$. We will denote by τ timings in the former invocation and by $\bar{\tau}$ timings in the later one.

Observe that $\tau_q^G \leq last\_\tau_q$ and that $rt(last\_\tau_q) - 2d \leq rt(last\_\tau_p)$. When p sends its $support$, $rt(\bar{\tau}_p) - rt(last\_\tau_p) > 9d$ implying that $rt(\bar{\tau}_p) - rt(last\_\tau_q) > 7d$. Once any correct node will send $ready$ in the later invocation the resulting recording time of all correct nodes, including q will satisfy $rt(\bar{\tau}_q^G) \geq rt(\bar{\tau}_p) - 2d$, which implies $rt(\tau_q^G) \leq rt(last\_\tau_q) < rt(\bar{\tau}_p) - 7d \leq rt(\bar{\tau}_q^G) - 5d$. □

Proof of the MSGD-BROADCAST Properties

For brevity we do not present all the proofs. The proofs essentially follow the arguments in the original paper [73] though that paper is not self-stabilizing.

Lemma 6.5.1 *If a correct node p_i sends a message at local-time τ_i, $\tau_i \leq \tau_i^G + r \cdot \Phi$ on p_i's timer it will be received and processed by each correct node p_j at some local-time τ_j, $\tau_j \leq \tau_j^G + (r+1) \cdot \Phi$, on p_j's timer.*

Proof: Assume that node p_i sends a message at real-time t with local-time $\tau_i \leq \tau_i^G + r \cdot \Phi$. Thus, $\tau_i \leq \tau_i^G + r(t_{skew}^G + 2d)$. It should arrive at any correct node p_j within d on that node's timer. By IA-3A, τ_j^G will be defined and the message will be processed no later than another d. By IA-3A, $|rt(\tau_i^G) - rt(\tau_j^G)| < t_{skew}^G$. If $rt(\tau_i^G) \leq rt(\tau_j^G) + t_{skew}^G$ then at time $rt(\tau_j)$, by which the message arrived and processed at p_j, we get

$$rt(\tau_j) \leq rt(\tau_i) + 2d \leq rt(\tau_i^G) + r(t_{skew}^G + 2d) + 2d \ ,$$

and therefore

$$rt(\tau_j) \leq rt(\tau_j^G) + t_{skew}^G + r(t_{skew}^G + 2d) + 2d \leq rt(\tau_j^G) + (r+1) \cdot \Phi \ .$$

The case $rt(\tau_j^G) \geq rt(\tau_i^G)$ is simpler. □

Lemma 6.5.2 *If a correct node ever sends* $(echo', p, m, k)$ *then at least one correct node, say* q'*, must have sent* $(echo', p, m, k)$ *at some local-time* $\tau_{q'}$*,* $\tau_{q'} \leq \tau_{q'}^G + (2k+2) \cdot \Phi$.

Proof: Let t be the earliest real-time by which any correct node q sends the message $(echo', p, m, k)$. If $t > rt(\tau_q^G) + (2k+2) \cdot \Phi$, node q should have received $(echo', p, m, k)$ from $n - 2f$ distinct nodes, at least one of which from a correct node, say q', that was sent prior to local local-time $\tau_{q'}^G + (2k+2) \cdot \Phi$. □

Lemma 6.5.3 *If a correct node ever sends* $(echo', p, m, k)$ *then* p*'s message* $(init, p, m, k)$ *must have been received by at least one correct node, say* q'*, at some time* $\tau_{q'}$*,* $\tau_{q'} \leq \tau_{q'}^G + 2k \cdot \Phi$.

Proof: By Lemma 6.5.2, if a correct node ever sends $(echo', p, m, k)$, then some correct node q should send it at local-time τ_q, $\tau_q \leq \tau_q^G + (2k+2) \cdot \Phi$. By the primitive MSGD-BROADCAST, q have received $(init', p, m, k)$ from at least $n - f$ nodes by some local-time τ_q, $\tau_q \leq \tau_q^G + (2k+2) \cdot \Phi$. At least one of them is a correct node q'' who have received $n - 2f$ $(echo, p, m, k)$ at some local-time $\tau_{q''}$, $\tau_{q''} \leq \tau_{q''}^G + (2k+1) \cdot \Phi$. One of which was sent by a correct node $\bar{q}$ that should have received $(init, p, m, k)$ before sending $(echo, p, m, k)$ at some local-time $\tau_{\bar{q}}$, $\tau_{\bar{q}} \leq \tau_{\bar{q}}^G + 2k \cdot \Phi$. □

Lemma 6.5.4 *If a correct node* p *invokes the primitive* MSGD-BROADCAST (p, m, k) *at real-time* t_p*, then each correct node* q *accepts* (p, m, k) *at some real-time* t_q*, such that* $|t_p - t_q)| \leq 3d$.

Proof: The $init$ message of p sent in Line V will arrive to every node by $t_p + d$. By IA-3A, by $t_p + 2d$ all will have their τ^G defined and will process the $init$ message. By Lemma 6.5.1, all will execute Line W3 by that time. By $t_p + 3d$ all will execute Line X5 and will accept. □

Theorem 6.5.2 *The* MSGD-BROADCAST *primitive presented in Figure 6.3 satisfies properties [TSP-1] through [TSP-4].*

Proof:

Correctness: Assume that a correct node p MSGD-BROADCASTs (p, m, k) at τ_p, $\tau_p \leq \tau_p^G + (2k-1)\cdot\Phi$, on its timer. Any correct node, say q, receives $(init, p, m, k)$ and sends $(echo, p, m, k)$ at some τ_q, $\tau_q \leq \tau_q^G + 2k\cdot\Phi$ on its timer. Thus, any correct node, say $\bar{q}$ receives $n-f$ $(echo, p, m, k)$ from distinct nodes at some $\tau_{\bar{q}}$, $\tau_{\bar{q}} \leq \tau_{\bar{q}}^G + (2k+1)\cdot\Phi$, on its timer and accepts (p, m, k). The second part of the correctness is a result of Lemma 6.5.4.

Unforgeability: If a correct node p does not broadcast (p, m, k), it does not send $(init, p, m, k)$, and no correct node will send $(echo, p, m, k)$ at some τ, $\tau \leq \tau^G + 2k\cdot\Phi$, on its timer. Thus, no correct node accepts (p, m, k) by $\tau^G + (2k+1)\cdot\Phi$ on its timer. If a correct node would have accepted (p, m, k) at a later time it can be only as a result of receiving $n-f$ $(echo', p, m, k)$ distinct messages, some of which must be from correct nodes. By Lemma 6.5.3, p should have sent $(init, p, m, k)$, a contradiction.

Relay: The delicate point is when a correct node issues an accept as a result of getting echo messages. So assume that q_1 accepts (p, m, k) at $t_1 = rt(\tau_1)$ as a result of executing Line X5. By that time it must have received $(echo, p, m, k)$ from $n-f$ nodes, at least $n-2f$ of them sent by correct nodes. Since every correct node among these has sent its message by $\tau^G + 2k\cdot\Phi$ on its timer, by Lemma 6.5.1, all those messages should have arrived to every correct node q_i by $\tau_i \leq \tau_i^G + (2k+1)\cdot\Phi$ on its timer. Thus, every correct node q_i should have sent $(init', p, m, k)$ at some τ_i, $\tau_i \leq \tau_i^G + (2k+1)\cdot\Phi$, on its timer. As a result, every correct node will receive $n-f$ such messages by some $\bar{\tau}$, $\bar{\tau} \leq \tau^G + (2k+2)\cdot\Phi$ on its timer and will send $(echo', p, m, k)$ at that time, which will lead each correct node to accept (p, m, k) at a local-time τ_i.

Now observe that all $n-2f$ $(echo, p, m, k)$ were sent before time t_1. By $t_1 + d$ they arrive to all correct nodes. By $t_1 + 2d$ all will have their τ^G defined and will process them. By $t_1 + 3d$ their $(init', p, m, k)$ will arrive to all correct nodes, which will lead all correct nodes to send $(echo', p, m, k)$. Thus, all correct nodes will accept (p, m, k) at time $\tau_i \leq t_1 + 4d$.

By assumption, $t_1 = rt(\tau_1) \leq rt(\tau_1^G) + r\cdot\Phi$. By IA-3A, $rt(\tau_1^G) \leq rt(\tau_i^G) + t_{skew}^G$. Therefore we conclude: $rt(\tau_i) \leq rt(\tau_1) + 4d \leq rt(\tau_1^G) + r\cdot\Phi + 4d \leq rt(\tau_i^G) + t_{skew}^G + r\cdot\Phi + 4d \leq rt(\tau_i^G) + (r+2)\cdot\Phi$.

The case that the accept is a result of executing Line Z5 is a special case of the above arguments.

Detection of broadcasters: As in the original proof, we first argue the second part. Assume that a correct node q adds node p to $broadcasters$. It should have received $n-2f$ $(init', p, m, k)$ messages. Thus, at least one correct node has sent $(init', p, m, k)$ as a result of receiving $n-2f$ $(echo, p, m, k)$ messages. One of these should be from a correct node that has received the original broadcast message of p.

To prove the first part, we consider two similar cases to support the Relay property. If $r = k$ and the correct node, say q, accepts (p, m, k) as a result of receiving $(echo, p, m, k)$ from $n-f$ nodes by some τ_q, $\tau_q \leq \tau_q^G + (2k+1)\cdot\Phi$, on its timer. At least $n-2f$ of them were sent by correct nodes. Since each correct node among these has sent its message at some τ, $\tau \leq \tau^G + 2k\cdot\Phi$, by Lemma 6.5.1, all those messages should have arrived to any correct node, say q_i, by some τ_i, $\tau_i \leq \tau_i^G + (2k+1)\cdot\Phi$ on its timer. Thus, each correct node, say q_j should have sent $(init', p, m, k)$ at some τ_j, $\tau_j \leq \tau_j^G + (2k+1)\cdot\Phi$, on its timer. As a result, by Lemma 6.5.1, each correct node, say q', will receive $n-f$ such messages by some $\tau_{q'}$, $\tau_{q'} \leq \tau_{q'}^G + (2k+2)\cdot\Phi$ on its timer and will add p to $broadcasters$.

Otherwise, q accepts (p, m, k) as a result of receiving from $n-f$ nodes $(echo', p, m, k)$ by some τ_q on its timer. By Lemma 6.5.2 a correct node, say q_i, sent $(echo', p, m, k)$ at some τ_i, $\tau_i \leq \tau_i^G + (2k+2)\cdot\Phi$. It should have received $n-f$ $(init', p, m, k)$ messages by that time. All

such messages that were sent by correct nodes were sent at some τ, $\tau \leq \tau^G + (2k+1)\cdot\Phi$, on their timers and should arrive at each node q_j, at some τ_j, $\tau_j \leq \tau_j^G + (2k+2)\cdot\Phi$, on its timer. Since there are at least $n-2f$ such messages, all will add p to $broadcasters$ at some τ, $\tau \leq \tau^G + (2k+2)\cdot\Phi$, on their timers.

□

Proof of the SS-BYZ-AGREE Properties

Theorem 6.5.3 *(Convergence) Once the system is coherent, any invocation of* SS-BYZ-AGREE *presented in Figure 6.1 satisfies the Termination property. When $n > 3f$, it also satisfies the Agreement and Validity properties.*

Proof:

Notice that the General G itself is one of the nodes, so if it is faulty then there are only $f-1$ potentially faulty nodes. We do not use that fact in the proof since the version of SS-BYZ-AGREE presented does not refer explicitly to the General. One can adapt the proof and reduce Δ by $2\cdot\Phi$ when specifically handling that case.

Let $\hat{t}$ be the real-time by which the network is correct and there are at least $n-f$ non-faulty nodes. These nodes may be in an arbitrary state at that time. If G does not send any (Initiator, G, M) message for $\Delta + 7d$, all spurious invocations of the primitives and the protocol will be reset by all correct nodes. If G sends such an Initiator message, then within $\Delta + 3d$ of the time that any non-faulty node invokes the protocol, either a decision will take place (by all non-faulty nodes) or all will reset the protocol and its primitives. Beyond that time, any future invocation will happen when all data structures are reset at all non-faulty nodes. Note, that before that time a non-faulty G will not send the Initiator message again.

Thus, by time $\hat{t} + 2\Delta + 10d$, when the system becomes coherent, any invocation of the protocol will take place with empty (fresh) data structures and will follow the protocol as stated.

Lemma 6.5.5 *If a correct node p aborts at local-time τ_p, $\tau_p > \tau_p^G + (2r+1)\cdot\Phi$, on its timer, then no correct node q decides at a time τ_q, $\tau_q \geq \tau_q^G + (2r+1)\cdot\Phi$, on its timer.*

Proof: Let p be a correct node that aborts at time τ_p, $\tau_p > \tau_p^G + (2r+1)\cdot\Phi$. In this case it should have identified at most $r-2$ broadcasters by that time. By the detection of the broadcasters property [TPS-4], no correct node will ever accept $\langle G, m'\rangle$ and $r-1$ distinct messages (q_i, m', i) for $1 \leq i \leq r-1$, since that would have caused each correct node, including p, to hold $r-1$ broadcasters by some time τ, $\tau \leq \tau^G + (2(r-1)+2)\cdot\Phi$ on its timer. Thus, no correct node, say q, can decide at a time $\tau_q \geq \tau_q^G + (2r+1)\cdot\Phi$ on its timer. □

Lemma 6.5.6 *If a correct node p decides at time τ_p, $\tau_p \leq \tau_p^G + (2r+1)\cdot\Phi$, on its timer, then each correct node, say q, decides by some time τ_q, $\tau_q \leq \tau_q^G + (2r+3)\cdot\Phi$ on its timer.*

Proof:

Let p be a correct node that decides at local-time τ_p, $\tau_p \leq \tau_p^G + (2r+1)\cdot\Phi$. We consider the following cases:

1. $r = 0$: No correct node can abort by a time τ, $\tau \leq \tau^G + (2r+1)\cdot\Phi$, since the inequality will not hold. Assume that node p have accepted $\langle G, m' \rangle$ by $\tau_p \leq \tau_p^G + 4d \leq \tau_p^G + \Phi$. By the relay property [TPS-3] each correct node will accept $\langle G, m' \rangle$ by some time τ, $\tau \leq \tau^G + 3\cdot\Phi$ on its timer. Moreover, p invokes MSGD-BROADCAST $(p, m', 1)$, by the Correctness property [TPS-1] it will be accepted by each correct node by time τ, $\tau \leq \tau^G + 3\cdot\Phi$, on its timer. Thus, all correct nodes will have $value \neq \perp$ and will broadcast and stop by time $\tau^G + 3\cdot\Phi$ on their timers.

2. $1 \leq r \leq f$: Node p must have accepted $\langle G, m' \rangle$ and also accepted r distinct (q_i, m', i) messages for all $i, 2 \leq i \leq r$, by time τ, $\tau \leq \tau^G + (2r+1)\cdot\Phi$, on its timer. By Lemma 6.5.5, no correct node aborts by that time. By Relay property [TPS-3] each (q_i, m', i) message will be accepted by each correct node by some time τ, $\tau \leq \tau^G + (2r+3)\cdot\Phi$, on its timer. Node p broadcasts $(p, m', r+1)$ before stopping. By the Correctness property, [TPS-1], this message will be accepted by every correct node at some time τ, $\tau \leq \tau^G + (2r+3)\cdot\Phi$, on its timer. Thus, no correct node will abort by time τ, $\tau \leq \tau^G + (2r+3)\cdot\Phi$, and all correct nodes will have $value \neq \perp$ and will thus decide by that time.

3. $r = f + 1$: Node p must have accepted a (q_i, m', i) message for all $i, 1 \leq i \leq f$, by τ_p, $\tau_p \leq \tau_p^G + (2f+3)\cdot\Phi$, on its timer, where the $f+1$ q_i's are distinct. At least one of these $f+1$ nodes, say q_j, must be correct. By the Unforgeability property [TPS-2], node q_j invoked MSGD-BROADCAST (q_j, m', j) by some local-time τ, $\tau \leq \tau^G + (2j+1)\cdot\Phi$ and decided. Since $j \leq f+1$ the above arguments imply that by some local-time τ, $\tau \leq \tau^G + (2f+3)\cdot\Phi$, each correct node will decide.

□

Lemma 6.5.6 implies that if a correct node decides at time τ, $\tau \leq \tau^G + (2r+1)\cdot\Phi$, on its timer, then no correct node p aborts at time τ_p, $\tau_p > \tau_p^G + (2r+1)\cdot\Phi$. Lemma 6.5.5 implies the other direction.

Termination: Each correct node either terminates the protocol by returning a value, or by time $(2f+3)\cdot\Phi + 3d$ all entries will be reset, which is a termination of the protocol.

Agreement: If no correct node decides, then all correct nodes that execute the protocol abort, and return a $\perp$ value. Otherwise, let q be the first correct node to decide. Therefore, no correct node aborts. The value returned by q is the value m' of the accepted $(p, m', 1)$ message. By [IA-4] if any correct node I-accepts, all correct nodes I-accept with a single value. Thus all correct nodes return the same value.

Validity: Since all the correct nodes invoke the primitive SS-BYZ-AGREE as a result of a value sent by a correct G, they will all invoke INITIATOR-ACCEPT within d of each other with fresh data structure, hence [IA-1] implies validity.

Timeliness:

1. (agreement) For every two correct nodes q and q' that decide on (G, m) at τ_q and $\tau_{q'}$, respectively:

(a) If validity hold, then $|rt(\tau_q) - rt(\tau_{q'})| \leq 2d$, by [IA-3A]; Otherwise, $|rt(\tau_q) - rt(\tau_{q'})| \leq 3d$, by [TPS-1].

(b) $|rt(\tau_q^G) - rt(\tau_{q'}^G)| \leq 5d$ by [IA-3A].

(c) $rt(\tau_q^G), rt(\tau_{q'}^G) \in [t_1 - 2d, t_2]$ by [IA-3B].

(d) $rt(\tau_r^G) \leq rt(\tau_r)$, by [IA-3C], and if the inequality $rt(\tau_r) - rt(\tau_r^G) \leq \Delta$ would not hold, the node would abort right away.

2. (validity) If all correct nodes invoked the protocol in an interval $[t_0, t_0 + d]$, as a result of (Initiator, G, m) sent by a correct G that spaced the sending by $6d$ from its last agreement, then for every correct node q that may have decided $3d$ later than G, the new invocation will still happen with fresh data structures, since they are reset $3d$ after decision. By that time it already reset the data structures (including *latest_accept*) of the last execution, and the new decision time τ_q, satisfies $t_0 - d \leq rt(\tau_q^G) \leq rt(\tau_q) \leq t_0 + 3d$ as implied by [IA-1D].

3. (separation) By [IA-4] the real-times of the I-accepts satisfy the requirements. Since a node will not reset its data structures before terminating the protocol, it will not send a $support$ before completing the previous protocol execution. Therefore, the protocol itself can only increase the time difference between agreements. Thus, the minimal difference is achieved when a decision takes place right after the termination of the INITIATOR-ACCEPT primitive.

□

Chapter 7

Self-stabilizing Byzantine Pulse Synchronization using SS. Byz. Agreement

7.1 Specific Definitions

The nodes regularly invoke "pulses", ideally every *Cycle* real-time units. The invocation of the pulse is preceded by the sending of a message to all the nodes stating the intention of invoking a pulse.

- Δ_{BYZ} represents the maximal real-time required to complete the specific self-stabilizing Byzantine agreement protocol used. (Using SS-BYZ-AGREE in Chapter 6 it becomes $7(2f+3)d$.)

DEFINITION 7.1.1 *The communication network is* **correct** *following* $\Delta_{net} \geq d$ *real-time of continuous non-faulty behavior.*

DEFINITION 7.1.2 *A node is* **correct** *following* $\Delta_{node} \geq$ *Cycle* $+ cycle_{\text{max}}$ *real-time of continuous non-faulty behavior during a period that the communication network is correct.*

7.2 Self-stabilizing Byzantine Pulse-Synchronization

The definitions involved in the notion of pulse synchronization start by defining a subset of the system states, called *pulse_states*, that are determined only by the elapsed real-time since each individual node invoked a pulse (the ϕ's). Nodes that have "tight" or "close" ϕ's will be called a *synchronized* set of nodes. To complete the definition of synchrony there is a need to address the recurring brief time periods in which a node in a synchronized set of nodes has just invoked a pulse while others are about to invoke one. This is addressed by considering nodes whose ϕ's are almost a *Cycle* apart.

The Pulse Synchronization Algorithm

The self-stabilizing Byzantine pulse synchronization algorithm presented is called AB-PULSE-SYNCH (for *Agreement-based Pulse Synchronization*). A *cycle* is the time interval

between two successive pulses that a node invokes. The input value *Cycle* is the ideal length of the *cycle*. The actual real-time length of a *cycle* may deviate from the value *Cycle* in consequence of the clock drifts, uncertain message delays and behavior of faulty nodes. In the proof of Lemma 7.2.12 the extent of this deviation is explicitly presented.

The environment is one without any granted synchronization among the correct nodes besides a bound on the message delay. Thus, it is of no use whether a sending node attaches some time stamp or round number to its messages in order for the nodes to have a notion as to when those messages supposedly were sent. Hence in order for all correct nodes to symmetrically relate to any message disseminated by some node, a mechanism for agreeing on which phase of the algorithm or "time" that the message relates to must be implemented. This is fulfilled by using SS-BYZ-AGREE, a self-stabilizing Byzantine agreement protocol presented in [20]. The mode of operation of this protocol is as follows: A node that wishes to initiate agreement on a value does so by disseminating an initialization message to all nodes that will bring them to (explicitly) invoke the SS-BYZ-AGREE protocol. Nodes that did not invoke the protocol may join in and execute the protocol in case enough messages from other nodes are received during the protocol. The protocol requires correct initiating nodes not to disseminate initialization messages too often. In the context of the current paper, a "Support-Pulse" message serves as the initialization message.

When the protocol terminates, the protocol SS-BYZ-AGREE returns at each node q a triplet (p, m, τ_q^p), where m is the agreed value that p has sent. The value τ_q^p is an estimate, on the receiving node q's local clock, as to when node p have sent its value m. We also denote it as the "recording time" of (p, m). Thus, a node q's decision value is $\langle p, m, \tau_q^p \rangle$ if the nodes agreed on (p, m). If the sending node p is faulty then some correct nodes may agree on $(p, \perp)$, where $\perp$ denotes a non-value, and others may not invoke the protocol at all. The function $rt(\tau_q)$ represents the real-time when the local clock of q reads τ_q. The AB-PULSE-SYNCH algorithm uses the SS-BYZ-AGREE protocol for a single message only ("Support-Pulse" message) and not for every message communicated. Thus the agreement is on whether a certain node sent a "Support-Pulse" message and when, and not on any actual value sent. Correct nodes do not send this message more than once in a *cycle*.

The SS-BYZ-AGREE protocol satisfies the following typical Byzantine agreement properties:

Agreement: If the protocol returns a value ($\neq\perp$) at a correct nodes, it returns the same value at all correct nodes;
Validity: If all correct nodes are triggered to invoke the protocol SS-BYZ-AGREE by a value sent by a correct node p, then all correct nodes return that value;
Termination: The protocol terminates in a finite time;

It also satisfies some specific timeliness properties that are listed in Section 7.2.

The heuristics behind AB-PULSE-SYNCH protocol are as following:

- Once the node approaches its end of *Cycle*, as measured on its physical timer, it sends a "Propose-Pulse" message stating so to all nodes.
- When $(n - f)$ distinct "Propose-Pulse" messages are collected, the node sends a "Support-Pulse" message that states so to all nodes. This serves as the initialization message for invoking agreement.

- Upon receiving such a message a receiving node invokes self-stabilizing Byzantine agreement ([20]) on the fact that it received such a message from the specific node. We require that *Cycle* be long enough to allow the agreement instances to terminate.
- If all correct nodes invoked agreement on the same message within a short time window then they will all agree that the sender indeed sent this "Support-Pulse" message and all will have proximate estimates as of when that node could have sent this message.
- The time estimate is then used to reset the countdown timer for the next pulse invocation and a consequent "reset" messages to be sent. Each new agreement termination causes a renewed reset.
- Upon arrival of a reset message the sending node is taken off the list of nodes that have ended their *Cycle* (as indicated by the earlier arrival of a "Propose-Pulse" message for that node).
- Thus, some short time after all correct nodes have done at least one reset of their cycle countdown timer, no new agreement can be initiated by any node (faulty or correct).
- Thus, there is one agreement termination that marks a small time-window within which all correct nodes do a last reset of the cycle countdown timer. Thus, essentially, all correct nodes have synchronized the invocation of their next pulse.

The algorithm is executed in an "event-driven" manner. Thus, each node checks the conditions and executes the steps (blocks) upon an event of receiving a message or a timer event. To simplify the presentation it is assumed in the algorithm that when a correct node sends a message it receives its own message through the communication network, like any other correct node.

The algorithm assumes a timer that measures interval of time of size *Cycle*. The algorithm uses several sets of messages or references that are reset throughout the algorithm, and every message that have arrived more than *Cycle* $+ 2d$ ago is erased.

The algorithm assumes the ability of nodes to estimate some time intervals, like at Line C2. These estimates can be carried out also in a self-stabilizing environment, by tagging each event according to the reading of the local timer. So even if the initial values are arbitrary and cause the non-faulty node to behave inconsistently, by the time it is considered correct the values will end up resetting to the right values. Note that the nodes do not exchange clock values, rather they measure time locally on their own local timers. It is assumed that a non-faulty node handles the wrap around of its local timer while estimating the time intervals.

Note that there is no real reason to keep a received message after it has been processed and its sender been referred to in the appropriate data structures. Hence, if messages are said to be deleted after a certain period, the meaning is to the reference of the message and not the message itself, which can be deleted subsequent to processing.

For reasons of readability we have omitted the hardware clock skew ρ, from the constants, equations and proofs. The introduction of ρ does not change the protocol whatsoever nor any of the proof arguments. It only adds a small insignificant factor to many of the bounds.

We now seek to explain in further detail the blocks of the algorithm:

Block A: We assume that a background process continuously reduces $cycle\_countdown$, intended to make the node count *Cycle* time units on its physical timer. On reaching 0, the background process resets the value back to *Cycle*. It expresses its intention to synchronize its forthcoming pulse invocation with the pulses of the other nodes by sending an endogenous **"Propose-Pulse"**

Algorithm AB-PULSE-SYNCH*(n, f, Cycle)* */* continuously executed at node q */*

/ assumes a background process that continuously reduces cycle_countdown */*

A1. **if** $(cycle\_countdown = 0)$ **then**
A2. $cycle\_countdown :=$ *Cycle*;
A3. send "Propose-Pulse" message to all; */* endogenous message */*
B1. **if** received "Propose-Pulse" message from a sender p and $p \notin$ *recent_reset*$_q$ **then**
B2. add p to *proposers*$_q$; C1. **if** $q \in$ *proposers*$_q$ & $\|$*proposers*$_q\| \geq n - f$ and
C2. did not send a "Support-Pulse" in the last *Cycle* $- 8d$ **then**
C3. send "Support-Pulse(*proposers*$_q$)" to all; */* support the forthcoming pulse */*
D1. **if** received "Support-Pulse(*proposers*$_p$)" message from a sender p and in the last *Cycle* $- 11d$
D2. did not invoke SS-BYZ-AGREE (p,"support") or decide on $\langle p$,"$support$",_$\rangle$ and
D3. within d of its reception $\|$(*proposers*$_q \cup$ *recent_reset*$_q$) $\cap$ *proposers*$_p\| \geq f + 1$ **then**
D4. SS-BYZ-AGREE(p, "$support$") */* invoke agreement on the pulse supporter */*;
E1. **if decided** on $\langle p$, "$support$", $\tau_q^p\rangle$ at some local-time τ_q **then** */* on non $\perp$ value */*
E2. **if** $\tau_q^p \geq$ *latest_accept*$_q$ **then**
E3. *latest_accept*$_q := \tau_q^p$; */* the latest agreed supporter so far at q */*
E4. **if** not invoked a pulse since local-time $\tau_q - (\Delta_{\rm BYZ} + 6d)$ **then** */* pulse separation */*
E5. **invoke** the *pulse* event;
E6. $cycle\_countdown :=$ *Cycle* $- (\tau_q - \tau_q^p)$; */* reset cycle */*
E7. send "Reset" message to all and remove yourself, q, from *proposers*$_q$;
F1. **if** received "Reset" from a sender p **then**
F2. move p from *proposers*$_q$ to *recent_reset*$_q$; */* recent_reset decay within $2d + \epsilon$ time */*

Continuously ongoing cleanup:

G1. delete an older message if a subsequent one arrives from the same sender;
G2. delete any data in *recent_reset*$_q$ after $2d + \epsilon$ time units;
G3. reset *cycle_countdown* to be *Cycle* if *cycle_countdown* $\notin [0,$ *Cycle*$]$;
G4. reset *latest_accept*$_q$ to be $\tau -$ *Cycle* if *latest_accept*$_q \notin [\tau -$ *Cycle*$, \tau]$;
G5. delete any other message or data that is older than *Cycle* $+ 2d$ time units;

Figure 7.1: The AB-PULSE-SYNCH Pulse Synchronization Algorithm

message to all nodes. Note that a reset is also done if $cycle\_countdown$ holds a value not between 0 and *Cycle*. The value of $cycle\_countdown$ is also reset once the "pulse" is invoked. Observe that nodes typically send more than one message in a *cycle*, to prevent cases in which the system may be invoked in a deadlocked state.

Block B: The "Propose-Pulse" messages are accumulated at each correct node in its *proposers* set. We say that two messages are **distinct** if they were sent by different nodes.

Block C: These messages are accumulated until enough (at least $n - f$) have been collected. If in addition the node has already proposed itself then the node will declare this event through the sending of a **"Support-Pulse"** message, unless it has already sent such a message not long ago. The message bears a reference to the nodes in the *proposers* set of the sender. Note that a node that was not able to send the message because sending one not long ago, may send it later when

the conditions will hold.

Block D: Any such "Support-Pulse" message received is then checked for credibility by verifying that the history it carries has enough (at least $f + 1$) backing-up in the receiver's *proposers* set and that a previous message was not sent recently. It is only then that agreement is initiated, on a credible pulse supporter. Note that a correct node would not have supported a pulse (sent a "Support-Pulse" message) unless it received $n - f$ propose messages and has not sent one recently. Thus all correct nodes will receive at least $f + 1$ propose messages from correct nodes and will join the agreement initiation by the pulse supporter within d real-time units.

Block E: The Byzantine agreement protocol decides whether a certain node issued a "Support-Pulse" message. Each node q decides at some local-time τ_q. The agreement protocol also returns an estimate as of when, on the deciding node's local clock, the message was sent by the initiating node. This time is denoted τ_q^p. Correct nodes end up having bounded differences in the real-time translation of their τ^p values, for a specific agreement.

When a node decides on a value it checks whether the τ^p returned by the agreement protocol is the most recent decided on so far in the current *cycle*. Only then are lines E3-E7 executed. Note that the same agreement instance may return a τ^p, which is the most recent one for a certain correct node but may not be the most recent at another correct node. This can happen because correct nodes terminate the SS-BYZ-AGREE protocol within $3d$ time units of each other,[1] and their translation of the realtime of the τ^p values may differ by $5d$. Thus, this introduces a $3d$ time units uncertainty between the execution of the subsequent lines at correct nodes.

In Line E4-E5 a pulse is invoked if no pulse has recently been invoked. In Line E6 the node now resets the *cycle* so that the next pulse invocation is targeted to happen at about one *Cycle* later. In Line E7 a "Reset" message is sent to all nodes to inform that a reset of the *cycle* has been done. The function of this message is to make every node that resets, be taken out of the *proposers* set of all other correct nodes[2]. To ensure that only one pulse is invoked in the minimal time span of a *cycle* a pulse will not be invoked in Line E4 if done so recently.

Block F: This causes all correct nodes to eventually remove all other correct nodes from their *proposers*. Thus, about $2d$ after all correct nodes have executed Line E7 at least once, no instance of SS-BYZ-AGREE will be initiated by any correct node and consequently no more agreements can terminate (beyond the currently running ones). The last agreement decision of the correct nodes, done within a short time-window of each other, returns different but closely bounded τ^p values at the correct nodes. Consequently they all reset their $cycle\_countdown$ counters to proximate values. This yields a quiescent window between the termination of the last agreement and the next pulse invocation, which will be invoked within a small time window of each other.

Block G: The scheme outlined above is not sufficient to overcome the cases in which some nodes initialize with reference to spurious messages sent by other nodes while such messages were not actually sent. The difficulty lies in the fact that Byzantine nodes may now intervene and

[1] It is part of the timeliness properties of the SS-BYZ-AGREE protocol, see Section 7.2.

[2] Note that a node may send multiple "Reset" messages. It is done in order to simplify some of the claims in the proof.

constantly keep the correct nodes with asymmetric views on the sets of messages received. To overcome this, AB-PULSE-SYNCH has a decay process in which each data that is older than some period is deleted.

Note that the decaying of values is carefully done so that correct nodes never need to consider messages that arrived more than *Cycle* $+ 2d$ ago.

Proof of Correctness

The proof of correctness requires very careful argumentation and is not a straightforward standard proof of the basic properties. The critical parts in the proof is showing that despite the complete chaotic initialization of the system the correct nodes are able to produce some relation among their local clocks and force the faulty nodes to leave a short interval of time into which no recording time refers to, followed by an interval during which no correct node updates its *latest_accept*. After such intervals we can argue about the convergence of the states of the correct nodes, proving that stability is secured. The nontraditional values of the various constants bounding *Cycle* has to do with the balance between ensuring the ability to converge and limiting the ability of the Byzantine nodes to disturb the convergence by introducing critically timed pulse events that may disunite the correct nodes.

The proof shows that when the constants are chosen right, no matter what the faulty nodes will do and no matter what the initial values are, there will always be two intervals of inactivity, concurrently at all correct nodes, after which the correct nodes restore consistency of their pulses.

The proof uses the following specific properties of the SS-BYZ-AGREE protocol ([20]):

Timeliness-Agreement Properties:

1. (agreement) For every two correct nodes q and q' that decides $\langle p, m, \tau_q^p \rangle$ and $\langle p, m, \tau_{q'}^p \rangle$ at local times τ_q and $\tau_{q'}$, respectively:
   (a) $|rt(\tau_q) - rt(\tau_{q'})| \leq 3d$, and if validity holds, then $|rt(\tau_q) - rt(\tau_{q'})| \leq 2d$.
   (b) $|rt(\tau_q^p) - rt(\tau_{q'}^p)| \leq 5d$.
   (c) $rt(\tau_q^p), rt(\tau_{q'}^p) \in [t_1 - 2d, t_2]$, where $[t_1, t_2]$ is the interval within which all correct nodes that actually invoked SS-BYZ-AGREE(p, m) did so.
   (d) $rt(\tau_q^p) \leq rt(\tau_q)$ and $rt(\tau_q) - rt(\tau_q^p) \leq \Delta_{\text{BYZ}}$ for every correct node q.
2. (validity) If all correct nodes invoked the protocol in an interval $[t_0, t_0 + d]$, as a result of some initialization message containing m sent by a correct node p that spaced the sending by at least $6d$ from the completion of the last agreement on its message, then for every correct node q, the decision time τ_q, satisfies $t_0 - d \leq rt(\tau_q^p) \leq rt(\tau_q) \leq t_0 + 3d$.
3. (separation) Let q be any correct node that decided on any two agreements regarding p at local times τ_q and $\bar{\tau}_q$, then $t_2 + 5d < \bar{t}_1$ and $rt(\tau_q) + 5d < \bar{t}_1 < rt(\bar{\tau}_q)$, where t_2 is the latest real-time at which a correct node invoked SS-BYZ-AGREE in the earlier agreement and $\bar{t}_1$ is the earliest real-time that SS-BYZ-AGREE was invoked by a correct node in the later agreement.

The AB-PULSE-SYNCH requires the following bounds on the variables:

- $\mathit{Cycle} \geq \max[(10f + 16)d,\ \Delta_{\text{BYZ}} + 14d]$.
- $\Delta_{node} \geq \mathit{Cycle} + cycle_{\text{max}}$.
- $\Delta_{net} \geq d$.

The requirements above, and the definitions of correctness imply that from an arbitrary state the system becomes coherent within 2 cycles.

Note that in all the theorems and lemmata in this paper, if not stated differently, it is assumed that the system is coherent, and the claims hold as long as the system stays coherent.

In the proof, whenever we refer to correct nodes that decide we consider only decisions on $\neq\perp$ values. When the agreement returns $\perp$ it is not considered a decision, and in such a case the agreement at other correct nodes may not return anything or may end up in decaying all related messages.

Theorem 7.2.1 *(*Convergence*) From an arbitrary (but coherent) state a synchronized_pulse_state is reached within 4 cycles, with $\sigma = 3d$.*

Proof: A node that recovers may find itself with arbitrary input variables and in an arbitrary step in the protocol. Within a *cycle* a recovered node will decay all spurious "messages" that may exist in its data structures. Some of these might have been resulted from incorrect initial variables, such as when invoking the SS-BYZ-AGREE protocol without the specified pre-conditions. Such effects also die out within a *cycle*.

The above argument implies that by the time the node is considered correct, all messages sent by non-faulty nodes that are reflected in its data structures were actually sent by them (at the arbitrary state at which they are). Thus, by the time that the system becomes coherent the set of correct nodes share the values they hold in the following sense: if a message sent by a non-faulty node is received by a correct node, then within d it will be received by all other correct nodes; and all future messages sent by correct nodes are based on actual messages that were received.

Once the system is coherent, then there are at least $n - f$ correct nodes that follow the protocol, and all messages sent among them are delivered by the communication network and processed by the correct nodes within d real-time units.

Lemma 7.2.1 *Within d real-time units of the sending of a "Propose-Pulse" message by a correct node p, it appears in $\mathit{proposers}_q$ of any correct node q. Furthermore, it appears in $\mathit{proposers}_q$ only if p sent a "Propose-Pulse" message within the last d units of time.*

Proof: From the coherence of the system, p's message arrives to all within d real-time. By the Timeliness-Agreement Property (1d) and the bounds on *Cycle*, a node that have recently sent a "Reset" message resets its *cycle_countdown* to a value that is at least $\mathit{Cycle} - \Delta_{\text{BYZ}} > 14d$. Thus, the minimum real-time between the receipt of its past "Reset" and its current "Propose-Pulse" at any correct node is more than $2d$ apart, and therefore by the time its "Propose-Pulse" message arrives it will not appear in $\mathit{recent\_reset}_q$ at any correct node q. The second part is true because p can be in $\mathit{proposers}_q$ without prior sending of a "Propose-Pulse" message only if node q recovered in that state. But by the time node q is considered correct any reference to such a message has already been decayed. □

Lemma 7.2.2 *In every real-time interval equal to Cycle, every correct node sends either a "Propose-Pulse" message or a "Reset" message.*

Proof: Recall that every correct node's $cycle\_countdown$ timer is continuously running in the background and would be reset to hold a value within *Cycle* if it initially held an out-of-bound value. Thus, if the $cycle\_countdown$ is not reset to a new value when a "Reset" is invoked, then within *Cycle* real-time units the $cycle\_countdown$ timer will eventually reach 0 and a "Propose-Pulse" message will consequently be sent. Whenever a $cycle\_countdown$ is reset, its value is always at most *Cycle*. □

Lemma 7.2.3 *Within d real-time units of sending a "Reset" message by a correct node p, that node does not appear in $proposers_q$ of any correct node q. Furthermore, a correct node p is deleted from $proposers_q$ only if it sent a "Reset" message.*

Proof: The first part follows immediately from executing the protocol in a coherent state. The only sensitive point arises when a "Propose-Pulse" message that was sent by p prior to the "Reset" message arrives after the "Reset" message. This can happen only if the "Propose-Pulse" message was sent within d of the "Reset" message. But in this case the protocol instructs node q not to add p to $proposers_q$. For proving the second part we need to show that a correct node is not removed from $proposers_q$ because q decayed it. By Lemma 7.2.1 it appears in $proposers_q$ only because of sending a "Propose-Pulse" message. By Lemma 7.2.2 it will resend a new message before q decays the previous message, because messages are decayed (Block G) only after $Cycle + d$. □

Lemma 7.2.4 *Every correct node invokes* SS-BYZ-AGREE $(p, \text{"}support\text{"})$ *within d real-time units of the time a correct node p sends a "Support-Pulse" message.*

Proof: If a correct node p sent a "Support-Pulse" message in Line C3, then the preconditions of Line D2 hold because the last reception of "Support-Pulse" and the last invocation of SS-BYZ-AGREE $(p, \text{"}support\text{"})$ that followed took place at least $Cycle - 8d - d$ ago, proving the first condition. By the Timeliness-Agreement property (2) the last decision took place at least $Cycle - 8d - 3d$ ago, proving the other condition. The condition in Line D3 clearly holds for all correct nodes. This is because within d real-time units every correct node in $proposers_p$ will appear in $proposers_q$ and every correct node that was deleted from $proposers_q$ and is not in $recent\_reset_q$ should have been already deleted from $proposers_p$. To prove this last claim, assume that node q received "Reset" from a correct node v at real-time t. By $t + d$ this message should arrive at p, and therefore any "Propose-Pulse" message from p that contains v should be sent before that and should be received before $t + 2d$, thus before removing v from $recent\_reset_q$. □

Lemma 7.2.5 *If a correct node p sends "Support-Pulse" at real-time t_0 then every correct node q decides $\langle p, \text{"}support\text{"}, \tau_q^p \rangle$ at some local-time τ_q, such that $t_0 - d \leq rt(\tau_q^p) \leq rt(\tau_q) \leq t_0 + 3d$ and $t_0 \leq rt(\tau_q)$.*

Proof: By Lemma 7.2.4 all correct nodes invoke SS-BYZ-AGREE $(p, \text{"}support\text{"})$ in the interval $[t_0, t_0 + d]$. Thus the precondition conditions for the Timeliness-Agreement property (2) hold. Therefore, each correct node q decides on $\langle p, \_, \tau_q^p \rangle$ at some real-time $rt(\tau_q)$ that satisfies $t_0 - d \leq rt(\tau_q^p) \leq rt(\tau_q) \leq t_0 + 3d$. □

Lemma 7.2.6 *Let $[t, t + Cycle]$ be an interval such that for no correct node $rt(latest\_accept) \in [t, t + Cycle]$, then by $t + Cycle + 4d$ all correct nodes decide.*

Proof: Assume that all decisions by correct nodes resulted in $rt(latest\_accept) \leq t$. Thus, since there are no updates to *cycle_countdown* the *cycle_countdown* at all correct nodes should expire by $t + Cycle$. By Lemma 7.2.5, if any correct node would have sent "Support-Pulse" in that interval, then we are done. Otherwise, by that time all should have sent a "Propose-Pulse" message. Since no node removes old messages for $Cycle + 2d$, and more than $Cycle - 8d$ real-time passed, by $t + Cycle + d$ at least one correct node will send a "Support-Pulse" message. By Lemma 7.2.4, all will invoke SS-BYZ-AGREE within another d real-time units. The Timeliness-Agreement property (2) implies that by $t + Cycle + 4d$ all will decide. □

Note that if a faulty node sends "Support-Pulse", some correct node may join and some may not, and the actual agreement on a value $\neq \perp$ and the time of such an agreement depends on the behavior of the faulty nodes. We address that later on in the proof. We first prove a technical lemma.

Lemma 7.2.7 *Let t', be a time by which all correct nodes decided on some values since the system became coherent. Let B' and B satisfy $B' \leq B$, and $3d \leq B$. If no correct node decides on a value that causes updating latest_accept to a value in an interval $[t', t' + B]$, and no correct node updates its latest_accept or resets its cycle_countdown during the real-time interval $[t' + B', t' + B]$, then for any pair of correct nodes $|cycle\_countdown_q(t'') - cycle\_countdown_{q'}(t'')| \leq 5d$ for any t'', $t' + B' \leq t'' \leq t' + B$.*

Proof: By assumption, the agreements prior to t' satisfy the Timeliness-Agreement properties. Past $t' + B'$ and until $t' + B$ no node updates its *latest_accept*. Thus, for all nodes the value of $rt(latest\_accept)$ is bounded by $rt(latest\_accept) \leq t'$. Let q be the correct node with the maximal $rt(latest\_accept_q)$ that was set following a decision $\langle p_1, \_, \tau_1^{p_1} \rangle$ at timer τ_1, where $latest\_accept_q = \tau_1^{p_1}$. By the Timeliness-Agreement property (1a), any correct node v will execute Line E2 following a decision on $\langle p_1, \_, \mu_1^{p_1} \rangle$ at some timer μ_1, such that $|rt(\tau_1) - rt(\mu_1)| \leq 3d$. By property (1b), $rt(\tau_1^{p_1}) - rt(\mu_1^{p_1}) \leq 5d$. Assume first that $latest\_accept_v = \mu_1^{p_1}$.

At local-time τ_1, at q:

$$cycle\_countdown_q(\tau_1) = Cycle - (\tau_1 - \tau_1^{p_1}) = Cycle - (rt(\tau_1) - rt(\tau_1^{p_1})).$$

At real-time t'', $t'' \geq rt(\tau_1^{p_1})$, at q:

$$cycle\_countdown_q(t'') = Cycle - (rt(\tau_1) - rt(\tau_1^{p_1})) - (t'' - rt(\tau_1)) = Cycle - (t'' - rt(\tau_1^{p_1})).$$

Similarly at real-time t'', $t'' \geq rt(\mu_1^{p_1})$, at v:

$$cycle\_countdown_v(t'') = Cycle - (t'' - rt(\mu_1^{p_1})).$$

Thus,

$$|cycle\_countdown_q(t'') - cycle\_countdown_v(t'')| \leq 5d.$$

Otherwise, v assigned $latest\_accept_v$ as a result of deciding on some $\langle p_2, \_, \mu_2^{p_2} \rangle$ at some timer μ_2, $rt(\mu_2) \leq t'$, where $latest\_accept_v = \mu_2^{p_2}$. Let τ_2 be the timer at q when it decided $\langle p_2, \_, \tau_2^{p_2} \rangle$.

By the Validity and the Timeliness-Agreement properties, $|rt(\tau_2) - rt(\mu_2)| \leq 3d$ and $|rt(\tau_2^{p_2}) - rt(\mu_2^{p_2})| \leq 5d$.

By assumption,

$$rt(\tau_1^{p_1}) \geq rt(\mu_2^{p_2}) \geq rt(\mu_1^{p_1}) \geq rt(\tau_1^{p_1}) - 5d.$$

At local-time τ_1, at q:

$$\textit{cycle\_countdown}_q(\tau_1) = \textit{Cycle} - (\tau_1 - \tau_1^{p_1}) = \textit{Cycle} - (rt(\tau_1) - rt(\tau_1^{p_1})).$$

At local-time μ_2, at v:

$$\textit{cycle\_countdown}_v(\mu_2) = \textit{Cycle} - (\mu_2 - \mu_2^{p_2}).$$

Let $t'' = rt(\tau_1'') = rt(\mu_1'')$, then

$$\textit{cycle\_countdown}_q(\tau_1'') = \textit{Cycle} - (t'' - rt(\tau_1^{p_1})),$$

and

$$\textit{cycle\_countdown}_v(\mu_2'') = \textit{Cycle} - (t'' - rt(\mu_2^{p_2})).$$

Therefore, we conclude

$$|\textit{cycle\_countdown}_q(t'') - \textit{cycle\_countdown}_v(t'')| \leq 5d.$$

□

Lemma 7.2.8 *If a correct node p sends a "Support-Pulse" at some real-time t_0 then:*

1. *No correct node will invoke* SS-BYZ-AGREE *during the period $[t_0 + 6d, t_0 + \textit{Cycle} - d]$;*
2. *No correct node sends a "Support-Pulse" or "Propose-Pulse" during that period;*
3. *The cycle_countdown counters of all correct nodes expire within $5d$ of each other at some real-time in the interval $[t_0 + \textit{Cycle} - d, t_0 + \textit{Cycle} + 6d]$.*

Proof: By Lemma 7.2.5 each correct node decides on p's "Support-Pulse". Each correct node that did not update its *latest_accept* recently, will send a "Reset" message as a result of this decision. Since several agreements from different nodes may be executed concurrently, we need to consider their implication on the resulting behavior of the correct nodes.

Consider first the case that a correct node reached a decision and sent "Reset" before deciding on p's "Support-Pulse". If the decision took place before $t_0 - d$ then, by the Timeliness-Agreement property (2), it will update it's *latest_accept* after the decision on p's "Support-Pulse".

By the same Timeliness-Agreement properties, every correct node that has not sent "Reset" already, will end up updating its *latest_accept* and sending "Reset" at some time during the interval $[t_0 - d, t_0 + 3d]$. By $t_0 + 4d$ no correct node will appear in *proposers* of any correct node and until it will send again a "Propose-Pulse" message, since its "Reset" message will arrive to all non-faulty nodes. Thus, from time $t_0 + 4d$ and until some correct node will send a new "Propose-Pulse" message, no correct node will send "Support-Pulse" message. Moreover, past $t_0 + 6d$ no correct node will invoke a SS-BYZ-AGREE in Line D3, because all correct nodes will not appear also

in *recent_reset*. Observe that if there is a "Propose-Pulse" message in transit from some correct node v, or if a correct node v happened to send one just before sending the "Reset" message, that "Propose-Pulse" will arrive within d of receiving the "Reset" message, and therefore by the time that node will be removed from *recent_reset* all such messages will arrive and therefore node v will not be added to *proposers* as a result of that message later than $t_0 + 6d$.

Even though different correct nodes may compute their $\mathit{latest\_accept}_q$ as a result of different agreements, by the Timeliness-Agreement properties (1d) and (2), at time $t_0 + 6d$ the value of $\mathit{latest\_accept}_q$ satisfies $rt(\mathit{latest\_accept}_q) \in [t_0 - d, t_0 + 6d]$, for every correct node q.

Past time $t_0 + 6d$ and until $t_0 + 6d + \Delta_{\text{BYZ}}$ correct nodes may still decide on values from other agreements that were invoked in the past by faulty nodes. By the Timeliness-Agreement property (1c) no such value result in a *latest_accept* later than $t_0 + 6d$, since no correct node will invoke SS-BYZ-AGREE until some correct node will send a future "Propose-Pulse" message.

Let t_q be the latest real-time a correct node q updated the calculation of *cycle_countdown* because of a *latest_accept* value in the interval $[t_0 - d, t_0 + 6d]$. It will send its next "Propose-Pulse" message at $t_q + \mathit{cycle\_countdown} = t_q + \mathit{Cycle} - (t_q - rt(\mathit{latest\_accept}_q)) = \mathit{Cycle} + rt(\mathit{latest\_accept}_q) \geq t_0 + \mathit{Cycle} - d$. Thus, the earliest real-time a correct node will send "Propose-Pulse" message will be at $t_0 + \mathit{Cycle} - d$. Until that time no correct node will send a "Propose-Pulse" or "Support-Pulse" message or invoke SS-BYZ-AGREE, proving (1) and (2).

The bound on *Cycle*, implies that during the real-time interval $[t_0 + 6d, t_0 + \mathit{Cycle} - \Delta_{\text{BYZ}}]$ there is a window of at least $14d - 6d - d > 3d$ with no recording time that refers to it. Denote this interval by $[t', t' + B\prime]$, where $B\prime \leq t_0 + \mathit{Cycle} - \Delta_{\text{BYZ}} \leq t_0 + \mathit{Cycle} - d$. The above argument implies that in the interval $[t' + B\prime, t_0 + \mathit{Cycle} - d]$ no correct node will update its *latest_accept*, and therefore the conditions of Lemma 7.2.7 hold.

Thus, the *cycle_countdown* counters of all correct nodes expire within $5d$ past time $t_0 + \mathit{Cycle} - d$. Looking back at the latest real-time, $t_q \in [t_0 - d, t_0 + 6d]$, at which a correct node q updated the calculation of *cycle_countdown* the node will send its next "Propose-Pulse" message at $t_q + \mathit{cycle\_countdown} = t_q + \mathit{Cycle} - (t_q - rt(\mathit{latest\_accept}_q)) = \mathit{Cycle} + rt(\mathit{latest\_accept}_q) \leq t_0 + \mathit{Cycle} + 6d$. Proving (3). □

Lemma 7.2.6 above implies that the nodes will not deadlock, despite the arbitrary initial states they could have recovered at. Moreover, by Lemma 7.2.8, once a correct node succeeds in sending a "Support-Pulse" message, all correct nodes will converge. We are therefore left with the need to address the possibility that the faulty nodes will use the divergence of the initial values of correct nodes to prevent convergence by constantly causing them to decide and to update their *cycle_countdown* counter without enabling a correct node to reach a point at which it sends a "Support-Pulse" message.

By Lemma 7.2.6 within $\mathit{Cycle} + 4d$ of the time the system becomes coherent all correct nodes execute Line E1, thus within $\mathit{Cycle} + 4d$ from the time the system became coherent. Let t_1 be some real-time in that period by which all non-faulty nodes executed Line E1. If any correct node sends a "Support-Pulse" message, then we are done. Assume otherwise. Since no correct node will invoke SS-BYZ-AGREE for any node more than once within a $\mathit{Cycle} - 11d$, as we prove later, there will be at most f decisions between t_1 and $t_1 + \mathit{Cycle} - 11d$. Since each decision returns recording times to nodes that range over at most a $5d$ real-time window, and since $\mathit{Cycle} > (10f + 16)d$, there should be a real-time interval $[t_2, t_2 + 5d]$, that no recording time refers to any real-time within it. This reasonings leads to the following lemma.

Lemma 7.2.9 *Assume that no correct node decision results in a recording time τ_q^p that refers to real-time $rt(\tau_q^p)$ in the real-time interval $[t', t' + 5d]$. Then by $t' + Cycle + 4d$ all correct nodes decide, update their latest_accept and send "Reset", within $3d$ real-time units of each other.*

Proof: By the Timeliness-Agreement property (1b), any decision that will take place later than $t' + \Delta_{BYZ}$ would result in *latest_accept* $> t'$. By Lemma 7.2.6, by $t' + Cycle + 4d$ all correct nodes' decisions lead to $rt(latest\_accept) > t'$, and by assumption to $rt(latest\_accept) > t' + 5d$. Let q be the first correct node to decide and update its *latest_accept* to a value larger than $t' + 5d$ on some $\langle p, \_, \tau_q^p \rangle$ for $rt(\tau_q^p) > t' + 5d$, at some real-time $t'' \geq t'$. By the Timeliness-Agreement property (1d), $t'' \geq rt(\tau_q^p)$. Moreover, since the $rt(\tau^p)$ are at most $3d$ apart, by $t'' + 3d$ all correct nodes will decide on some values and will update the *latest_accept* value. Therefore, in the interval $[t'', t'' + 3d]$ all correct nodes should update their *latest_accept*, with $rt(latest\_accept) \geq t'$. Thus, all correct nodes will execute Line E7 as a result of such decisions. Therefore, all correct nodes will send a "Reset" messages within $3d$ of each other. □

Let t_2 be a real time at which the above lemma holds. Let t_3 be the real-time past t_2 by which all correct nodes send "Reset" as Lemma 7.2.9 claims. Thus, all correct nodes sent "Reset" in the real-time interval $[t_3 - 3d, t_3]$ and by $t_3 + d$ no correct node will appear in the *proposers* of any other correct node.

The final stage of the proof is implied from the following lemma.

Lemma 7.2.10 *If all correct nodes send a "Reset" in the period $[t_0, t_0 + 3d]$ then:*

1. *No correct node will invoke* SS-BYZ-AGREE *during the period $[t_0 + 6d, t_0 + Cycle - \Delta_{BYZ}]$;*
2. *No correct node sends a "Support-Pulse" or "Propose-Pulse" during that period;*
3. *The cycle_countdown counters of all correct nodes expire within $5d$ of each other at some real-time in the interval $[t_0 + Cycle - \Delta_{BYZ}, t_0 + Cycle + 6d]$.*

Proof: By real-time $t_0 + 4d$ all correct nodes will receive all the $n - f$ "Reset" messages and will remove the correct nodes from *proposers*. Past that time and until some correct node will send a "Propose-Pulse" in the future, no correct node will send a "Support-Pulse" message. Similarly, past $t_0 + 6d$ and until some correct node will send a "Propose-Pulse" in the future no correct node will invoke a SS-BYZ-AGREE in Line D3.

At that time the range of *cycle_countdown* may be in $[Cycle - \Delta_{BYZ}, Cycle]$, since, by the Timeliness-Agreement property (1d), faulty nodes may bring the correct nodes to decide on values that are at most Δ_{BYZ} in the past.

Until $t_0 + 6d + \Delta_{BYZ}$, correct nodes may still decide on values from other agreements invoked by faulty nodes. By the Timeliness-Agreement property (1c), until some correct node will invoke a SS-BYZ-AGREE , no correct node will happen to decide on any message with $rt(\tau') \geq t_0 + 6d$ (latest possible recording time).

Let t_q be the latest real-time a correct node q updated the calculation of *cycle_countdown* at some time during the interval $[t_0, t_0 + 6d]$. It will send its next "Propose-Pulse" message at $t_q + cycle\_countdown = t_q + Cycle - (t_q - rt(latest\_accept_q)) = Cycle + rt(latest\_accept_q)$. By Timeliness-Agreement property (1d), and because the computation of $latest\_accept_q$ takes place in the interval $[t_0, t_0 + 6d]$ we conclude that interval $rt(latest\_accept_q) \geq t_0 + Cycle - \Delta_{BYZ}$. Thus,

$t_q + \mathit{cycle\_countdown} \geq t_0 + \mathit{Cycle} - \Delta_{\text{BYZ}}$. Thus, the earliest time a correct node will send a "Propose-Pulse" message will be at $t_0 + \mathit{Cycle} - \Delta_{\text{BYZ}}$. Until that time no correct node will send a "Propose-Pulse" or "Support-Pulse" message or invoke SS-BYZ-AGREE, proving (1) and (2).

The bound on *Cycle*, implies that during the real-time interval $[t_0 + 6d, t_0 + \mathit{Cycle} - \Delta_{\text{BYZ}}]$ there is a window of at least $14d - 6d - d > 3d$ with no recording time that refers to it. Denote this interval by $[t', t' + B']$, where $B\prime \leq t_0 + \mathit{Cycle} - \Delta_{\text{BYZ}} \leq t_0 + \mathit{Cycle} - d$. The above argument implies that in the interval $[t' + B', t_0 + \mathit{Cycle} - d]$ no correct node will update its *latest_accept*, and therefore the conditions of Lemma 7.2.7 hold.

Thus, the *cycle_countdown* counters of all correct nodes expire within $5d$ past time $t_0 + \mathit{Cycle} - d$. Looking back at the latest real-time, t_q, at which a correct node q updated the calculation of *cycle_countdown*, since it took place in the interval $[t_0 - d, t_0 + 6d]$ and that $rt(\mathit{latest\_accept}_q)$ cannot be larger than the time at which it is computed, the node will send its next "Propose-Pulse" message at $t_q + \mathit{cycle\_countdown} = t_q + \mathit{Cycle} - (t_q - rt(\mathit{latest\_accept}_q)) = \mathit{Cycle} + rt(\mathit{latest\_accept}_q) \leq t_0 + \mathit{Cycle} + 6d$. Proving (3).

□

Corollary 7.2.11 *In the conditions of Lemma 7.2.10, if no correct node invoked* SS-BYZ-AGREE *in the interval* $[t_0 - \Delta_{\text{BYZ}}, t_0 - B]$ *then the bound of* $t_0 + \mathit{Cycle} - \Delta_{\text{BYZ}}$ *in Lemma 7.2.10 can be replaced by* $t_0 + \mathit{Cycle} - B - 2d$.

Proof: By the Timeliness-Agreement property (1c) no decision can return a recording time that is earlier by more than $2d$ from an invocation of SS-BYZ-AGREE by a correct node. Therefore, in the proof of Lemma 7.2.10 the minimal value for $\mathit{latest\_accept}_q$ for any correct node q can be $t_0 - B - 2d$. Let t_q be the latest real-time a correct node q updated the calculation of *cycle_countdown* in the interval $[t_0, t_0 + 6d]$. It will send its next "Propose-Pulse" message at $t_q + \mathit{cycle\_countdown} = t_q + \mathit{Cycle} - (t_q - rt(\mathit{latest\_accept}_q)) = \mathit{Cycle} + rt(\mathit{latest\_accept}_q) \geq \mathit{Cycle} - B - 2d$. Thus, the earliest real-time a correct node will send "Propose-Pulse" message will be at $t_0 + \mathit{Cycle} - B - 2d$. Until that time no correct node will send a "Propose-Pulse" or "Support-Pulse" message or invoke SS-BYZ-AGREE. Thus the bound of $t_0 + \mathit{Cycle} - \Delta_{\text{BYZ}}$ in Lemma 7.2.10 can be replaced by $t_0 + \mathit{Cycle} - B - 2d$. □

We can now state the "fixed-point" lemma:

Lemma 7.2.12 *If the cycle_countdown counters of all correct nodes expire in the period* $[t_0, t_0 + 5d]$ *and no correct node sent "Support-Pulse" in* $[t_0 - (\mathit{Cycle} - 8d), t_0]$ *and no correct node invoked* SS-BYZ-AGREE *in* $[t_0 - (\Delta_{\text{BYZ}} + 6d), t_0]$ *then:*

1. *All correct nodes invoke a pulse within* $3d$ *real-time units of each other before* $t_0 + 9d$*;*
2. *There exists a real-time* $\bar{t}_0$, $t_0 + \mathit{Cycle} - 2d \leq \bar{t}_0 \leq t_0 + \mathit{Cycle} + 12d$ *for which the conditions of Lemma 7.2.12 hold by replacing* t_0 *with* $\bar{t}_0$.

Proof: Assume first that a correct node decided in $[t_0, t_0 + 6d]$. Let q be the first such correct node to do so, at some real-time t_q. By the Timeliness-Agreement property (1c), and since no correct node has invoked SS-BYZ-AGREE in $[t_0 - (\Delta_{\text{BYZ}} + 6d), t_0]$, the recording time needs to be in the interval $[t_0 - 2d, t_q]$. By Timeliness-Agreement property (1a), in the interval $[t_q, t_q + 3d]$ all correct

nodes will decide, and the decision of all correct nodes will imply updating of *latest_accept* and the conditions for invoking a pulse hold.

Moreover, in the interval $[t_q, t_q + 3d]$ the preconditions of Lemma 7.2.10 holds. Using Corollary 7.2.11 for $B = 0$ we obtain the bounds of no "Support-Pulse" in $[t_q + 6d, t_q + \mathit{Cycle} - 2d]$, an interval of $\mathit{Cycle} - 8d$, and all *cycle_countdown* expire within $5d$ in the interval $[t_q + \mathit{Cycle} - 2d, t_q + \mathit{Cycle} + 6d]$. Since $t_q \in [t_0, t_0 + 6d]$, we conclude that for $\bar{t}_0 \in [t_q + \mathit{Cycle} - 2d, t_q + \mathit{Cycle} + 12d]$ the conditions of the lemma hold.

Otherwise, no correct node decided in $[t_0, t_0 + 6d]$. This implies that all correct nodes will end up sending their "Propose-Pulse" by $t_0 + 5d$ and a correct node will send "Support-Pulse" by $t_0 + 6d$. Lemma 7.2.8 completes the proof in a similar way. □

Observe that once Lemma 7.2.12 holds, it will hold as long as the system is coherent, since its preconditions continuously hold. So to complete the proof of the theorem we need to show that once the system becomes coherent, the preconditions of Lemma 7.2.12 will eventually hold.

Denote by $\tilde{t}$ the real-time at which the system became coherent. By Lemma 7.2.6 by $\tilde{t} + \mathit{Cycle} + 4d$ all correct node executes Line E1. Let t_1 be some real-time in that period by which all correct nodes executed Line E1. If any correct node sends a "Support-Pulse" message, then by Lemma 7.2.8 the precondition to Lemma 7.2.12 hold.

Assume otherwise. By the Timeliness-separation property there are no concurrent agreements associated with the same sender of "Support-Pulse" message. Since the separation between decisions is at least $5d$, every correct node will be aware of a decision before invoking the next SS-BYZ-AGREE and therefore, the test in Line D2 will eliminate having more than a single decision per sending node within $\mathit{Cycle} - 11d$. Since $\mathit{Cycle} > (10f + 16)d$, there will be at most f decisions between t_1 and $t_1 + \mathit{Cycle} - 11d$. Since each decision returns recording times to nodes that range over at most $5d$ real-time window, there should be a real-time interval $[t_2, t_2 + 5d]$, that no recording time of any correct node refers to any real-time within it. Note that $t_2 \leq t_1 + \mathit{Cycle} - 11d - 5d \leq \tilde{t} + 2 \cdot \mathit{Cycle} - 12d$. By Lemma 7.2.9, by $\tilde{t} + 2 \cdot \mathit{Cycle} - 12d + \mathit{Cycle} + 4d \leq \tilde{t} + 3 \cdot \mathit{Cycle} - 8d$ there exist a t_3 such that all correct nodes sends "Reset" in the interval $[t_3 - 3d, t_3]$. Thus, the preconditions to Lemma 7.2.10 hold. Thus, by $\tilde{t} + 3 \cdot \mathit{Cycle} - 8d - 3d + \mathit{Cycle} + 6d = \tilde{t} + 4 \cdot \mathit{Cycle} - 5d$ the preconditions to Lemma 7.2.12 hold because either a correct node has sent "Support-Pulse" before that or from Lemma 7.2.10.

Thus the system converges within less than $4 \cdot \mathit{Cycle}$ from a coherent state. One can save one *Cycle* in the bound by overlapping the first one with the second one when the non-faulty nodes are not being considered correct.

From that time on, all correct nodes will invoke pulses within $3d$ of each other and their next pulse will be in the range stated by Lemma 7.2.12. The Lemma immediately implies that the bound on $cycle_{max}$ is $\mathit{Cycle} + 9d$. Similarly, it claims that past $t_0 + 9d$ no "Propose-Pulse" will be sent before $t_0 + \mathit{Cycle} - 2d$, thus potentially the shortest time span between pulses at a node is $\mathit{Cycle} - 11d$. This implies that $cycle_{min} = \mathit{Cycle} - 11d$. Moreover, the discussion also implies that:

Lemma 7.2.13 *Once the conditions of Lemma 7.2.12 hold, no correct node will invoke more than a single pulse in every $cycle_{min}$ real-time interval. It will invoke at least one pulse in every $cycle_{max}$ real-time interval.*

This concludes the Convergence requirement with $\sigma = 3d$, since the correct nodes will always invoke pulses within $3d$ real-time units of each other. This completes the proof of Theorem 7.2.1. □

Theorem 7.2.2 *(*Closure*) If the system is in a synchronized pulse_state at time t_s, then the system is in a synchronized pulse_state at time t, $t \geq t_s$.*

Proof: Let the system be in a synchronized pulse_state at the time immediately following the time the last correct node sent its "Propose-Pulse" message. Thus, all correct nodes have sent their "Propose-Pulse" messages. As a result, all will invoke their pulses within $3d$ of each other, and will reset *cycle_countdown* to be at least $Cycle - 2d$. The faulty nodes may not influence the *cycle* length to be shorter than $cycle_{min}$ or longer than $cycle_{max}$. □

Thus we have proved the main theorem:

Theorem 7.2.3 (Convergence and Closure) *The* AB-PULSE-SYNCH *algorithm solves the Self-stabilizing Pulse Synchronization Problem if the system remains coherent for at least* 4 *cycles.*

Proof: Convergence follows from Theorem 7.2.1. The first Closure condition follows from Theorem 7.2.2. The second Closure condition follows from Lemma 7.2.13. □

Since we defined non-faulty to be considered correct within 2 cycles, we conclude:

Corollary 7.2.14 *From an arbitrary state, once the network become correct and $n - f$ nodes are non-faulty, the* AB-PULSE-SYNCH *algorithm solves the Self-stabilizing Pulse Synchronization Problem if the system remains so for at least* 6 *cycles.*

Lemma 7.2.15 *(*Join of recovering nodes*) If the system is in synchronized state, a recovered node becomes synchronized with all correct nodes within Δ_{node} time.*

Proof: The proof follows the arguments used in the proofs leading to Theorem 7.2.3. Within a *cycle* of non-faulty behavior of the recovering node it clears its variable and data structures of old values. Within $cycle_{max}$ it will synchronize with all other correct nodes, though it might not issue a pulse if it issued one in the first *Cycle*. But by the end of Δ_{node} its *cycle_countdown* will synchronize with all the correct nodes and will consequently produce the next pulse in synchrony with them. □

Chapter 8

Conclusions and Discussion

A number of papers have recently postulated on the similarity between elements connected with biological robustness and design principles in engineering [4, 49]. In the current work we have observed and understood the mechanisms for robustness in a comprehensible and vital biological system. We have then shown how to make specific use of analogies of these elements in distributed systems in order to attain high robustness in a practical manner. The same level of robustness has not been practically achieved earlier in distributed systems. We postulate that our result elucidates the feasibility and adds a solid brick to the motivation to search for and to understand biological mechanisms for robustness that can be carried over to computer systems.

Possible implications in biology: The shape of the neurobiological refractory function is largely perceived to be determined by the physical ionic properties in the single cell only and not due to factors related to robustness and network topology. Our algorithm which generalizes a neurobiological model does show dependence of the shape of the refractory function on factors related to robustness and network topology. We may thus speculate on possible new meanings of the refractory function in the biological context. Could there possibly be some correlation between the number of faults and their severity that the biological neuronal network must face and the network size required to reach synchronization? Could there be some relationship between the shape of the neuronal refractory function and the size of the biological neural network? Does the shape of the refractory function of the neurons play a role in the networks ability to tolerate faults or noise? Does the shape of the refractory function of the neurons play a role in the networks ability to alter the synchronized firing pace?

Refractory-like behavior has not been suggested earlier to have a role in models for biological synchronization. Our work may imply that it can have a pivotal role. We thus suggest that refractory-like behavior may play a more general role in synchronization, and thus may be an important element for synchronization in other biological systems such as in the fireflies, synchronized clapping, synchronized swimming of schools of fish, etc. For example, a person may be less reluctant to reset the clapping immediately following her clapping, but be increasingly receptive to such a reset as the end of the interval for a new clap becomes imminent.

Possible implications in distributed systems: This work showed for the first time that practical linear-time algorithms may be developed in the confluence of the self-stabilization and Byzantine fault models, without relying on any external means for synchronization. The practical impli-

cation of this is the possibility to design highly robust algorithms for situations in which transient incidences might temporarily corrupt the workability of the system, but leave a bounded fraction of the system permanently corrupt. An example of this may be unmanned space missions, in which it might not be possible to reset the whole system following severe faults or even to detect such a situation. The motivation to develop solutions operating in this fault domain is underscored by the recent interest of bodies such as NASA in handling such scenarios [59, 58].

To state the implications in a more generalized or "down-to-earth" manner, our results enables to design practical algorithms without needing to a-priori characterize the severity or nature of the faults, or for a limited period of time, the extent of the faults. We further postulate this may simplify the model statements in which an algorithm operates, which in turn may decrease the possibility to incorrectly comprehend the model when implementing an algorithm.

Bibliography

[1] Y. Afek, S. Dolev. Local stabilizer. *Proc. of the 5th Israeli Symposium on Theory of Computing Systems (ISTCS97)*, Bar-Ilan, Israel, Jun 1997.

[2] D. Agrawal, A. E. Abbadi. A token-based fault-tolerant distributed mutual exclusion algorithm source. *Journal of Parallel and Distributed Computing archive*, 24(2):164–176, Feb 1995.

[3] G. Alari, J. Beauquier, A. Datta, C. Johnen, V. Thiagarajan. Fault-tolerant token passing algorithm on tree networks. *IEEE International Performance Computing and Communications Conference (IPCCC'98)*, 1998.

[4] U. Alon, M. Surette, N. Barkai, S. Leibler. Robustness in bacterial chemotaxis. *Nature*, 397(6715):168–171, Jan 1999.

[5] E. Anagnostou, V. Hadzilacos. Tolerating transient and permanent failures. *Proc. of the 7th International Workshop on Distributed Algorithms*, Les Diablerets, Switzerland, Sep 1993.

[6] E. Anceaume, I. Puaut. Performance evaluation of clock synchronization algorithms. Technical Report 3526, INRIA, 1998.

[7] A. Arora, S. Dolev, , M. Gouda. Maintaining digital clocks in step. *Parallel Processing Letters*, 1:11–18, 1991.

[8] A. Arora, M. Gouda. Distributed reset. *In Proceedings of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science*. Lecture Notes in Computer Science, 1990.

[9] A. Arora, S. Kulkarni. Component based design of multitolerance. *IEEE Transactions on Software Engineering*, 24(1):63–78, Jan 1998.

[10] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, G. Varghese. Time optimal self-stabilizing synchronization. *Proc. of the 25th Symp. on Theory of Computing*, 1993.

[11] B. Awerbuch, B. Patt-Shamir, G. Varghese. Self-stabilization by local checking and correction. *In Proceedings of the 32nd IEEE Symp. on Foundation of Computer Science*, 1991.

[12] J. Beauquier, S. Kekkonen-Moneta. Fault-tolerance and self-stabilization: Impossibility results and solutions using failure detectors. *Int. J of Systems Science*, 28(11):1177–1187, 1997.

[13] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. *Proc. of the Second Annual ACM Symposium on Principles of Distributed Computing*, strony 27–30, Canada, Aug 1983.

[14] J. Brzeziǹski, M. Szychowiak. Self-stabilization in distributed systems - a short survey. *Foundations of Computing and Decision Sciences*, 25(1), 2000.

[15] J. Buck, E. Buck. Synchronous fireflies. *Scientific American*, 234:74–85, May 1976.

[16] R. W. Buskens, R. P. Bianchini. Self-stabilizing mutual exclusion in the presence of faulty nodes. *In Proceedings of the 25th International Symposium on Fault-Tolerant Computing Systems (FTCS'95)*, 1995.

[17] K. M. Chandy, L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Computer Systems*, 9(1):63–75, 1985.

[18] B. Coan, D. Dolev, C. Dwork, L. Stockmeyer. The distributed firing squad problem. *Proc. of the 7th Annual ACM Symposium on Theory of Computing*, Providence, Rhode Island, May 1985.

[19] A. Daliot, D. Dolev. Self-stabilization of byzantine protocols. *In Proc. of the 7th Symposium on Self-Stabilizing Systems (SSS'05)*, Barcelona, Spain, Oct 2005.

[20] A. Daliot, D. Dolev. Self-stabilizing byzantine agreement. *In Proc. of the Twenty-fifth ACM Symposium on Principles of Distributed Computing (PODC'06)*, Denver, Colorado, Jul 2006.

[21] A. Daliot, D. Dolev, H. Parnas. Linear time byzantine self-stabilizing clock synchronization. *Proc. of 7th Int. Conference on Principles of Distributed Systems (OPODIS'03)*, La Martinique, France, Dec 2003. A corrected version appears in http://arxiv.org/abs/cs.DC/0608096.

[22] A. Daliot, D. Dolev, H. Parnas. Self-stabilizing pulse synchronization inspired by biological pacemaker networks. *In Proceedings of the Sixth Symposium on Self-Stabilizing Systems (DSN SSS '03)*, LNCS 2704, San Francisco, Jun 2003. A full and revised version appears in http://arxiv.org/abs/0803.0241.

[23] W. Dijkstra. Self-stabilization in spite of distributed control. *Commun. of the ACM*, 17:643–644, 1974.

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

[25] D. Dolev, J. Halpern, H. R. Strong. On the possibility and impossibility of achieving clock synchronization. *J. of Computer and Systems Science*, 32(2):230–250, 1986.

[26] D. Dolev, J. Y. Halpern, B. Simons, R. Strong. Dynamic fault-tolerant clock synchronization. *J. Assoc. Computing Machinery*, 42(1):143–185, Jan 1995.

[27] D. Dolev, N. A. Lynch, E. Stark, W. E. Weihl, S. Pinter. Reaching approximate agreement in the presence of faults. *J. of the ACM*, 33:499–516, 1986.

[28] D. Dolev, H. R. Strong. Polynomial algorithms for multiple processor agreement. *In Proc. of the 14th ACM SIGACT Symposium on Theory of Computing (STOC-82)*, May 1982.

[29] D. Dolev, H. R. Strong. Authenticated algorithms for byzantine agreement. 12(4):656–666, 1983.

[30] S. Dolev. Possible and impossible self-stabilizing digital clock synchronization in general graphs. *Journal of Real-Time Systems*, 12(1):95–107, 1997.

[31] S. Dolev. *Self-Stabilization*. The MIT Press, 2000.

[32] S. Dolev, A. Israeli, S. Moran. Uniform dynamic self-stabilizing leader election. *IEEE Trans. Parallel and Distributed Systems*, 8(4):424–440, 1997.

[33] S. Dolev, J. L. Welch. Wait-free clock synchronization. *Algorithmica*, 18(4):486–511, 1997.

[34] S. Dolev, J. L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults. *Journal of the ACM*, 51(5):780–799, 2004.

[35] P. Dutta, R. Guerraoui, L. Lamport. How fast can eventual synchrony lead to consensus? *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN'05)*, Yokohama, Japan, Ju 2005.

[36] M. J. Fischer, N. A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, 1982.

[37] M. J. Fischer, N. A. Lynch, M. Merritt. Easy impossibility proofs for ddistributed consensus problems. *Distributed Computing*, 1:26–39, 1986.

[38] M. J. Fischer, N. A. Lynch, M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. of the ACM*, 32(2):374–382, 1985.

[39] F. C. Freiling, S. Ghosh. Code stabilization. *Proc. of the 7th Symposium on Self-Stabilizing Systems (SSS'05)*, Barcelona, Spain, Oct 2005.

[40] W. O. Friesen. Physiological anatomy and burst pattern in the cardiac ganglion of the spiny lobster panulirus interuptus. *J. Comp. Physiol.*, 101:173–189, 1975.

[41] W. O. Friesen. Synaptic interaction in the cardiac ganglion of the spiny lobster panulirus interuptus. *J. Comp. Physiol.*, 101:191–205, 1975.

[42] H. Garcia-Molina. Elections in a distributed computing system. *IEEE TRANS. COMP.*, C-13(1):48–59, 1982.

[43] S. Ghosh, A. Gupt. An exercise in fault-containment: self-stabilizing leader election. *Information Processing Letters*, 59:281–288, 1996.

[44] A. S. Gopal, K. J. Perry. Unifying self-stabilization and fault-tolerance. *IEEE Proceedings of the 12th annual ACM symposium on Principles of distributed computing*, Ithaca, New York, 1993.

[45] T. Herman. Phase clocks for transient fault repair. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1048–1057, 2000.

[46] A. Israeli, M. Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. *Proc. of the 9th annual ACM symposium on Principles of distributed computing (PODC'90)*, Quebec, Canada, 1990.

[47] S. Katz, K. J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, 1993.

[48] J. Kiniwa. Avoiding faulty privileges in self-stabilizing depth-first token passing. *Proceedings of the Eighth International Conference on Parallel and Distributed Systems (ICPADS'01)*, 2001.

[49] H. Kitano. Biological robustness. *Nature*, 5, Nov 2004.

[50] C. Koch. *Biophysics Of Computation: Information Processing In Single Neurons*. Oxford University Press, 2004.

[51] S. Kulkarni, A. Arora. Compositional design of multitolerant repetitive byzantine agreement. *Proceedings of the 18th Int. Conference on the Foundations of Software Technology and Theoretical Computer Science*, India, 1997.

[52] S. Kulkarni, A. Arora. Multitolerance in distributed reset. *Chicago Journal of Theoretical Computer Science, Special Issue on Self-Stabilization*, 1998.

[53] S. Kutten, B. Patt-Shamir. Time-adaptive self stabilization. *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC97)*, 1997.

[54] L. Lamport, R. Shostak, M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–301, 1982.

[55] B. Liskov. Practical use of synchronized clocks in distributed systems. *Proceedings of 10th ACM Symposium on the Principles of Distributed Computing*, 1991.

[56] J. Lundelius, N. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62:190–205, 1984.

[57] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[58] M. R. Malekpour. A byzantine-fault tolerant self-stabilizing protocol for distributed clock synchronization systems. *Proc. of 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'06)*, Dallas, Texas, Nov 2006.

[59] M. R. Malekpour, R. Siminiceanu. Comments on the byzantine self-stabilizing pulse synchronization protocol: Counterexamples. Technical Memorandum NASA-TM213951, NASA, Feb 2006. http://hdl.handle.net/2002/16159.

[60] T. Masuzawa, S. Tixeuil. A self-stabilizing link-coloring protocol resilient to unbounded byzantine faults in arbitrary networks. Technical Report 1396, Laboratoire de Recherche en Informatique, Jan 2005.

[61] R. E. Mirollo, S. H. Strogatz. Synchronization of pulse-coupled biological oscillators. *SIAM J. Appl. Math*, 50:1645–1662, 1990.

[62] S. Mishra, S. Srimani. Fault-tolerant mutual exclusion algorithms. *P. K. Journal of Systems and Software*, 11(2):111–129, 1990.

[63] Z. Nèda, E. Ravasz, Y. Brechet, T. Vicsek, A. L. Barabàsi. Self-organizing process: The sound of many hands clapping. *Nature*, 403:849–850, 2000.

[64] M. Nesterenko, A. Arora. Dining philosophers that tolerate malicious crashes. *22nd Int. Conference on Distributed Computing Systems*, 2002.

[65] M. Nesterenko, A. Arora. Local tolerance to unbounded byzantine faults. *21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, 2002.

[66] M. Papatriantafilou, P. Tsigas. On self-stabilizing wait-free clock synchronization. *Parallel Processing Letters*, 7(3):321–328, 1997.

[67] H. Parnas, E. Sivan. *SONN - Simulator of Neuronal Networks*. Hebrew University, Jerusalem, Israel, 1996. http://www.ls.huji.ac.il/∽parnas/Sonn2/sonn.html.

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

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

[70] E. Sivan, H. Parnas, D. Dolev. Fault tolerance in the cardiac ganglion of the lobster. *Biol. Cybern.*, 81:11–23, 1999.

[71] S. D. Stoller. Detecting global predicates in distributed systems with clocks. *Distributed Computing*, 13(2):85–98, 2000.

[72] S. H. Strogatz, I. Stewart. Coupled oscillators and biological synchronization. *Scientific American*, 269:102–109, Dec 1993.

[73] S. Toueg, K. J. Perry, T. K. Srikanth. Fast distributed agreement. *SIAM Journal on Computing*, 16(3):445–457, Jun 1987.

[74] T. J. Walker. Acoustic synchrony: two mechanisms in the snowy tree cricket. *Science*, 166:891–894, 1969.

[75] J. L. Welch, N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77:1–36, 1988.

[76] J. Widder. Booting clock synchronization in partially synchronous systems. *In Proc. the 17th Int. Symposium on Distributed Computing (DISC'03)*, Sorrento, Italy, Oct 2003.