Bridging the Gap: Byzantine Faults and Self-stabilization

Thesis submitted for the degree of
DOCTOR of PHILOSOPHY

by

Ezra N. Hoch

Submitted to the Senate of

The Hebrew University of Jerusalem

September 2010

This work was carried out under the supervision of:

Prof. Danny Dolev.

Acknowledgements

As someone I know once said, the acknowledgements section is the only real part of the PhD thesis that is actually read. So I would like to take a moment, and thank those that helped me on this adventure.

First, I would like to thank both of my parents, but not just because I wouldn't be here without them. I grew up in a house which appreciated education and encouraged knowledge. My parents took special care to ensure that I get a good eduction, by caring for extra curriculum lessons during junior high, and especially by choosing the high school I went to. All in all, I think that my love for learning stems from the choices my parents made way way back.

Second, I thank my advisor, professor Danny Dolev. Luckily, I had the pleasure and honor to have Danny as my advisor for both my MA and PhD. It all started on the last year of my BA. I took the distributed algorithms course (taught by Danny) and was amazed: it was so interesting, and so challenging. Danny teaches the course in a way that students are encouraged to participate and to think about solutions by themselves. After three lessons, I was so fascinated by the subject I asked Danny to tutor me on a project; which lead, after several years, to this thesis. During all that time, Danny helped me evolve. He has this very special ability to guide in a way that is both supportive and independent. He always gave me the freedom of choosing what problems to work on, while commenting and suggesting directions which helped me focus. For me, Danny was the perfect advisor: combining successfully both having the time to meet and discuss research issues and providing the leeway so many times required during a PhD.

Third, I learned a lot from the people I collaborated with: professor Michael Ben-Or, Dr. Danny Bickson, Dr. Ariel Daliot and professor Yoram Moses. I would like to specially thank Michael: in the last two years I worked a lot with Michael, and I truly appreciate the amount of time he spent with me. Michael introduced me to the wonders of randomization in distributed algorithms. The more I learn about the subject, and the more I read papers on the subject, the more I'm astonished and bewildered at the beautiful and intricate ideas that govern this field. I was fortunate to have the chance to study this area, and more so, to be acquainted to it by a researcher like Michael.

Fourth, I want to thank my friends. From the DANSS group, from other floors in Ross building and those from outside the campus grounds. The last

few years were hard at times and joyous at others. In both cases, it is good to have friends to share it with. In addition, I think I should apologize to all those for whom I've wasted so many research hours in conversations, discussions, and just updates on day-to-day life.

Last, but definitely not least, I owe so much to my (newlywed) wife. Daphna has seen, at least once, every single presentation I ever gave during my PhD. She has understandingly accepted all the evenings I had to work on during deadline periods, and all the ups-and-downs immanent to the peer-review process. She listened to new ideas, when they were still immature; and her questions helped shape those ideas. These are just the direct contributions Daphna had to my PhD; there are infinitely many indirect ones.

Abstract

Distributed systems are everywhere. As everyday lives become more and more dependent on distributed systems, they are expected to withstand different kinds of failures. Different models of failures exist which aim at modeling network errors, hardware failures, soft errors, etc. This thesis concentrates on dealing with a combination of two different kinds of fault tolerance.

The Byzantine failure model aims at modeling hardware failures. A system is said to be tolerant to Byzantine failures if it can withstand any arbitrary behavior of (usually) up to a constant percentage of its nodes. That is, a constant percentage of the nodes may behave in any way and may collude together to try and prevent the system from operating correctly. Clearly, if a system is Byzantine tolerant then it will be able to overcome hardware failures, as long as enough nodes operate correctly.

The self-stabilizing failure model aims at modeling soft errors. A system is said to be self-stabilizing if starting from any memory state, it will eventually converge to an operating condition. For example, a self-stabilizing clock synchronization will eventually (even if it is started when different nodes are out of sync) have all nodes in the system agree on the same time. If a system is self-stabilizing and a soft error occurred, then eventually (if there are no more soft errors) it will converge to an operational state.

Combining self-stabilizing and Byzantine failures is a challenging task. It requires the algorithm to tolerate both external errors (*i.e.,* neighboring nodes which are Byzantine) and internal errors (*i.e.,* soft errors which may lead to an arbitrary memory state). However, an algorithm that is both self-stabilizing and Byzantine tolerant has highly desired robust properties: Even if the assumed working conditions are invalidated, when they are eventually re-guaranteed the algorithm will converge to an operating state.

Three different problems are discussed herein. First, the self-stabilizing Byzantine tolerant clock synchronization problem; in which the system may start in any arbitrary state, and eventually (in the ongoing presence of Byzantine nodes) all nodes should agree on the same clock value, and increase it regularly. Two different solutions to this problem are given. One operates in a synchronous network and provides an expected constant convergence rate; while the other operates in a semi-synchronous (sometimes denoted "bounded-delay") network and provides a (deterministic) linear convergence time.

Second, the stability of a system's output is investigated. Specifically,

assume that a system has some changing input which is transformed (via some computation) into the output of the system. In addition, suppose that eventually the input does not change (for example, since the temperature in a server room becomes stable), what can be said on the output? Can it be guaranteed to stabilize? A self-stabilizing Byzantine tolerant algorithm that solves this problem in an optimal way is given.

The third problem considers a variation of the Byzantine model. The classical Byzantine model assumes any subset of nodes may fail, as long as the subset is of limited size. Such a model is less useful when considering failures that are locally bounded. For example, suppose there is some manufacturing error causing every tenth sensor to be faulty. In addition, suppose that the sensors are dispersed from an airplane in a uniform manner. The hardware failure distribution is most likely not to concentrate at a specific location. When considering a Byzantine adversary that is locally confined (*i.e.,* it is limited by the number of nodes it can control in every local area), a solution to the Byzantine consensus problem is achieved, which is more efficient than the classical one, both in space, time and message complexities.

Contents

Chapter 1

Introduction

The field of fault tolerance in distributed systems branches in many directions, a few of which are the subject of this thesis. Specifically, self-stabilization and Byzantine tolerance. Informally, a self-stabilizing system is a system which can start in any initial state and converge to an operating state. A Byzantine tolerant system can continue operating while some of the nodes are faulty and try to "bring the system down". A system that is both self-stabilizing and Byzantine tolerant is one which starting from any state, in the continuous presence of Byzantine nodes, will eventually convergence to an operating state. The main results of this thesis are algorithms which are both self-stabilizing and Byzantine tolerant.

When designing fault-tolerant algorithms there are many pitfalls and obstacles to overcome. Modular planing of algorithms using fault-tolerant building blocks helps construct solutions and helps in the correctness proofs (it is not uncommon for self-stabilizing algorithms to be short, while their proofs are long and intricate). A main focus of this thesis is to create such building blocks, that will help in simplifying the building of fault-tolerant algorithms, and consequently increase the fault tolerance of distributed systems. In particular, the thesis focuses on the fundamental problem of agreement / synchronization (either on a specific value or on the current time) in distributed systems, while withstanding both self-stabilizing and Byzantine failures.

1.1 Background and Related Work

1.1.1 Timing Models

There are 3 major time models used in distributed algorithms: synchronous, asynchronous and semi-synchronous (also termed bounded-delay). (See [17] for a formal definition of different timing models). In the synchronous model nodes advance in synchronous rounds. *I.e.,* all nodes start a round by sending their messages, the messages are received by their recipients, and then the round ends and the following round starts. Each node p receives all of the i-th round messages together and is aware that these messages correspond to the i-th round.

In the semi-synchronous model (bounded-delay) there is an a priori bound d on the time a message takes to reach its recipient. *I.e.,* if node p sends a message to node q, it is guaranteed to arrive at q after no more than d time units. Usually all nodes are assumed to be able to measure time intervals, thus allowing node p to send a message to node q and reason that if q returns a message immediately when receiving a message, then p should have received a message back within $2 \cdot d$ time units.

In the asynchronous model there is no bound on message delivery time. The only guarantee is that every message that is sent is eventually delivered. Usually in an asynchronous setting nodes are oblivious of passing time and are event-driven in the sense that their operations occur due to receiving messages.

The algorithms presented herein operate in the synchronous and semi-synchronous models: Chapter 2 operates in the semi-synchronous model while Chapter 3, Chapter 4 and Chapter 5 all operate in the synchronous model.

1.1.2 Byzantine Failures

Byzantine failures aim to model any arbitrary node failure. A system is said to be Byzantine tolerant if it reaches its goal while some of the nodes are Byzantine - they can behave in any way (even collude together) to prevent the system from acquiring its target.

Since the first appearance of Byzantine failures in the literature [21, 16], they have become a central failure model in distributed computing. The most known problem relating to Byzantine failures is the Byzantine agreement (BA

for short) problem. In the BA each node p of the n nodes in the system has an input value v_p and an output value o_p. There are two requirements:

1. all non-faulty nodes agree on the same output value (*i.e.,* $o_p = o_q$ for every non-faulty p, q);

2. the agreed output value is an input value of some non-faulty node.

The BA problem has been thoroughly researched and produced a plethora of results, including many lower bounds: $O(f)$ rounds lower bound [14] where f is the number of Byzantine nodes, a lower bound on the number of Byzantine nodes [21, 12] (f must be less than $\frac{1}{3}n$), lower bounds on the connectivity requirements [7] (there must be at least $2f + 1$ distinct routes between any two non-faulty nodes), *etc.*

In addition, BA has been shown to be impossible in asynchronous systems [13]. This impossibility result can be overcome by adding randomization [1, 23]. Randomization is used to implement a distributed shared-coin primitive, which every round has probability c to produce the same random bit at all non-faulty nodes. Using a shared-coin it is possible to solve BA within expected $O(\frac{1}{c})$ rounds.

When considering randomized algorithms it is interesting to differentiate between a few classes of Byzantine adversaries. The adversary can be static, in which case it must choose the Byzantine nodes before the algorithm starts; or the adversary can be dynamic, in which case it can choose the Byzantine nodes during the execution (after it has seen the outcome of the coin flips). Another classification is whether the adversary has full-information (*i.e.,* the adversary knows the state of each node), or there are some pieces of information which the adversary does not have access to. Specifically, in Chapter 3 private-channels are assumed; *i.e.,* the adversary cannot read the state of the non-faulty nodes and cannot read the content of messages sent between non-faulty nodes.

The above categories lead to a few classes of adversaries: dynamic full-information (such as in [1]), dynamic with private-channels (such as in [11]), static full-information (such as in [20]). Chapter 3 assumes the same Byzantine adversary model as in [11].

In deterministic algorithms (such as [21, 16]) there is no difference between the above classes, since the adversary also controls the input values and together with the determinism of the algorithm, the state of each node can be concluded. Thus, there is no difference between full-information and

non-full-information and between dynamic or static adversaries. Chapter 2, Chapter 4 and Chapter 5 all operate in a deterministic setting.

1.1.3 Self-stabilization

Self-stabilization models transient failures. A system is self-stabilizing if starting from any arbitrary state it eventually reaches a state in which it operates correctly. More specifically, a state of a system is represented by the state of each node in addition to the state of the message buffers (or other communication devices). A self-stabilizing system, starting from any arbitrary state, will eventually reach a state in which it operates according to its specification. The motivation behind self-stabilizing systems is to handle soft errors such as buffer overruns, electronic interference and other transient failures. If a system is indeed self-stabilizing then once there are no more transient errors (*i.e.,* the electronic interference has ceased), the system will converge to an operating state.

Usually, self-stabilizing systems exhibit two properties: closure and convergence. Convergence ensures that starting from any initial state, the system will eventually (within a finite number of rounds) converge to a legal state (a state in which it operate correctly). Closure ensures that once the system is in a legal state it will continue to be in legal states (*i.e.,* any execution from a legal state stays in legal states). Thus, convergence and closure together ensure that the system will eventually reach a state from which it operates correctly forever.

The field of self-stabilization started with [6] and flourished there after. See [9] for many examples of self-stabilizing results. Self-stabilizing problems and solutions prevail in many different models: in synchronous, semi-synchronous and asynchronous timing models; in message passing and in shared memory models; in deterministic and randomized algorithms, *etc.* In the current work, Chapter 2, Chapter 3 and Chapter 4 all contain self-stabilizing solutions; while Chapter 5 operates in a non-self-stabilizing setting.

1.1.4 Combining Byzantine Failures and Self-stabilization

Combining Byzantine failures with self-stabilization is an interesting challenge. To illustrate the difficulties that arise in such a setting, consider the problem of digital clock synchronization: each node has a bounded integer

counter which should eventually be the same at all nodes and increase by one (modulo the size of the counter) in every round. In the following discussion, assume the network is synchronous and fully connected (*i.e.,* every pair of nodes can send messages among themselves).

Solving the digital clock synchronization when there are only Byzantine failures is quite simple: each node p starts a Byzantine agreement on its current clock value and at round $f+1$ p will update its clock value to be the median of all the results of the different BAs. Since by round $f+1$ all BAs have terminated, all non-faulty nodes see the same set of values and thus all update their clock in the same way. In all following rounds, p will increase its clock value by one each round. Clearly, the above solves the digital clock synchronization in a Byzantine tolerant manner. Notice that if the nodes start in an arbitrary state, the algorithm immediately stops working: for example, if a node starts while thinking it is in round $f+2$, it might never synchronize its clock value with the other nodes.

Solving the digital clock synchronization in a self-stabilizing manner (when there are no Byzantine failures) is even simpler: every round each node will send its current clock value plus one to all other nodes, and will take the maximum among all received values. Since all nodes are non Byzantine, in every round they all receive the same set of value and thus they always agree on the same clock value. In addition, after the first round the clock value will increase by one each round. Clearly, the above solves the digital clock synchronization in a self-stabilizing fashion. Notice that if there is even a single Byzantine node in the system, it can send different clock value to different nodes, thus ensuring that there is never an agreement on the current clock value.

In the Byzantine tolerant solution, all non-faulty nodes start with a "clean" state, and can thus utilize Byzantine-tolerant primitives to overcome the Byzantine failures. In the self-stabilizing solution, all nodes "know" they can trust the values received from their neighbors, and can thus use a very simple update rule. In other words, in first case non-faulty nodes can trust themselves (but cannot trust their neighbors), while in the second case the nodes can trust their neighbors (but cannot trust themselves).

When combining self-stabilization with Byzantine failures, the nodes in the algorithm cannot trust their neighbors (which might be Byzantine) while they cannot trust themselves (since their initial state might be faulty). Therefore, it is a challenging task to constructing self-stabilizing and Byzantine tolerant algorithms.

Due to the challenges of the field there are few works that combine self-stabilization and Byzantine tolerance [24, 19, 18, 10, 5, 2, 4]. These results can be divided into two classes: algorithms in which nodes that are close to other Byzantine nodes may reach undesired states (*i.e.,* [24, 19, 18]) and algorithms which require all non-Byzantine nodes to behave correctly. In this thesis, all self-stabilizing and Byzantine tolerant algorithms assume a fully connected network. Thus, allowing Byzantine neighbors to be in inconsistent states will invalidate the entire system (hence all following algorithms are of the second class).

1.1.5 Clock Synchronization vs. Byzantine Agreement

Byzantine agreement (BA) is probably the most fundamental problem in distributed algorithm, since it tackles the most basic issue of having more than one computer operate together: synchronization. Specifically, it requires the different nodes to reach agreement on their initial values, while there are permanent Byzantine failures.

Digital clock synchronization, in a sense, is the self-stabilizing equivalent of the BA problem. In self-stabilizing algorithms there is no notion of starting time, and therefore, there is no meaning for "initial values". Thus, the natural extension of BA into the self-stabilizing world is to require agreeing on some value in a non-trivial way. Specifically, the digital clock synchronization problem requires to agree on the current (integer) clock value and increase it every round (which requires a non-trivial solution).

In [5, 8] it is shown how BA can be used to achieve self-stabilizing Byzantine tolerant clock synchronization (SS-Byz-CS, for short). *I.e.,* any algorithm $\mathcal{A}$ that solves BA (in a none self-stabilizing system) within ℓ rounds can be used to solve SS-Byz-CS in a self-stabilizing and Byzantine tolerant manner, with $O(\ell)$ convergence time and with the same connectivity requirements of $\mathcal{A}$.

Moreover, any algorithm $\mathcal{B}$ that solves the SS-Byz-CS problem and converges within ℓ rounds induces a BA algorithm that terminates within $O(\ell)$ rounds (in a none self-stabilizing system). To see why this holds, consider a synchronous algorithm $\mathcal{B}$ that runs is self-stabilizing and supports f Byzantine failures. Therefore, there is some global memory state (*i.e.,* a vector of local memory states) such that if the system starts with s then all non-Byzantine nodes will have clock value "0" (and increase it by one each round); denote this state as s_0. Similarly, there is a global memory state s_1 for the

clock value "1". In addition, since $\mathcal{B}$ solves the SS-Byz-CS problem, there is some value $k = O(\ell)$ such that if $\mathcal{B}$ is run for k rounds, all non-Byzantine nodes agree on the same clock value.

Now, consider the algorithm $\mathcal{A}$, running in a synchronous non-self-stabilizing system: each node p starts with its local view of s_0 or s_1 according to whether p's initial value is "0" or "1". Each node runs $\mathcal{B}$ for k rounds, and considers the clock value at round k: if it is k, return "0"; otherwise, return "1".

Notice the following properties:

1. At the end of $\mathcal{A}$'s execution, all non-Byzantine nodes have the same output value, since they all execute $\mathcal{B}$ for k rounds, and therefore they all see $\mathcal{B}$'s same clock value.

2. If all non-faulty nodes start with initial value "0" then they all agree on the value "0", since if all non-fault nodes start with initial value "0", then they start with memory state s_0, which ensures that the clock value of $\mathcal{B}$ is "0" and therefore, after k rounds, the clock value of $\mathcal{B}$ will be k, and all nodes will return "0" in $\mathcal{A}$.

3. If all non-faulty nodes start with initial value "1" then they all agree on the value "1" (for similar reasons to the case of "0").

The above discussion shows that BA (in a non-self-stabilizing system) and SS-Byz-CS (in a self-stabilizing system) are equivalent and different lower (resp. upper) bounds which apply to BA induce lower (resp. upper) bounds on SS-Byz-CS, and vice versa.

1.1.6 Clock Synchronization vs. Pulse Synchronization

The self-stabilizing Byzantine tolerant pulse synchronization problem (SS-Byz-PS for short) consists of eventually converging to a state where all non-Byzantine nodes pulse together at regular intervals. *I.e.,* at some time all non-Byzantine nodes pulse, then they do not pulse for *Cycle* time, then they all pulse again, and so on.

The SS-Byz-CS problem is equivalent (up to a constant factor) to the SS-Byz-PS problem. In [3] it is first shown that SS-Byz-CS can be achieved by a SS-Byz-PS algorithm. In [8] a more simple solution (for the synchronous case) is given. In Chapter 2 a solution for the bounded-delay model is given.

The above results show that given a SS-Byz-PS algorithm, it is possible to construct a SS-Byz-CS algorithm with similar convergence time and Byzantine resiliency ratio. To conclude the equivalence of SS-Byz-CS to SS-Byz-PS, it is required to show that SS-Byz-CS can be used to solve SS-Byz-PS: suppose $\mathcal{A}$ is an algorithm solving SS-Byz-CS, then construct and algorithm $\mathcal{B}$ that runs $\mathcal{A}$ and pulses every time $\mathcal{A}$'s clock equals "0". Clearly, once $\mathcal{A}$ converges, all non-Byzantine nodes see the same clock value, and therefore, all pulse together exactly when $\mathcal{A}$'s clock is "0", and then do not pulse again until it is again "0".

1.2 Thesis Structure

1.2.1 Byzantine Self-stabilizing Pulse in a Bounded-Delay Model

The work in Chapter 2 operates in the bounded-delay timing model. It investigates the pulse synchronization problem in a way that is both self-stabilizing and Byzantine tolerant. Such a solution can be used to achieve a self-stabilizing Byzantine tolerant clock synchronization algorithm (one which allows for continuous clock values, *i.e.,* not necessarily integer clock values).

The main result builds upon a self-stabilizing Byzantine tolerant agreement protocol (from [4]) and utilizes it to solve the pulsing algorithm. The pulsing algorithm is constructed in two layers: the first one creates an algorithm that has an erratic pulsing pattern. The second layer smooths out the pulsing pattern so that it is regular, as required.

Denote by d the bound on the time it takes a message to reach its destination. The algorithms in Chapter 2 solve the self-stabilizing Byzantine tolerant pulsing problem and achieve pulse tightness of $3d$ (the time difference between two non-Byzantine nodes' pulses). The solution converges within linear time and is resilient up to $\frac{1}{3}n$ Byzantine nodes (*I.e.,* it is optimal in both the convergence time and Byzantine resiliency).

1.2.2 Fast Self-Stabilizing Byzantine Tolerant Digital Clock Synchronization

The work in Chapter 3 operates in a synchronous model, while assuming private channels. It considers the digital clock synchronization problem and presents a probabilistic algorithm that solves the digital clock synchronization problem in a self-stabilizing manner while tolerating up to $\frac{1}{3}n$ Byzantine nodes. The expected convergence rate of the solution is $O(1)$, which is optimal.

Moreover, Chapter 3 contains a building block that produces a stream of random bits in a self-stabilizing manner. That is, eventually it will produce every round, at all non-Byzantine nodes, a new random bit that with constant probability is "0" and with constant probability is "1".

1.2.3 OCD: Obsessive Consensus Disorder (or Repetitive Consensus)

The work in Chapter 4 operates in a synchronous model and the solution is deterministic (thus, the adversary has full-information and is dynamic). The work investigates the stability of iterative multiple consensuses. More specifically, assuming that the input values of the non-Byzantine nodes do not change (over a long period of time), what can be said on the output value of repetitive consensuses?

Chapter 4 contains both upper and lower bounds, one of which shows that the Byzantine adversary can change the output of the consensuses at least once, and the adversary can delay this change indefinitely (*i.e.,* there is no round after which the system's output can be guaranteed not to change). Moreover, the solution is shown to be self stabilizing.

1.2.4 Constant-space Localized Byzantine Consensus

The work in Chapter 5 operates in a synchronous model and the solution is deterministic. The adversarial model assumes the Byzantine nodes are "local"; instead of allowing the adversary to control any node (up to some fraction of n), the adversary is disallowed to control too many nodes in a small area. This model was first suggested in [15, 22], and is motivated by a more realistic approach to Byzantine failures. For example, consider a sensor network scattered over a large area, and assume Byzantine behavior

aims at modeling hardware manufacturing problems. Assuming the sensors are dispersed in some random manner (*i.e.,* by an airplane), it is highly unlikely that the Byzantine nodes will all be concentrated in one location.

The locality restriction on the adversary yields improved solutions to previous problems. Specifically, in Chapter 5 it is shown how Byzantine consensus can be solved (on a family of graphs) in $O(\log n)$ time and $O(n \log n)$ messages.

Chapter 2

Byzantine Self-stabilizing Pulse in a Bounded-Delay Model

Danny Dolev and Ezra N. Hoch, Proceedings of the 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS '07), Paris, France, Pages: 234-252, Nov. 2007.

Byzantine Self-stabilizing Pulse in a Bounded-Delay Model

Danny Dolev* and Ezra N. Hoch

School of Engineering and Computer Science,
The Hebrew University of Jerusalem, Israel
{dolev,ezraho}@cs.huji.ac.il

Abstract. "Pulse Synchronization" intends to invoke a recurring distributed event at the different nodes, of a distributed system as simultaneously as possible and with a frequency that matches a predetermined regularity. This paper shows how to achieve that goal when the system is facing both transient and permanent (*Byzantine*) failures.

Byzantine nodes might incessantly try to de-synchronize the correct nodes. 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 thus cannot use any aspect of their own local states to infer anything about the states of other correct nodes. The algorithm we present here guarantees that eventually all correct nodes will invoke their pulses within a very short time interval of each other and will do so regularly.

The problem of pulse synchronization was recently solved in a system in which there exists an outside beat system that synchronously signals all nodes at once. In this paper we present a solution for a bounded-delay system. When the system in a steady state, a message sent by a correct node arrives and is processed by all correct nodes within a bounded time, say d time units, where at steady state the number of Byzantine nodes, f, should obey the $n > 3f$ inequality, for a network of n nodes.

1 Introduction

When constructing distributed systems, fault tolerance is a major consideration. Will the system fail if part of the memory has been corrupted (e.g. by a buffer overrun)? Will it withstand message losses? Will it overcome network disconnections? To build distributed systems that are fault tolerant to different types of faults, two main paradigms have been used: The Byzantine model and the self-stabilizing model

The *Byzantine* fault paradigm assumes that up to some fraction of the nodes in the system (typically one-third) may behave arbitrarily. Moreover, these nodes can collude in order to try and bring the system down (for more on *Byzantine* faults, see [1]).

* Part of the work was done while the author visited Cornell university. The work was funded in part by ISF, ISOC, NSF, CCR, and AFOSR.

T. Masuzawa and S. Tixeuil (Eds.): SSS 2007, LNCS 4838, pp. 234–252, 2007.

The self-stabilization model assumes that the system might be thrown out of its assumed working conditions for some period of time. Once the system is back to its normal boundaries, all nodes should converge to the desired solution. For example, starting from any memory state, after a finite time, all nodes should have the same clock value (for more on self-stabilization, see [2]).

The strength of self-stabilizing systems emerges from their ability to continue functioning after recovering from a massive disruption of their assumed working conditions. The advantage of *Byzantine* tolerant systems comes from being able to withstand any kind of faults while the system operates in its known boundaries. By combining these two fault models, a distributed system can continue operating properly in the presence of faults as long as "everything is going well"; however, if "things aren't going well", the system will be able to recover once the conditions hold again, and the ratio of *Byzantine* nodes hold.

Clock synchronization is a fundamental building block in many distributed systems; hence, creating a self-stabilizing *Byzantine* tolerant clock synchronization is a desirable goal. Once such an algorithm exists, one can stabilize *Byzantine* tolerant algorithms that were not designed for self-stabilization (see [3]). Clock synchronization can be created upon a PULSEing algorithm (see [4]), which is the main motivation behind the current paper.

The main contribution of the current paper is to develop a pulse synchronization algorithm that converges once the communication network resumes delivering messages within bounded, say d, time units, and the number of Byzantine nodes, f, obeys the $n > 3f$ inequality, for a network of n nodes. The attained pulse synchronization tightness is $3d$ with a deterministic convergence time of a constant number of pulse cycles (each containing $O(f)$ communication rounds).

1.1 Related Work

Algorithms combining self-stabilization and *Byzantine* faults, can be divided into two classes. The first consists of problems in which the state of each node is determined locally (see [5,6,7]). The other class contains problems such that a node's state requires global knowledge - for example, clock synchronization such that every two nodes' clocks have a bounded difference that is independent of the diameter of the network (see [8,9,4]). The current paper is of the latter class.

The current paper makes use of the self-stabilizing *Byzantine* agreement algorithm (ss-Byz-Agree) presented in [10]. The above work operates in exactly the same model as the current paper, and its construction will be used as the basic building block in our current solution. Appendix A lists the main properties of this building block.

When discussing clock synchronization, it is common to represent the clocks as an integer value that progresses linearly in time (see [11]). This was previously termed digital clock synchronization ([12,13,14,15]) or "synchronization of phase-clocks" ([16]). In the current paper we provide a PULSEing algorithm; however, when comparing it to other results, we consider the digital clock synchronization algorithm that can be built upon it (as in [4]).

The first ever algorithm to address self-stabilizing *Byzantine* tolerant clock synchronization is presented in [8]. [8] discusses two models; one is synchronous, that is, all nodes are connected to some global "tick" system that produces "ticks" that reach all nodes at the same time, and messages sent at any given tick reach their destination before the following tick. The second model is a bounded-delay network, in which there is no common tick system, but messages have a bounded delay on their delivery time. There is no reason to consider an asynchronous model, since even a single fail-stop failure can't be overcome (see [17]). Note that the bounded-delay model contains the first (synchronous) one. [8] gives two solutions, one for each model, both of which converge in expected exponential time; both algorithms support $f < \frac{n}{3}$.

In [9] clock synchronization is reached in deterministic linear time. However, [9] addresses only the synchronous model, and supports only up to $f < \frac{n}{4}$. In [18], a PULSEing algorithm that operates in the synchronous model is presented, which converges in deterministic linear time, and supports $f < \frac{n}{3}$, matching the lower bounds both in the maximal number of *Byzantine* nodes, and in the convergence time (see [19] for lower bounds). In [20] a very complicated pulse synchronization protocol, in the same model as the current paper, was presented.

The current paper presents a PULSE-synchronization algorithm, which has deterministic linear convergence time, supports $f < \frac{n}{3}$, and operates in a bounded-delay model.

2 Model and Problem Definition

The model used in this paper consists of n nodes that can communicate via message passing. Each message has a bounded delivery time, and a bounded processing time at each node; in addition the message sender's identity can be validated. The network is not required to support broadcast.

Each node has a local clock. Local clocks might show different readings at different nodes, but all clocks advance at approximately the real-time rate.

Nodes may be subject to transient faults, and at any time a constant fraction of nodes may be *Byzantine*, where f, the number of *Byzantine* nodes satisfies $f < \frac{n}{3}$.

Definition 1. *A node is* **non-faulty** *if it follows its algorithm, processes messages in no more than π time units and has a bounded drift on its internal clock. A node that is not* **non-faulty** *is considered* **faulty** *(or Byzantine). A node is* **correct** *if it has been* **non-faulty** *for Δ_{node} time units.*[1]

Definition 2. *A communication network is* **non-faulty** *if messages arrive at their destinations within δ time units, and the content of the message, as well as the identity of the sender, arrive intact. A communication network is* **correct** *if it has been* **non-faulty** *for Δ_{net} time units.*[2]

[1] The value of Δ_{node} will be stated later.
[2] The value of Δ_{net} is stated below.

The value of Δ_{net} is chosen so if at time t_1 the communication network is non-faulty and stays so until $t_1 + \Delta_{net}$, then only messages sent *after* t_1 are received by non-faulty nodes.

Definition 3. *A system is* **coherent** *if the network is* **correct** *and there are at least* $n - f$ **correct nodes**.

Once the system is coherent, a message between two correct nodes is sent, received and processed within d time units, where d is the sum of δ, π and the upper bound on the potential drift of correct local timers during such a period. Δ_{net} should be chosen in such a way as to satisfy $\Delta_{net} \geq d$.. Since d includes the drift factor, and since all the intervals of time will be represented as a function of d, we will not explicitly refer to the drift factors in the rest of the paper.

2.1 Self-stabilizing Byzantine Pulse-Synchronization

Intuitively, the PULSE synchronization problem consists of synchronizing the correct nodes so they invoke their pulses together *Cycle* time apart. That is, all correct nodes should invoke pulses within a short interval, then not invoke a pulse for approximately *Cycle* time, then invoke pulses again within a short interval, and so on. Adding "Self-stabilizing *Byzantine*" to the PULSE synchronization problem, means that starting from any memory state and in spite of ongoing *Byzantine* faults, the correct nodes should eventually invoke pulses together *Cycle* time apart.

Since message transmission time varies and also due to the *Byzantine* presence, one cannot require the correct nodes to invoke pulses exactly *Cycle* time apart. Instead, $cycle_{min}$ and $cycle_{max}$ are values that define the bounds on the actual CYCLE length in a correct behavior. The protocol presented in this paper achieves $cycle_{min} = Cycle \leq \text{CYCLE} \leq Cycle + 12d = cycle_{max}$.

To formally define the PULSE synchronization problem, a notion of "PULSEing together" needs to be addressed.

Definition 4. *A correct node p invokes a pulse* **near** *time unit t if it invokes a pulse in the time interval* $[t - \frac{3}{2} \cdot d, t + \frac{3}{2} \cdot d]$. *Time unit t is a* **pulsing point** *if every correct node invokes a pulse near t.*

Definition 5. *A system is in a* **synchronized_pulsing_state** *in the time interval* $[r_1, r_2]$ *if*

1. *there is some pulsing point* $t_0 \in [r_1, r_1 + cycle_{max}]$ *;*
2. *for every pulsing point* $t_i \leq r_2 - cycle_{\max}$ *there is another pulsing point* t_{i+1}, $t_{i+1} \in [t_i + cycle_{min}, t_i + cycle_{\max}]$*;*
3. *for any other pulsing point* $\bar{t} \in [r_1, r_2]$, *there exists i, such that* $|t_i - \bar{t}| \leq \frac{3}{2} \cdot d$.

Intuitively, the above definition says that in the interval $[r_1, r_2]$ there are pulsing points that are spaced at least $cycle_{min}$ apart and no more than $cycle_{max}$ apart.

Definition 6. *Given a coherent system,* The Self-Stabilizing Pulse Synchronization Problem *requires that:*

Convergence: *Starting from an arbitrary system state, the system reaches a synchronized_pulsing_state within a finite amount of time.*
Closure: *If the system is in a synchronized_pulsing_state in some interval* $[t_1, t_2]$ *(s.t.* $t_2 > t_1 + cycle_{\max}$*), then it is also in a synchronized_pulsing_state in the interval* $[t_1, t]$ *for any* $t > t_2$*.*

3 Solution Overview

The main algorithm, Erratic-Pulser, assumes a self-stabilizing, *Byzantine* tolerant, distributed agreement primitive, $\mathcal{Q}$, which is defined in the following section. A protocol providing the requirements of $\mathcal{Q}$ is presented in Section 5.

Using $\mathcal{Q}$, the Erratic-Pulser algorithm produces agreement among the correct nodes on different points in time at which they invoke pulses; and these points become sparse enough. Using this basic point-in-time agreement, a full pulse algorithm is built, named Balanced-Pulser. By using the basic pulseing pattern produced by Erratic-Pulser, Balanced-Pulser manages to solve the pulse-synchronization problem.

During the rest of this paper, the constants *Cycle*, $cycle_{min}$ and $cycle_{max}$ are used freely. However, it is important to note that *Cycle* must be chosen such that it is large enough. The exact limitations on the possible values of *Cycle* will be stated later. An explanation on how to create a pulseing algorithm with an arbitrary *Cycle* value is presented in Section 9.

4 The $\mathcal{Q}$ Primitive

$\mathcal{Q}$ is a primitive executed by all the nodes in the system. However, each invocation of a specific $\mathcal{Q}$ is associated with some node, p, hence a specific invocation will sometimes be referred to as $\mathcal{Q}(p)$. That is, $\mathcal{Q}(p)$ is a distributed algorithm, executed by all nodes, and triggered by p (p's special role in the execution of $\mathcal{Q}(p)$ will be elaborated upon later). In the following discussion several instances of $\mathcal{Q}(p)$ may coexist, but it will be clear from the context to which instance $\mathcal{Q}(p)$ refers. Each instance is a separate copy of the protocol and each node executes each instance separately.

$\mathcal{Q}(p)$ is a "consensus primitive", that is, each node q has an input value v_q, and upon completing the execution of $\mathcal{Q}(p)$ it produces some output value V_q. The input values and output values are boolean, i.e., $v_q, V_q \in \{0, 1\}$. Denote by τ_q the local-time at node q at which V_q is defined; that is τ_q is the local time at node q at which q determines the value of V_q (and terminates $\mathcal{Q}(p)$).

The un-synchronized and distributed nature of $\mathcal{Q}(p)$ requires distinguishing between two stages. The first stage is when p attempts to invoke $\mathcal{Q}(p)$; this attempted invocation involves exchanging messages among the nodes. The second stage is when enough correct nodes agree to join p's invocation of $\mathcal{Q}(p)$, and

hence start executing $\mathcal{Q}(p)$. When p is correct, the first stage and the second stage are close to each other; however, when p is faulty, no a priori bound can be set on the time difference between the first and the second stages. Note that p itself joins the instance of $\mathcal{Q}(p)$ only after the preliminary invocation stage.

Informally, $join_q$ is the time at which q agrees to join the instance of $\mathcal{Q}(p)$ (which is also the time at which q determines its input value v_q.) Following this stage it actively participates in determining the output value of $\mathcal{Q}(p)$. The implementation of $\mathcal{Q}(p)$ needs to explicitly instruct a node when to determine its input value.

In the following discussion, rt_{invoke} will denote the time at which p invoked $\mathcal{Q}(p)$ and $join_{first}$ will denote the time value at which the first correct node joins the execution of $\mathcal{Q}(p)$; $join_{last}$ will denote the time value at which the last correct node joins p in executing $\mathcal{Q}(p)$. That is, $join_{first} = \min_{\text{correct } q}\{join_q\}$ and $join_{last} = \max_{\text{correct } q}\{join_q\}$.

$\mathcal{Q}(p)$ is self-stabilizing, and its properties hold once the system executing it is coherent for at least $\Delta_{\mathcal{Q}}$ time. In other words, no matter what the initial values in the nodes' memory may be, after the system has been coherent for $\Delta_{\mathcal{Q}}$ time, the properties of $\mathcal{Q}(p)$ will hold.[3]

$\mathcal{Q}(p)$'s properties follow. Observe that there are different requirements, depending on whether p is a correct node or not.

1. For any node p invoking $\mathcal{Q}(p)$, the following holds:
   (a) *Agreement:* all correct nodes that have terminated have the same output value. That is, for any pair of correct nodes, q and q', which have completed $\mathcal{Q}(p)$, $V_q = V_{q'}$. V denotes this common output value.
   (b) *Validity:* if all correct nodes have the same input value ν then $V = \nu$.
   (c) *Termination:* if some correct node joins $\mathcal{Q}(p)$ then all correct nodes terminate within Δ_{max} time units from $join_{first}$ but no quicker than Δ_{min}. That is, for a correct q, $rt(\tau_q) \in [join_{first} + \Delta_{min}, join_{first} + \Delta_{max}]$, where τ_q is the local time at which q determines the value of V_q, and $rt(\tau_q)$ is the time at which this takes place.
   (d) *Tightness:* if a correct node terminates, then for any correct nodes q, q': $|rt(\tau_q) - rt(\tau_{q'})| \leq 3 \cdot d$.
   (e) *Collaboration:* if one correct node joins the execution of $\mathcal{Q}(p)$, then all correct nodes join the execution of $\mathcal{Q}(p)$ within $3 \cdot d$ of each other; that is, $|join_{last} - join_{first}| \leq 3 \cdot d$.
2. For a correct node p, starting the execution of $\mathcal{Q}(p)$ at time rt_{invoke}, the following holds:
   (a) *Strong Termination:* $join_{first} \leq rt_{invoke} + 3 \cdot d$. That is, the first correct node to join p in executing $\mathcal{Q}(p)$ does so within $3 \cdot d$ time from p's invocation of $\mathcal{Q}(p)$. Combined with *termination*, this property means that all correct nodes terminate by $rt_{invoke} + 3 \cdot d + \Delta_{max}$.
   (b) *Separation:* p does not start $\mathcal{Q}(p)$ more than once every $3 \cdot \Delta_{max}$ time units.

[3] $\Delta_{\mathcal{Q}}$ is defined below.

3. The following holds for a faulty p, invoking $\mathcal{Q}(p)$:
   (a) *Separation:* if a correct node q assigns an output value for $\mathcal{Q}(p)$ at some time t_1, then it does not assign an output value for $\mathcal{Q}(p)$ again before $t_1 + 2 \cdot \Delta_{min}$.

Remark 1. According to "termination" if $join_{first}$ is not defined, all correct nodes do not terminate. This implies that all correct nodes terminate if and only if some correct node joins p in executing $\mathcal{Q}(p)$.

Note that p may require the invocation of several $\mathcal{Q}(p)$ instances concurrently. To differentiate between these instances, they are marked with an additional index, e.g $\mathcal{Q}_1(p), \mathcal{Q}_2(p)$, etc. Each such instance has its own memory space, and hence is independent of other instances. According to the *separation* property, a correct node does not execute the same instance of $\mathcal{Q}(p)$ too often. That is, $\mathcal{Q}_1(p)$ is not executed until the previous $\mathcal{Q}_1(p)$ has terminated. A faulty node p may try to invoke $\mathcal{Q}_1(p)$ as often as it likes, however correct nodes will ignore the multiple executions.

5 Implementing $\mathcal{Q}(p)$, the ss-Byz-Q Algorithm

The implementation of $\mathcal{Q}(p)$ makes use of ss-Byz-Agree ([10]). The properties of ss-Byz-Agree and its guarantees are listed in Appendix A. In ss-Byz-Agree, when a node p wants to start an agreement on some value, it sends (*Initiator*, p, v_p) to all other nodes. Nodes receiving this message, initiate the ss-Byz-Agree algorithm, and start participating in the agreement. Other nodes, that have not received the (*Initiator*, p, v_p) message (in case p is *Byzantine*), join the ss-Byz-Agree algorithm once they are "convinced" that enough correct nodes are already executing ss-Byz-Agree on p's value.

This leads to the following insight. If a correct node q ignores an (*Initiator*, p, v_p) message sent by a *Byzantine* node (for any reason), it does not change the properties of ss-Byz-Agree. Since due to p's *Byzantine* nature, if p would have not sent this specific message to q, ss-Byz-Agree's properties would still hold. Hence, whether p sends the message and a correct node ignores it, or p doesn't send the message at all, the properties of ss-Byz-Agree remain the same. Note that this is true only if p is *Byzantine*. In what follows, when a node *rejects* a message it ignores it, and when it *accepts* a message it continues to execute the protocol as instructed.

Figure 1 presents an algorithm that implements the $\mathcal{Q}$ primitive. If node p wants to invoke $\mathcal{Q}(p)$, it does so by executing ss-Byz-Agree (p, *start_Q*) (this is the *Init* stage), which means it sends (*Initiator*, p, *start_Q*) messages to other nodes. This action triggers the *prolog* stage of the protocol. If this stage completes successfully, each correct node peforms a timing test to determine whether to join the computation of the primitive $\mathcal{Q}(p)$. The algorithm is executed in the background continuously, and it responds to messages / events that are triggered by p's execution of ss-Byz-Agree (p, *start_Q*).

Algorithm ss-Byz-Q
(implementing $\mathcal{Q}(p)$) /* executed at node q */

Init: **If** $p = q$ invoke ss-Byz-Agree $(p, start\_Q)$);
/* by sending (Initiator, p, $start\_Q$)) message to all */

Prolog: **On** receiving (*Initiator*, p, $start\_Q$) message from p
if $local_q > last_q[p] + 2 \cdot \Delta_{max}$ **then** accept the message;
else ignore the message;

The Primitive $\mathcal{Q}(p)$:

1. **On** returning from ss-Byz-Agree for p with value "$start\_Q$" **do**
   if $local_q > last_q[p] + \Delta_{max} + 3 \cdot d$ **then**
   begin
   determine the input value v_q; /* this is when q joins $\mathcal{Q}(p)$ */
   $start_q[p] := local_q$;
   reset $val_q[p, \_]$;
   wait for $3 \cdot d$ and then invoke ss-Byz-Agree $(q, (p, v_q))$;
   /* by sending (Initiator, q, (p, v_q)) message to all */
   end
   $last_q[p] := local_q$;
2. **On** receiving (*Initiator*, p', $(p, v_{p'})$)
   if $local_q \leq start_q[p] + 7 \cdot d$ **then** accept the message;
   else ignore the message;
3. **On** returning from ss-Byz-Agree for p' with value $(p, v_{p'})$ **do**
   $val_q[p, p'] = v_{p'}$;
4. **At** time $local_q = start_q[p] + \Delta + 17 \cdot d$
   for all p', check the agreement values $val_q[p, p']$
   if there are $n - f$ 1's, then set $V_q[p] := 1$, otherwise set $V_q[p] := 0$;
   return $V_q[p]$ as $\mathcal{Q}(p)$'s output value;

Cleanup:
for any p: if $last_q[p] > local_q$ then $last_q[p] := local_q$
for any p: if $start_q[p] > local_q$ then $start_q[p] := local_q$

Fig. 1. An algorithm that implements $\mathcal{Q}(p)$

The values of the constants for the ss-Byz-Q algorithm are: $\Delta_{max} := \Delta + 20 \cdot d$ and $\Delta_{min} := \Delta_{max} - 3 \cdot d$, where Δ represents the maximal time required to complete ss-Byz-Agree ($\Delta := 7(2f + 3)d$, see Appendix A).

In the $\mathcal{Q}(p)$ protocol in Figure 1, $local_q$ represents the local time at each node q; in addition there are two arrays of values: $start_q, last_q$. These arrays hold local-time values (per node p) of events regarding $\mathcal{Q}(p)$'s execution at q. $last_q[p]$ is used to ensure that q doesn't participate in $\mathcal{Q}(p)$ too often. $start_q[p]$ is used so that all correct nodes know when to stop collecting values of other nodes (regarding $\mathcal{Q}(p)$'s instance); these values are stored in $val_q[p, p']$.

Remark 2. In the protocols, all the comparisons of the value of $local_q$ to some other value, always compare values that are at most some bounded range apart, say D. To deal with the possible wraparound of the counter $local_q$, it is enough

that the range of values of $local_q$ will be $D' > 2D$. The "cleanup" stage of the protocol (See Figure 1) ensures that comparisons over a circle of size D' are uniquely determined.

Note that the protocol parameters n, f and *Cycle* (as well as the system characteristic d) are fixed constants and thus considered part of the incorruptible correct code.[4] Thus we assume that non-faulty nodes do not hold arbitrary values of these constants.

The value of Δ_{node} is crucial for the following claims. Δ_{node} is used to ensure that non-faulty nodes "run" for some time before they become correct. In the context of this paper, a non-faulty node should not be considered correct when it executes SS-BYZ-Q that it might have joined before it was non-faulty. Moreover, since SS-BYZ-Q uses $ss-$ BYZ-AGREE which has its own requirements for a node's correctness, we set $\Delta_{node} := \Delta_{\text{node-ss-byz-agree}} + \Delta_{max} + 3 \cdot d$.[5]

Lemma 1. *Once the system is coherent, if all correct nodes pass the condition in Line 1 during a time interval* $[t_1, t_2]$ *s.t.*
1. $t_2 - t_1 \leq 3 \cdot d$*, and*
2. at t_1 *for any correct node* q *it holds that* $local_q > start_q[p] + \Delta_{max}$*,*
then Agreement, Validity, Termination, Tightness *and* Collaboration *hold.*

Proof. First we show that no $ss-$ BYZ-AGREE$(p', (p, v_{p'}))$ that was initiated before t_1, terminates after t_1. By assumption, at time t_1, each correct node q has $local_q > start_q[p] + \Delta_{max}$, which means that no correct node has accepted (*Initiator*, p', $(p, v_{p'})$) in the time interval $[t_1 - \Delta_{max} + 7 \cdot d, t_1]$. In the protocol, any correct node that accepts (*Initiator*, p', $(p, v_{p'})$) before $t_1 - \Delta_{max} + 7 \cdot d$, must have terminated the ss-BYZ-AGREE no later than $t_1 - \Delta_{max} + 7 \cdot d + \Delta + 7 \cdot d$, and hence all correct nodes must have terminated the ss-BYZ-AGREE no later than $t_1 - \Delta_{max} + 17 \cdot d + \Delta$. Since $\Delta_{max} := \Delta + 20 \cdot d$, we conclude that any ss-BYZ-AGREE that was invoked before t_1 terminated before t_1.

In addition, due to setting of $last_q[p]$ in Line 1, no correct node will pass the condition in Line 1 again, before $t_1 + \Delta_{max} + 3 \cdot d$. Hence, during the interval $[t_2, t_2 + \Delta_{max}]$ no correct node passes the condition in Line 1. Note that each correct node passes the condition in Line 1 exactly once in the interval $[t_1, t_2]$. Hence, all correct nodes reset $val_q[p, \_]$ in the interval $[t_1, t_2]$ and never do so again before $t_2 + \Delta_{max}$. In a sense, the above means that all correct nodes join p in the interval $[t_1, t_2]$ and do not join p again, until after time $t_2 + \Delta_{max}$.

From the lemma's condition; for any two correct nodes q, q', it holds that $|rt(start_q[p]) - rt(start_{q'}[p])| \leq 3 \cdot d$. At this stage, each correct node joins p's execution of $\mathcal{Q}(p)$ and hence *Collaboration* holds.

For any pair of correct nodes, q, q', $|rt(start_q[p]) - rt(start_{q'}[p])| \leq 3 \cdot d$. Moreover, q sends its (*Initiator*, q, (p, v_q)) message $3 \cdot d$ after its $start_q[p]$. Since $|rt(start_q[p]) - rt(start_{q'}[p])| \leq 3 \cdot d$, q' has already set its $start_{q'}[p]$ value when it receives q's (*Initiator*, q, (p, v_q)) message. Similarly, q' receives q's (*Initiator*, q, (p, v_q)) message within $7 \cdot d$ of $start_{q'}[p]$ ($3d$ is the waiting of $3d$ in Line 1 of the

[4] A system cannot self-stabilize if the entire code space can be perturbed, see [21].
[5] $\Delta_{\text{node-ss-byz-agree}} := 14(2 \cdot f + 3) \cdot d + 10 \cdot d$ (see [10]).

protocol, additional $3d$ is the time difference in $start_q$, and d is the uncertainty in message delivery), and thus does not ignore it.

This last argument implies that for every correct node q, any other correct node q' accepts its (*Initiator*, q, (p, v_q)) message, and hence finishes ss-Byz-Agree (q, v_q) before time $start_{q'} + \Delta + 7 \cdot d$. Therefore, every correct node "hears" every other correct node's value. That is, for any triplet of correct nodes, q, q', q'' it holds that $val_q[p, q''] = val_{q'}[p, q'']$.

Consider a *Byzantine* node q. If some correct node q' has accepted its (*Initiator*, q, (p, v_q)) message, then according to point 3 of the "*Timeliness-Agreement*" property (see Appendix A), ss-Byz-Agree will terminate within $\Delta + 7 \cdot d$ time units. Hence, any other correct node q'' will terminate within $3 \cdot d$. Since $|start_{q'} - start_{q''}| \leq 3 \cdot d$, node q'' will have accepted the same value no later than $start_{q''} + \Delta + 16 \cdot d$ ($7d$ come from above, $3d$ come from the difference in $start_q$, $3d$ come from the difference in the termination of ss-Byz-Agree and another $3d$ from the waiting after setting $start_q[p]$; all together $7d + 3d + 3d + 3d = 16d$). Note that this proof holds even though correct nodes may ignore an (*Initiator*, q, (p, v_q)) message sent by a *Byzantine* node q. Since no ss-Byz-Agree that was invoked before t_1 is accepted after t_1, it holds that $val_{q'}[p, q] = val_{q''}[p, q]$. As a result all correct nodes have the same set of values when they consider the output value $v_q[p]$, hence they all agree on the same output value. In addition, if all correct nodes started with "0", they will see at most f "1"s, and hence decide $V = 0$. Moreover, if all correct nodes started with "1", then all correct nodes will decide 1. Thus *Agreement* and *Validity* hold.

Each correct node terminates within $\Delta + 17 \cdot d$ of returning from p's invocation of ss-Byz-Agree (which is the joining point of each correct node to $\mathcal{Q}(p)$), and they all terminate within $3 \cdot d$ time units of each other (since $|rt(start_q) - rt(start_{q'})| \leq 3 \cdot d$). Hence *Termination* and *Tightness* hold. □

The following shows that ss-Byz-Q converges in $\Delta_{\mathcal{Q}} := 4 \cdot \Delta_{max} + \Delta_{ss-\text{Byz-Agree}}$. For ss-Byz-Q to operate correctly, ss-Byz-Agree must converge as well. Hence, in the following, we will assume that $\Delta_{ss-\text{Byz-Agree}}$ time has already passed. [6]

Lemma 2. *Once the system has been coherent for $4 \cdot \Delta_{max}$ time units, then for a faulty p, the properties of $\mathcal{Q}(p)$ hold for* ss-Byz-Q.

Proof. Note that once the system is coherent, $start_q[p], last_q[p] \leq local_q$. Notice that $start_q[p]$ can only be updated at Line 1.

Consider the first $2 \cdot \Delta_{max}$ time units following the time at which the system became coherent. If some correct node terminates ss-Byz-Agree $(p, start\_\mathcal{Q})$ during this period, then all correct nodes do so within $3 \cdot d$ of each other. Hence, they all set their $last_q[p]$ variable within $3 \cdot d$ time units of each other. That is, the values $rt(last_q[p])$ are at most $3 \cdot d$ units apart from each other. If no correct node terminates ss-Byz-Agree $(p, start\_\mathcal{Q})$ for $2 \cdot \Delta_{max}$, then all $last_q[p]$ haven't been updated for $2 \cdot \Delta_{max}$ and hence all $last_q[p] + 2 \cdot \Delta_{max} < local_q$ for every correct node q.

[6] $\Delta_{ss-\text{Byz-Agree}} := 2\Delta + 10d$ (see [10]).

Thus, we conclude that after $2 \cdot \Delta_{max}$ time units either all correct nodes have $rt(last_q[p])$ within $3 \cdot d$ of each other or all correct nodes have $last_q[p] + 2 \cdot \Delta_{max} < local_q$. Note that this state continues to hold as long as no correct node enters Line 1 since $last_q[p]$ is not updated at any correct node. If some correct node does update $last_q[p]$ at Line 1, then all correct nodes do so $3 \cdot d$ time units apart.

Now consider the period between $2 \cdot \Delta_{max}$ and $4 \cdot \Delta_{max}$ time units following the time the system became coherent. If no correct node terminated ss-Byz-Agree (p, $start\_\mathcal{Q}$), then each correct node, q, has $local_q > start_q[p] + \Delta_{max}$ (since $start_q[p]$ had not been updated for at least $2 \cdot \Delta_{max}$ time units). Otherwise, if some correct node q' has terminated ss-Byz-Agree (p, $start\_\mathcal{Q}$), it means that there exists some correct node $\bar{q}$ that accepted (*Initiator*, p, $start\_\mathcal{Q}$) at the *Prolog* stage. Thus, $local_{\bar{q}} > last_{\bar{q}}[p] + 2 \cdot \Delta_{max}$, which means that until $last_{\bar{q}}[p]$ is reset, $local_{\bar{q}} > last_{\bar{q}}[p] + \Delta_{max} + 3 \cdot d$. Remember that either the $rt(last_q[p])$ of each correct node $\bar{q}$ is within $3 \cdot d$ time units of all other correct nodes, or each correct node has $last_q[p] + 2 \cdot \Delta_{max} < local_q$. Therefore, we have that for all correct nodes, until $last_q[p]$ is reset, $local_q > last_q[p] + \Delta_{max} + 3 \cdot d$; which means that when q terminates ss-Byz-Agree (p, $start\_\mathcal{Q}$) it passes the condition of Line 1, along with all other correct nodes (within a $3 \cdot d$ interval). Therefore, the $rt(start_q[p])$ of all correct nodes are within $3 \cdot d$ time units.

Thus, after $4 \cdot \Delta_{max}$, the $rt(start_q[p])$ of all correct nodes are within $3 \cdot d$ time units *and* $rt(last_q[p])$ within $3 \cdot d$ time units of each other or all correct nodes have $last_q[p] + 2 \cdot \Delta_{max} < local_q$. Note that the next time p invokes ss-Byz-Agree, all correct nodes values of $start_q[p]$ will be greater than their $local_q$ by at least $2 \cdot \Delta_{max}$. Hence, if p invokes ss-Byz-Agree and some correct node terminates that instance of ss-Byz-Agree, then all correct nodes pass the condition of Line 1 within $3 \cdot d$ of each other, and each correct node has $local_q > start_q[p] + \Delta_{max}$. Hence, by Lemma 1 all properties except for *Separation* hold.

To show that *Separation* holds, notice that once a correct node has passed Line 1, it won't do so again for at least $2 \cdot \Delta_{max} - 3 \cdot d$ time units. In addition, it will terminate the current instance of $\mathcal{Q}$ within Δ_{max}. Hence, the next invocation of $\mathcal{Q}$ cannot terminate before $2 \cdot \Delta_{min}$. And *Separation* holds. □

Lemma 3. *Once the system has been coherent for $4 \cdot \Delta_{max}$ time units, then for a correct p, the properties of $\mathcal{Q}(p)$ hold for* ss-Byz-Q*, given that p does not initiate ss-*Byz-Agree *(p, start_Q) earlier than $3 \cdot \Delta_{max}$ time units following its previous invocation.*

Proof. Since *Agreement, Validity, Termination, Tightness* and *Collaboration* were proven to hold even if p is faulty (under the lemma's conditions), they clearly hold if p is correct. Hence, we still need to prove *Strong Termination* and *Separation*. To prove *Strong Termination*, note that if p is correct, and it has not invoked $\mathcal{Q}(p)$ for $3 \cdot \Delta_{max}$ time units, then when it does invoke $\mathcal{Q}(p)$, all correct nodes will accept the message (*Initiator*, p, $start\_\mathcal{Q}$) and hence, according to item 2 of the "*Timeliness-Agreement*" property of ss-Byz-Agree (see Appendix A), all correct nodes will terminate within $3 \cdot d$ time units following p's invocation of ss-Byz-Agree ($p, start\_\mathcal{Q}$) and join the execution of $\mathcal{Q}(p)$. *Separation* follows from the conditions of the lemma. □

From the above lemmas, we conclude that after $4 \cdot \Delta_{max} + \Delta_{ss-\text{Byz-Agree}}$ time units, ss-Byz-Q behaves according to $\mathcal{Q}$'s properties. Setting $\Delta_{\mathcal{Q}} := 4 \cdot \Delta_{max} + \Delta_{ss-\text{Byz-Agree}}$ satisfies the claim that if the system has been coherent for $\Delta_{\mathcal{Q}}$ time units, then the properties of $\mathcal{Q}$ hold.

Since ss-Byz-Q implements $\mathcal{Q}$'s properties correctly, in the rest of the paper we will use ss-Byz-Q and $\mathcal{Q}$ interchangeably.

6 Constructing the Erratic-Pulser Algorithm

The Erratic-Pulser algorithm (Figure 2) is written in an event-driven fashion; that is, it is continuously executed in the background and no explicit initialization is needed. The algorithm requires invoking two $\mathcal{Q}$ instances per node ($\mathcal{Q}_{start}$ and $\mathcal{Q}_{end}$). In addition, each node has three timers TIMER_{start}, TIMER_{end} and TIMER_{main} with elapsed time of CYCLE_{start}, CYCLE_{end} and CYCLE_{main}, respectively. When TIMER_{start} or TIMER_{end} elapse, an instance of ss-Byz-Q is invoked ($\mathcal{Q}_{start}$ for TIMER_{start} and $\mathcal{Q}_{end}$ for TIMER_{end}). TIMER_{main} is used to determine the value of $WantToPulse$, which is used as the input value for $\mathcal{Q}_{start}$ and $\mathcal{Q}_{end}$. If TIMER_{main} is elapsed, then $WantToPulse := 1$, and once TIMER_{main} is reset, $WantToPulse := 0$ until it elapses again.

The intuition behind the algorithm is that $WantToPulse$ determines when p is willing to invoke a pulse. Once all correct nodes have $WantToPulse = 1$, the next time a ss-Byz-Q instance is invoked, all of them will invoke pulses.

Remark: Notice that there is a difference between TIMER_{start}, TIMER_{end} and TIMER_{main}. TIMER_{start}, TIMER_{end} are timers that when they elapse, an event occurs, and the algorithm performs some action. These timers are always set, that is, once they elapse, they are reset immediately. TIMER_{main}, on the other

Algorithm **Erratic-Pulser** /* executed at node p */
/* the $\mathcal{Q}$s are executed in the background */
/* the input value v_q for each $\mathcal{Q}$ instance, is the value of $WantToPulse$ at the time q joins $\mathcal{Q}$ */

1. **when** TIMER_{start} elapses
   reset TIMER_{start} with CYCLE_{large};
   reset TIMER_{end};
   invoke $\mathcal{Q}_{start}(p)$;
2. **when** TIMER_{end} elapses
   reset TIMER_{end} with CYCLE_{large};
   invoke $\mathcal{Q}_{end}(p)$;
3. $WantToPulse := 1$ if TIMER_{main} has elapsed, and $WantToPulse := 0$, otherwise;
4. **on** returning from either $\mathcal{Q}_{start}(q)$ or $Q_{end}(q)$ for some q with value $V = 1$
   (a) invoke a pulse;
   (b) reset TIMER_{main};
   (c) reset TIMER_{start};

cleanup:
if a TIMER is set with invalid value (below 0 or above its maximal value), reset it; for TIMER_{main}, 0 is a valid value;

Fig. 2. An algorithm achieving basic synchronized PULSEing

hand, will remain in its elapsed state until it is reset. That is, Line 3 is not executed only when TIMER_{main} elapses, but rather it is executed continuously. In a sense, when q wants to read its $WantToPulse$ variable value, it checks whether TIMER_{main} has elapsed; if so then it considers $WantToPulse = 1$, otherwise it reads $WantToPulse = 0$.

The following are the values of the constants used in ERRATIC-PULSER.

$$\text{CYCLE}_{start} := \text{CYCLE}_{main} := Cycle - \Delta_{max} - \Delta_{min};$$
$$\text{CYCLE}_{end} = \Delta_{min} - 10 \cdot d;$$
$$\text{CYCLE}_{large} := 2 \cdot (\Delta_{max} + \text{CYCLE}_{start} + \text{CYCLE}_{end}).$$

Note: CYCLE_{main} needs to be larger than $\Delta_{max} + 9 \cdot d$ time units, hence, $Cycle$ must be larger than $2 \cdot \Delta_{max} + \Delta_{min} + 9 \cdot d$ time units.

7 Erratic-Pulser's Correctness Proofs

Definition 7. *A correct node p* ***pulses-in-unison****, there is a pulsing point t, such that p invokes a pulse near t each time that p invokes a pulse. The system* ***pulses-in-unison****, if for every correct node p , p* ***pulses-in-unison***

Remark 3. The definition of "near t" implies that if p **pulses-in-unison** then each time p invokes a pulse there is a time interval $[t_1, t_2]$ such that $|t_2 - t_1| \leq 3 \cdot d$ and each correct node (including p) invokes a pulse within this interval. This also implies that if there exists a correct node p that **pulses-in-unison** then the system **pulses-in-unison**.

Lemma 4. *Once the system has been coherent for $\Delta_{\mathcal{Q}}$ time, the system pulses-in-unison.*

Proof. According to Lemma 2 and Lemma 3 (in Section 5), once the system has been coherent for $\Delta_{\mathcal{Q}}$time units, all copies of SS-BYZ-Q behave according to the requirements of $\mathcal{Q}$. This means that all correct nodes see the same output values. Since a correct node invokes a pulse only in accordance with the output of a $\mathcal{Q}$, if some correct node invokes a pulse, then within $3 \cdot d$ time units from its pulse, all correct nodes will also invoke pulses. This means that every correct node pulses-in-unison, which means that the system pulses-in-unison. □

The following lemma proves that a correct node will eventually invoke a pulse. The previous lemma claims that after some time, **if** a correct node invokes a pulse, then all the correct nodes invoke pulses.

Lemma 5. *Eventually some correct node will invoke a pulse. This happens no later than $\Delta_{\mathcal{Q}} + \Delta_{max} + \text{CYCLE}_{large} + \text{CYCLE}_{main} + 3 \cdot d$ time units after the point at which the system becomes coherent.*

Proof. Consider the system $\Delta_{\mathcal{Q}}$ after it becomes coherent: If a correct node invokes a pulse, the lemma holds. Otherwise, after CYCLE_{main} time units, all correct nodes will have $WantToPulse$ as 1. Eventually, after no more than CYCLE_{large},

TIMER_{start} at some correct p will expire, which will initiate $\mathcal{Q}_{start}(p)$, that terminates no more than $\Delta_{max} + 3 \cdot d$ time units afterwards (by *strong termination*), and will have the output value $V = 1$ (since all correct nodes had the input value of $v = 1$). By line 4, of the ERRATIC-PULSER, p will invoke a pulse. □

Lemma 6. *Once the system pulses-in-unison, let t_1 be a time unit at which a correct node p invokes a pulse. Let t_2 be the last time at which p invokes a pulse in the interval $[t_1, t_1 + \Delta_{max} + 3 \cdot d]$. p does not invoke a pulse in the interval $[t_2, t_2 + \text{CYCLE}_{main} - 3 \cdot d + \Delta_{min}]$. p invokes a pulse at some time t_3, where $t_3 \leq t_2 + \text{CYCLE}_{main} + 6 \cdot d + \Delta_{max}$.*

Proof. According to the lemma's assumption the system pulses-in-unison. Hence, when p invokes a pulse at t_1 all correct nodes invoke pulses before time $t_1 + 3 \cdot d$. Define $[t_s, t_e]$ to be the time interval in which all correct nodes have invoked a pulse, such that $t_1 \in [t_s, t_e]$ and $t_e - t_s \leq 3 \cdot d$. All correct nodes execute lines 4.a, 4.b and 4.c during the interval $[t_s, t_e]$. Therefore, the correct nodes' timers $\text{TIMER}_{main}, \text{TIMER}_{start}$ are reset. Hence, after t_e all correct nodes' values of $WantToPulse$ are 0, and hence any correct node that joins any $\mathcal{Q}$ instance after t_e has an input value $v_q = 0$. This holds until TIMER_{main} elapses at some correct node, that is until $t_s + \text{CYCLE}_{main}$. In other words, no correct node joins any $\mathcal{Q}$ instance in the interval $[t_e, t_s + \text{CYCLE}_{main}]$ with input value of 1.

By definition, t_2 is the last time that p invoked a pulse in the interval $[t_1, t_1 + \Delta_{max} + 3 \cdot d]$. Hence, after $t_2 + 3 \cdot d$ all correct nodes have invoked pulses, and hence have $WantToPulse$ as 0 for at least $\text{CYCLE}_{main} - 3 \cdot d$ time units. Therefore, in the interval $[t_2 + 3 \cdot d, t_2 + \text{CYCLE}_{main} - 3 \cdot d]$ no correct node joins any instance of$\mathcal{Q}$ with an input value of 1. Since $\text{CYCLE}_{main} \geq \Delta_{max} + 9 \cdot d$, it holds that $t_2 + 3 \cdot d \in [t_e, t_s + \text{CYCLE}_{main}]$, hence during the time interval $[t_e, t_2 + \text{CYCLE}_{main} - 3 \cdot d]$ no correct node joins any $\mathcal{Q}$ instance with an input value of 1. Hence, in the time interval $[t_e + \Delta_{max}, t_2 + \text{CYCLE}_{main} - 3 \cdot d + \Delta_{min}]$ no correct node invokes a pulse. Since $t_1 + 3 \cdot d + \Delta_{max} \geq t_e + \Delta_{max}$ and since t_2 is the last time p invoked a pulse before $t_1 + 3 \cdot d + \Delta_{max}$, it holds that p did not invoke a pulse in the time-interval $[t_2, t_2 + \text{CYCLE}_{main} - 3 \cdot d + \Delta_{min}]$.

Lastly, after $t_2 + 3 \cdot d$ time units have elapsed, all correct nodes have reset TIMER_{main} and TIMER_{start}. Since $\text{CYCLE}_{main} = \text{CYCLE}_{start}$, we have that when TIMER_{start} elapses at some correct node, then $WantToPulse = 1$ at that correct node. By time $t_2 + 3 \cdot d + \text{CYCLE}_{main}$ all correct nodes have set their value of $WantToPulse$ to 1. Consider the last correct node to do so, it starts executing $\mathcal{Q}$, as instructed by Line 1 (the elapsing of TIMER_{start}), and since the input values of all correct nodes are 1, it terminates with an output value of 1. This happens no later that $t_2 + 6 \cdot d + \text{CYCLE}_{main} + \Delta_{max}$. That is, p invokes a pulse no later than $t_3 = t_2 + 6 \cdot d + \text{CYCLE}_{main} + \Delta_{max}$. □

Note that the above lemma shows that a correct node p invokes a pulse in some pattern. That is, after each pulse there is a period of uncertainty, and afterwards there is a long period of no PULSEing. Then p invokes a pulse again, and so on. Note that in the above lemma, the TIMER_{end} was never used; it will be used in the following lemma, which claims the "uncertainty" period is of a constant

length, and at the end of it a pulse is invoked. This lemma will give us the required properties, since with it the PULSEing pattern of a correct node p will be constant, and since the system pulses-in-pattern, the entire PULSEing pattern of the system will be determined.

Lemma 7. *Consider t_1, t_2 to be as defined in Lemma 6. Once the system pulses-in-unison, the value of t_2 is in the interval $[t_1 + \Delta_{min} - 13 \cdot d, t_1 + \Delta_{max} + 3 \cdot d]$.*

Proof. According to Lemma 6, after the last pulse there is a silent period during which TIMER_{main} and TIMER_{start} tick away. Once they elapse (they both elapse together), the following happens. First, $WantToPulse$ is set to 1 (until the next pulse). Second, TIMER_{end} is reset; and third, a $\mathcal{Q}$ instance is initiated.

Since the system pulses-in-unison, after the last pulse (at time t_2) all correct nodes reset TIMER_{start}. This means that TIMER_{start} elapses at all correct nodes within a $3 \cdot d$ interval, which implies that TIMER_{end} elapses at all correct nodes within a $3 \cdot d$ interval. Consider the last node q to have had TIMER_{start} elapse (at time t'). No correct node has had TIMER_{start} elapse before time $t' - 3 \cdot d$, hence at time $t' - 3 \cdot d + \Delta_{min}$ all correct nodes still have $WantToPulse = 1$ (no $\mathcal{Q}$ instance managed to finish yet). Therefore, when q's TIMER_{end} elapses at time $t' - 10 \cdot d + \Delta_{min}$ (since $\text{CYCLE}_{end} = \Delta_{min} - 10 \cdot d$), all correct nodes are guaranteed to join q's $\mathcal{Q}(q)$ instance with input value of 1, and hence in time interval $[t' + \Delta_{min} - 10 \cdot d + \Delta_{min}, t' + \Delta_{min} - 7 \cdot d + \Delta_{max}]$ q invokes a pulse.

t'_1, t'_2 represent the same meaning as t_1, t_2, just for the "PULSEing cycle" that starts after t_2. Consider t'_1 to be the first time value at which a correct q' invokes a pulse after t_2 (note that according to Lemma 6, $t'_1 \in [t_2 + \text{CYCLE}_{main} - 3 \cdot d + \Delta_{min}, t_3]$). For q' to invoke a pulse, at least one correct node should have $WantToPulse = 1$ in the interval $[t'_1 - \Delta_{max}, t'_1 - \Delta_{min}]$. Since t'_1 is the first time some node invokes a pulse, and since $t' - 3 \cdot d$ is the first time some correct node has $WantToPulse = 1$ in the current "PULSEing cycle", we have that $t' \in [t'_1 - \Delta_{max}, t'_1 - \Delta_{min} + 3 \cdot d]$. Therefore, q invokes a pulse due to TIMER_{end}'s elapsing is in the interval $[t'_1 - \Delta_{max} + \Delta_{min} - 10 \cdot d + \Delta_{min}, t'_1 - 4 \cdot d + \Delta_{max}]$. Since $\Delta_{max} = \Delta_{min} + 3 \cdot d$, we have that the above time interval is $[t'_1 - 13 \cdot d + \Delta_{min}, t'_1 - 4 \cdot d + \Delta_{max}]$. This implies that $t'_2 \geq t'_1 - 13 \cdot d + \Delta_{min}$. Which means that starting from the first pulse, the next "PULSEing cycle" will have that $t_2 \geq t_1 - 13 \cdot d + \Delta_{min}$. □

The above lemmata show that if the system has been coherent for $\Delta_{\text{ERRATIC-PULSER}} := \Delta_{\mathcal{Q}} + 3 \cdot Cycle$, then all correct nodes invoke pulses together, and they have a distinctive PULSEing pattern: say a node invokes a pulse at some time t_1; during the interval $[t_1, t_1 + \Delta_{min} - 13 \cdot d]$ there could be some additional pulses, then during the interval $[t_1 + \Delta_{min} - 13 \cdot d, t_1 + \Delta_{max} + 3 \cdot d]$ at least one pulse is invoked, and then there is an interval of at least $\text{CYCLE}_{main} + \Delta_{min} - 3 \cdot d$ time units during which no pulse is invoked, and finally, within the next $12 \cdot d$ there will be new pulses and a new PULSEing "cycle" will start.

Note that the length of this "cycle" is bounded from below by $\Delta_{max} + \text{CYCLE}_{main} + \Delta_{min}$, and bounded from above by $\Delta_{max} + \text{CYCLE}_{main} + \Delta_{min} + 12 \cdot d$. In addition, notice that each such "cycle" starts with a "possibly noisy period" of length $\Delta_{max} + 3 \cdot d$, and ends with a "quiet period" of $\text{CYCLE}_{main} + \Delta_{min} - 3 \cdot d$

time. Since $\text{CYCLE}_{main} \geq \Delta_{max} + 9 \cdot d$, we have that the quiet period is at least Δ_{min} longer than the first period. This remark is important for the next section.

8 Creating the Balanced-Pulser

The above ERRATIC-PULSER synchronizes the correct nodes into some repetitive PULSEing pattern. However, to solve the PULSE-synchronization problem, an additional algorithm is required. We now present the BALANCED-PULSER algorithm, which starting from an arbitrary state, shortly after the system is coherent, produces pulses approximately once in a *Cycle*, despite the permanent presence of *Byzantine* nodes.

Algorithm **Balanced-Pulser** */* executed at node p */*

1. **execute** an instance $\mathcal{A}$ of ERRATIC-PULSER in the background;
2. **when** $\mathcal{A}$ produces a pulse
   if $\mathcal{A}$ has not produced a pulse for at least $\text{CYCLE}_{main} + \Delta_{min} - 3 \cdot d$ time, invoke a pulse.

Fig. 3. An algorithm solving the PULSE-synchronization problem

Theorem 1. *Algorithm* BALANCED-PULSER *solves the* PULSE*-synchronization problem in a self-stabilizing and* Byzantine *tolerant manner.*

Proof. Once the system is coherent for $\Delta_{\mathcal{Q}}$ time, by Lemma 4 the system pulses-in-unison. Hence, each time a correct node sees $\mathcal{A}$ PULSEing, within $3 \cdot d$ time units all other correct nodes see the same. In addition, by Lemma 6 and Lemma 7, the pulses that $\mathcal{A}$ produces have a distinct pattern. That is, a pulse, then a period of length $\Delta_{max} + 3 \cdot d$ with possible pulses and a period of length $\text{CYCLE}_{main} + \Delta_{min} - 3 \cdot d$ with no pulses. Then, within $12 \cdot d$, another pulse.

If a correct node hasn't heard $\mathcal{A}$ producing a pulse for $\text{CYCLE}_{main} + \Delta_{min} - 3 \cdot d$ time, it must mean that $\mathcal{A}$ has undergone the "quiet period", since the "possible noisy period" is short. Hence, the next PULSE produced must be the beginning of a new "cycle". Therefore, all correct nodes invoke pulses together in BALANCED-PULSER. In addition, all correct nodes invoke pulses only at the beginning of a "cycle", and they invoke pulses $3 \cdot d$ apart of each other. Since all correct nodes invoke pulses only at the beginning of a "cycle", we need only to argue about the length of the "cycle".

According to the lemmata in the previous section, the difference between the "long-cycle" and the "short-cycle" is at most $12 \cdot d$ time units. Setting $\text{CYCLE}_{min} := \Delta_{max} + \text{CYCLE}_{main} + \Delta_{min}$, and $\text{CYCLE}_{max} := \text{CYCLE}_{min} + 12 \cdot d$, we have that the system is in a synchronized_pulsing_state. That is, starting from any state, the system reaches a synchronized_pulsing_state; this proves *convergence*. In addition, according to the previous section, the PULSEing pattern remains as long is the system is coherent, thus *closure* also holds. □

The convergence time of BALANCED-PULSER is the same as the convergence of ERRATIC-PULSER + *Cycle*; that is, $\Delta_{\mathcal{Q}} + 4 \cdot Cycle$ time units.

9 Discussion

Time complexity: Once the system has become coherent, the BALANCED-PULSER algorithm converges in $O(f) + O(Cycle)$ time.

Message complexity: The BALANCED-PULSER algorithm executes $2 \cdot (n - f)$ SS-BYZ-Q instances each *Cycle*. Since SS-BYZ-Q has $O(f \cdot n^2)$ message complexity, then the message complexity becomes $O(f \cdot n^3)$ per CYCLE.

Executing fewer ss-Byz-Q: The main feature of ERRATIC-PULSER is that "eventually there will be a correct node that executes SS-BYZ-Q". As presented, ERRATIC-PULSER has each correct node execute SS-BYZ-Q once its timers elapse. The algorithm can be adapted such that only $f + 1$ of the nodes (predetermined and considered as part of the program, not memory) can invoke $\mathcal{Q}$. Since there will always be a correct node that invokes $\mathcal{Q}$, the correctness of the algorithm holds. This reduces the message complexity to $O(f^2 \cdot n^2)$.

Clock synchronization: The Digital clock synchronization problem consists of having all correct nodes agree on an integer value that progresses linearly with time. To build a digital clock synchronization algorithm using a PULSEing algorithm, all that is needed is to execute an agreement on the next clock's value each time a pulse is invoked. Setting the cycle of the PULSE to be long enough for the agreement algorithm to terminate, ensures that all correct nodes will agree on the clock value, and advance it appropriately. Note that the convergence time of such an algorithm is the convergence time of the underlying PULSE algorithm, plus an additional $cycle_{max}$time units. See [4] for a more detailed discussion.

Arbitrary *Cycle* values: According to the constraints of the BALANCED-PULSER algorithm, *Cycle* must be larger than $2 \cdot \Delta_{max} + \Delta_{min} + 9 \cdot d$ time units. For the purpose of clock synchronization it is enough to have *Cycle* in the order of Δ; for example, $Cycle = 5 \cdot \Delta$ would suffice for a linear convergence of the digital clock synchronization algorithm.

However, if one wishes to use PULSEing for other reasons, it is desired to be able to PULSE in any *Cycle*. To PULSE every $Cycle' < 2 \cdot \Delta_{max} + \Delta_{min} + 9 \cdot d$, set *Cycle* to be some multiplication of $Cycle'$ such that it falls within the constraints. Now, each time that BALANCED-PULSER produces a pulse, reset a timer of $Cycle'$ long, and when it elapses, invoke a pulse and reset the timer again. The PULSEing pattern will be a pulse by BALANCED-PULSER every *Cycle* and $Cycle/Cycle'$ pulses in between. This scheme is similar to what is done in [18]. The tricky part is to notice that if a pulse is invoked less than $Cycle'$ time before a PULSE by BALANCED-PULSER then the timer for the "small" pulses is reset, and hence a pulse is invoked again only in $Cycle'$ time units. Note that the difference between $cycle_{max}$ and $cycle_{min}$ is still $12 \cdot d$, hence there is no meaning to having $Cycle' \leq 12 \cdot d$. That is, $Cycle'$ should always be larger than $12 \cdot d$ time units.

References

1. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
2. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
3. Daliot, A., Dolev, D.: Self-stabilization of byzantine protocols. In: Tixeuil, S., Herman, T. (eds.) SSS 2005. LNCS, vol. 3764, Springer, Heidelberg (2005)
4. Daliot, A., Dolev, D., Parnas, H.: Linear time byzantine self-stabilizing clock synchronization. In: Papatriantafilou, M., Hunel, P. (eds.) OPODIS 2003. LNCS, vol. 3144, Springer, Heidelberg (2004), A corrected version appears in `http://arxiv.org/abs/cs.DC/0608096`
5. Sakurai, Y., Ooshita, F., Masuzawa, T.: A self-stabilizing link-coloring protocol resilient to byzantine faults in tree networks. In: Higashino, T. (ed.) OPODIS 2004. LNCS, vol. 3544, pp. 283–298. Springer, Heidelberg (2005)
6. Nesterenko, M., Arora, A.: Local tolerance to unbounded byzantine faults. In: IEEE SRDS 2002, pp. 22–31 (2002), `citeseer.ist.psu.edu/nesterenko02local.html`
7. Nesterenko, M., Arora, A.: Dining philosophers that tolerate malicious crashes. In: 22nd Int. Conference on Distributed Computing Systems (2002)
8. Dolev, S., Welch, J.L.: Self-stabilizing clock synchronization in the presence of byzantine faults. Journal of the ACM 51(5), 780–799 (2004)
9. Hoch, E.N., Dolev, D., Daliot, A.: Self-stabilizing byzantine digital clock synchronization. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, Springer, Heidelberg (2006)
10. Daliot, A., Dolev, D.: Self-stabilizing byzantine agreement. In: PODC 2006. Proc. of the Twenty-fifth ACM Symposium on Principles of Distributed Computing, Denver, Colorado (July 2006)
11. Liskov, B.: Practical use of synchronized clocks in distributed systems. In: Proceedings of 10th ACM Symposium on the Principles of Distributed Computing, ACM Press, New York (1991)
12. Arora, A., Dolev, S., Gouda, M.G.: Maintaining digital clocks in step. Parallel Processing Letters 1, 11–18 (1991)
13. Dolev, S.: Possible and impossible self-stabilizing digital clock synchronization in general graphs. Journal of Real-Time Systems 12(1), 95–107 (1997)
14. Dolev, S., Welch, J.L.: Wait-free clock synchronization. Algorithmica 18(4), 486–511 (1997)
15. Papatriantafilou, M., Tsigas, P.: On self-stabilizing wait-free clock synchronization. Parallel Processing Letters 7(3), 321–328 (1997)
16. Herman, T.: Phase clocks for transient fault repair. IEEE Transactions on Parallel and Distributed Systems 11(10), 1048–1057 (2000)
17. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2), 374–382 (1985)
18. Dolev, D., Hoch, E.N.: On self-stabilizing synchronous actions despite byzantine attacks. In: DISC2007. LNCS, vol. 4731, pp. 193–207. Springer, Heidelberg (2007)
19. Fischer, M.J., Lynch, N.A., Merritt, M.: Easy impossibility proofs for distributed consensus problems. Distributed Computing 1, 26–39 (1986)
20. Daliot, A., Dolev, D.: Self-stabilizing byzantine pulse synchronization. Technical report, Cornell ArXiv, (August 2005), `http://arxiv.org/abs/cs.DC/0608092`
21. Freiling, F.C., Ghosh, S.: Code stabilization. In: Tixeuil, S., Herman, T. (eds.) SSS 2005. LNCS, vol. 3764, Springer, Heidelberg (2005)

A The Use of SS-BYZ-AGREE

The mode of operation of the SS-BYZ-AGREE, a self-stabilizing Byzantine agreement protocol presented in [10] 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, an (*Initiator*, p, *) message serves as the initialization message.

When the protocol terminates, the SS-BYZ-AGREE protocol returns (in each correct 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 has 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 time at which the local clock of q reads τ_q.

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 within a finite time;

The proof uses the following properties of the SS-BYZ-AGREE protocol ([10]):

Timeliness-Agreement Properties:

1. (agreement) For every two correct nodes q and q' that decide $\langle p, m, \tau_q^p \rangle$ and $\langle p, m, \tau_{q'}^p \rangle$ at local times τ_q and $\tau_{q'}$ respectively: $|rt(\tau_q) - rt(\tau_{q'})| \leq 3d$.
2. (validity) If all correct nodes invoked the protocol in the 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 $t_0 - d \leq rt(\tau_q) \leq t_0 + 3d$.
3. (termination) The protocol terminates within Δ time units following its explicit invocation, and within $\Delta + 7d$ time units, in case it was not explicitly invoked[7].
4. (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 time at which a correct node invoked SS-BYZ-AGREE in the earlier agreement, and $\bar{t}_1$ is the earliest time that SS-BYZ-AGREE was invoked by a correct node in the later agreement.

[7] $\Delta := 7(2f + 3)d$.

Chapter 3

Fast Self-Stabilizing Byzantine Tolerant Digital Clock Synchronization

Michael Ben-Or and Danny Dolev and Ezra N. Hoch, Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing (PODC '08), Toronto, Canada, Pages: 385-394, Aug. 2008.

Fast Self-Stabilizing Byzantine Tolerant Digital Clock Synchronization*

Michael Ben-Or†
Hebrew University
benor@cs.huji.ac.il

Danny Dolev
Hebrew University
dolev@cs.huji.ac.il

Ezra N. Hoch
Hebrew University
ezraho@cs.huji.ac.il

ABSTRACT

Consider a distributed network in which up to a third of the nodes may be *Byzantine*, and in which the non-faulty nodes may be subject to transient faults that alter their memory in an arbitrary fashion. Within the context of this model, we are interested in the digital clock synchronization problem; which consists of agreeing on bounded integer counters, and increasing these counters regularly.

It has been postulated in the past that synchronization cannot be solved in a *Byzantine* tolerant and self-stabilizing manner. The first solution to this problem had an expected exponential convergence time. Later, a deterministic solution was published with linear convergence time, which is optimal for deterministic solutions.

In the current paper we achieve an expected constant convergence time. We thus obtain the optimal probabilistic solution, both in terms of convergence time and in terms of resilience to *Byzantine* adversaries.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Algorithms, Reliability, Theory

Keywords

Distributed computing, fault tolerance, self-stabilization, Byzantine failures, clock synchronization, digital clock synchronization.

*This work was funded in part by Israel Science Foundation.
†Incumbent of the Jean and Helena Alfassa Chair in Computer Science.

PODC'08, August 18–21, 2008, Toronto, Ontario, Canada.

1. INTRODUCTION

Clock synchronization is a fundamental building block of distributed systems. The vast majority of distributed tasks require some sort of synchronization; clock synchronization is a very straightforward and intuitive tool for supplying this. Thus, it is desirable to have a highly robust clock synchronization mechanism available. A typical self-stabilizing algorithm seeks to re-establish synchronization once lost; a *Byzantine* tolerant algorithm assumes synchronization is never lost and focuses on containing the influence of the permanent presence of faulty nodes. The robustness we provide achieves both synchronization and *Byzantine* tolerance.

We consider a system in which the nodes execute in lockstep by regularly receiving a common "pulse" (or "tick" or "beat" - we will use the term "beat" in order to stay clear of any confusion with "pulse synchronization" or "clock ticks"). The digital clock synchronization problem seeks to ensure that all correct nodes eventually hold the same value for the beat counter (*digital clock*) and as long as enough nodes remain correct, they will continue to hold the same value and to increase it by "one" at each beat.

Clock synchronization in a similar model has earlier been denoted as "digital clock synchronization" (see [1, 8, 16]) or "synchronization of phase-clocks" (see [14]); we will simply use the term "clock synchronization". However, these solutions do not consider *Byzantine* adversaries. In recent years there has been major advancement in self-stabilizing digital clock synchronization algorithms that are *Byzantine* tolerant. Starting from [9] which gives an expected exponential convergence time, and continuing with [7] which has deterministic linear convergence time. The current work continues this line, by providing a solution with expected constant convergence time.

In the classical "*Byzantine* field", it was shown that deterministic *Byzantine* agreement protocols require linear running time (see [13]). Randomization can be used to break this linear-time barrier, for example see [2], [4], [3] and [12]. The main idea behind such algorithms is to agree on a common coin with some probability. Each time all non-faulty nodes have the same random bit, they have a certain probability of reaching an agreement (on the input values).

The structure of such agreement protocols implies that their expected convergence time depends highly on the probability p that the common-coin algorithm will produce the same coin at all non-faulty nodes; specifically, their expected convergence time is $O(\frac{1}{p})$.

Our solution can use any common-coin protocol[1] operating in a synchronous network, provided that the protocol has a constant probability of producing the same random bit at all non-faulty nodes upon termination (such as the protocol described in [12]). By re-executing this protocol anew each round, a "stream" of random bits is produced - one bit each round. Combining this with a single round "agreement phase" produces the basis for our self-stabilizing *Byzantine* tolerant clock synchronization.

The main contribution of the paper is a self-stabilizing, *Byzantine*-tolerant digital clock synchronization algorithm that converges in expected constant time; the algorithm is optimal in its convergence time and in its resiliency (it is resilient to $f < \frac{n}{3}$ *Byzantine* nodes.)

Aside from the clock synchronization algorithm, we show how to turn a non-self-stabilizing common-coin algorithm into a self-stabilizing one that produces a random bit every round.

1.1 Previous Self-stabilizing Byzantine Tolerant Clock Synchronization Algorithms

Previously, two different models were considered within the scope of the self-stabilizing *Byzantine* tolerant clock synchronization problem. The Global Beat System model (also known as the "synchronous" model), and the Bounded-Delay model (also known as the "semi-synchronous" model). In the synchronous model there is some device which is connected to all nodes and simultaneously delivers a signal to all nodes regularly. The semi-synchronous model assumes a bound on the message delivery time between two nodes.

The clock synchronization problem is slightly different in the different models: in the synchronous model, all nodes have an integer counter, and all non-faulty nodes must agree on the counter's value and increment it every time they receive a signal from the global device; this problem is sometimes referred to as "*digital* clock synchronization". In the semi-synchronous model, each node is equipped with a physical clock that can only measure the passing of time (but cannot tell the current time). All non-faulty nodes' physical clocks advance (more or less) at the same rate. In this setting, the clock synchronization problem consists of all non-faulty nodes having clock variables s.t. the difference between any two non-faulty nodes' clocks is bounded. Clearly, it is easier to solve the clock synchronization problem in the synchronous model.

The self-stabilizing *Byzantine*-tolerant clock synchronization problem was first tackled in [10] (see [9] for the full version). It was solved in both models using probabilistic algorithms, with expected exponential convergence time. The *Byzantine* resiliency supported by [10] is optimal.

Later on, a series of papers addressed the clock synchronization problem in both models using deterministic algorithms. In the bounded-delay model the first deterministic polynomial solution was presented in [5]. The polynomial convergence was obtained using a pulse synchronization protocol as a building block.[2] [6] provides an optimal (deterministic) solution in terms of convergence time and *Byzantine* resiliency for the bounded delay model, using a much simpler pulse producing protocol.

In the synchronous model, [15] solved the digital clock synchronization problem with deterministic linear convergence time. The main drawback of [15] is its *Byzantine*-resiliency, which was limited to $f < \frac{n}{4}$, as opposed to the optimal $f < \frac{n}{3}$. The result of [7] supports $f < \frac{n}{3}$ *Byzantine* nodes, while maintaining the deterministic linear convergence time. In [7] the clock synchronization problem is solved via an underlying "pulse algorithm" (see [7] for more information).

In the current work we solve the digital clock synchronization problem in the synchronous model, in a probabilistic manner, with expected constant convergence time. This solution is optimal in terms of its convergence time and in terms of its *Byzantine* resiliency.

Table 1 presents a summary of previous results as compared to the current paper.

Table 1: Summary of previous results

Paper	Model	Convergence	Resiliency
[10]	sync, probabilistic	$O(2^{2(n-f)})$	$f < \frac{n}{3}$
[10]	semi-sync, probabilistic	$O(n^{6(n-f)})$	$f < \frac{n}{3}$
[15]	sync, deterministic	$O(f)$	$f < \frac{n}{4}$
[7]	sync, deterministic	$O(f)$	$f < \frac{n}{3}$
[6, 5]	semi-sync, deterministic	$O(f)$	$f < \frac{n}{3}$
current	sync, probabilistic	$O(1)$	$f < \frac{n}{3}$

The paper is structured as follows: in Section 2 the model is presented along with a self-stabilizing coin-flipping protocol. Section 3 defines the k-Clock problem and solves it for $k = 2$, Section 4 solves the 4-Clock problem, Section 5 includes a solution to the k-Clock problem for any k; and lastly, Section 6 concludes with a discussion of the results.

2. MODEL AND DEFINITIONS

Our model consists of a distributed network of n nodes, which are fully connected to each other. Nodes communicate via message passing, and are all connected to a global beat system that provides "beats" at regular intervals; each beat reaches all nodes simultaneously. Each message m that is sent from p to q at beat r is guaranteed to reach q before beat $r+1$. In the following discussion, the term "round" will be used in the context of non-self stabilizing synchronous algorithms, and "beat" will be used when talking about self-stabilizing synchronous algorithms. Notice that "beat" can be used to denote the signal received from the global beat system, as well as the interval between such two signals; when not stated otherwise, "beat" will refer to the second meaning.

A percentage of the nodes may be *Byzantine*, and behave arbitrarily; the presented algorithms are resilient to $f < \frac{1}{3} \cdot n$ such faulty nodes. We assume an information theoretic adversary with private channels. That is, the *Byzantine* adversary has access to all communications between faulty and non-faulty nodes; however, it does not have access to communications among non-faulty nodes. Moreover, the non-faulty nodes cannot use any computational assumptions (e.g. signatures) to guard against the adversary.

In addition to the faulty nodes, non-faulty nodes may undergo transient faults that change their memory in an arbitrary manner. Any resilient protocol is thus required to

[1]Due to self-stabilizing issues, the common coin protocol cannot rely on special initialization of all non-faulty nodes, such as assumed in [17].

[2]The pulse synchronization protocol in the original version of [5] had a flaw, but any other pulse synchronization protocols can be used, as appears in the corrected version there.

converge from any memory state. More specifically, following a long-enough period without any new transient faults, the system is required to converge to a state in which all correct nodes have synchronized digital clocks.

DEFINITION 2.1. *A node is* **non-faulty** *when it follows the given protocol. A node is* **faulty** *if it violates its protocol in any way. The terms Faulty and* Byzantine *will be used interchangeably.*

At times of transient failures one cannot assume anything about the state of any node, and the communication network may also behave erratically. Eventually the system becomes coherent again. In such a situation the system may find itself in an arbitrary state.

DEFINITION 2.2. *The communication network is* **non-faulty** *when the following conditions hold:*

1. *A message by a correct node p sent upon a receipt of a beat from the global beat system, arrives (and is processed) at its destination before the following beat is issued by the global beat system;*
2. *The sender's identity and the message context of any message received are not tampered with.*
3. *A message received by p was sent by some node no more than one beat ago. That is, "phantom" messages are not delivered.*

In real-world networks, it may take some time for the communication network to overcome transient faults. Specifically, the communication networks' buffers may contain messages that were not recently sent by any currently operating node, and the network may eventually deliver them. We consider the communication network to be *non-faulty* only after all of these "phantom" messages have been delivered or cleared away.

According to the above definition, once the network is non-faulty, it adheres to the global-beat-system model. Which means that messages cannot be lost and old messages cannot be stored for an arbitrarily long time.

Since a non-faulty node may find itself in an arbitrary state, there should be some period of continuous non-faulty behavior before it can be considered "correct".

DEFINITION 2.3. *A non-faulty node is considered* **correct** *only if it remains non-faulty for Δ_{node} beats during which the entire communication network is non-faulty.*[3]

Intuitively, for the system to converge to its desired state, it is required that a "critical mass" of non-faulty nodes have been clear of transient faults for a "long enough" period of time.

DEFINITION 2.4. *The system is* **coherent** *when the communication network is non-faulty and there are $n-f$ correct nodes.*

DEFINITION 2.5. *A beat interval $T = [r_1, r_2]$ is a* **coherent** *beat interval if during T the system is coherent and there is a set of the same $n-f$ nodes that are correct throughout T.*[4]

[3]The assumed value of Δ_{node} in the current paper will be defined later.

[4]The indexing of the beats is not available to the nodes, it is used only for presentation purposes.

Note that in the above definition, the set of non-faulty nodes may not change from beat to beat. Alternatively, we could require that at each beat $r \in T$ there must be $n-f$ correct nodes, but they do not need to be the same correct nodes each beat. However, such a definition would complicate the proofs. The algorithms presented below are valid under both definitions; for the sake of clarity, the stronger assumption is used.

REMARK 2.1. *The values of n and f are fixed constants and are considered part of the "code" and therefore non-faulty nodes cannot initialize with arbitrary values for these constants.*

2.1 Common Coin-Flipping

Our clock synchronization algorithm uses a common coin-flipping (coin-flipping for short) algorithm as a building block. A coin-flipping algorithm is a distributed algorithm that, with some constant probability, produces an output bit that is common to all non-faulty nodes. Different coin-flipping algorithms exist (see [12] and [11]), with different properties. Our formalization and requirements of a coin-flipping algorithm are as follows:

DEFINITION 2.6. *An algorithm $\mathcal{A}$ is said to be a probabilistic coin-flipping algorithm if $\mathcal{A}$ has the following properties:*
*(***model***): $\mathcal{A}$ operates in a synchronous network, communicates only via message passing, and is resilient to $f < \frac{n}{3}$* Byzantine *nodes;*
*(***termination***): There exists a constant $\Delta_{\mathcal{A}}$, such that $\mathcal{A}$ terminates within $\Delta_{\mathcal{A}}$ rounds of sending-and-receiving messages;*
*(***binary output***): The output of $\mathcal{A}$ at each node i is d_i, $d_i \in \{0,1\}$;*
*(***event** E_0*): The event that all non-faulty nodes have the same output "0", occurs with constant probability $p_0 > 0$;*
*(***event** E_1*): The event that all non-faulty nodes have the same output "1", occurs with constant probability $p_1 > 0$;*
*(***unpredictability***): If either E_0 or E_1 occurs, then the probability of any f nodes to predict the output of $\mathcal{A}$ by the end of round $\Delta_{\mathcal{A}} - 1$ is no more than $1 - \min\{p_0, p_1\}$.*

Intuitively, when executing a probabilistic coin-flipping algorithm $\mathcal{A}$ there is a constant probability that all non-faulty nodes have the same output value. Moreover, the *Byzantine* nodes do not "know" which of the possible outputs will be the common output until the very last round.

REMARK 2.2. *The probability space of the above definition is valid for any choice of* Byzantine *adversary. That is, an algorithm $\mathcal{A}$ is a probabilistic coin-flipping algorithm, if for any* Byzantine *adversary, the properties of Definition 2.6 hold regarding all possible runs. (for more details see [12]).*

In the following section, a probabilistic coin-flipping algorithm $\mathcal{A}$ is assumed to be "self-contained", in the sense that multiple invocations of $\mathcal{A}$ do not affect the probability of the events E_0 or E_1 occurring within each invocation, or the probability of *Byzantine* nodes predicting the output before round $\Delta_{\mathcal{A}}$ of each invocation. This "self-contained"-ness is required to allow multiple concurrent executions of $\mathcal{A}$ to run properly.

Ensuring such "self-contained"-ness could be a problem in asynchronous systems. However, it is easy to implement

Algorithm SS-BYZ-COIN-FLIP /* executed at each node, each beat */
/* $\mathcal{A}$ is a probabilistic coin-flipping algorithm */
/* the A_i's are $\Delta_{\mathcal{A}}$ instances of $\mathcal{A}$ */

On beat (signal from global beat system):

1. **For** $i := 1$ to $\Delta_{\mathcal{A}}$
   execute the ith round of A_i;
2. **Output** the value of $A_{\Delta_{\mathcal{A}}}$;
3. **For** $i := 1$ to $\Delta_{\mathcal{A}} - 1$
   $A_{i+1} := A_i$;
4. **Initialize** A_1 to be a new instance of $\mathcal{A}$;

Figure 1: A self-stabilizing coin-flipping algorithm.

in the global-beat-system model when each instance of $\mathcal{A}$ terminates within a finite number of rounds: simply add a "session number" to each instance of $\mathcal{A}$, and differentiate messages of co-executing instances using this session number. Since only a finite number of instances are concurrently executed at any round, the session numbers can be "recycled", thus avoiding problems of infinite counters in the setting of self-stabilization.

OBSERVATION 2.1. *The common coin protocol of [12] adheres to Definition 2.6. In [12] the common coin protocol is based on graded verifiable secret sharing (GVSS), which has 3 "phases":* share, decide, recover. *Up until the last phase, the secret is unrecoverable by any set of f or less nodes. Moreover, the* recover *phase is one round long. Thus, the "*unpredictability*" property of Definition 2.6 holds.*

[12]'s common coin protocol executes a GVSS protocol for each node in the system. However, the last step of the common coin protocol consists of executing the recover *phase of all the GVSS instances, which conserves the property that the output of the common coin is unpredictable by any set of $\leq f$ nodes, until the very last round.*

Lastly, the protocols of [12] operate in a synchronous model and are tolerant to $f < \frac{n}{3}$ Byzantine *nodes. Moreover, the adversarial model assumed in [12] allows "rushing"; thus, when using [12], our solution is also tolerant to rushing.*

REMARK 2.3. *The protocol of [12] requires the values of n, f as input, as well as additional constants; for example, the secret sharing protocol of [12] requires a prime $p > n$. These constants are assumed to be part of the "code" and non-faulty nodes do not initialize with arbitrary values of these constants.*[5]

2.2 Self-stabilizing Coin-flipping

When considering a system that is self-stabilizing, round numbers become a problematic notion, since different nodes may have different values as their current "round number". Thus, statements such as "$\mathcal{A}$ terminates within $\Delta_{\mathcal{A}}$ rounds" require some explanation. To this end, we define pipelined probabilistic coin-flipping, and later use it to define a self-stabilizing coin-flipping algorithm.

[5]These constants can be computed in a single way given the value of n (for example, let p be the smallest prime that is larger than n). Thus, this assumption does not weaken the result.

DEFINITION 2.7. *An algorithm $\mathcal{B}$ is said to be a pipelined probabilistic coin-flipping algorithm if $\mathcal{B}$ has the following properties:*
*(***model***): $\mathcal{B}$ operates in a synchronous network, communicates only via message passing, and is resilient to $f < \frac{n}{3}$* Byzantine *nodes;*
*(***binary output***): Each round, the output of $\mathcal{B}$ at each node i is d_i, $d_i \in \{0, 1\}$;*
*(***event** E_0*): Each round, the event that all non-faulty nodes have the same output value "0", occurs with constant probability $p_0 > 0$;*
*(***event** E_1*): Each round, the event that all non-faulty nodes have the same output value "1", occurs with constant probability $p_1 > 0$;*
*(***unpredictability***): The probability that either E_0 or E_1 occurs at some round is independent of the previous rounds. If either E_0 or E_1 occurs, then the probability that any f nodes will predict the output of $\mathcal{B}$ at round r by the end of round $r - 1$ is no more than $1 - \min\{p_0, p_1\}$.*

REMARK 2.4. *The "unpredictability" property means that as far as the adversary can know (considering all information it has access to) the output of the random bit at each round is independent of previous rounds. However, this is not the usual meaning of "independent", as if the adversary has all the information of all correct nodes, the random bits can be predicted. (see [12] for more information).*

In the rest of this paper we use the term "independent" to mean "as far as the adversary can tell, the events are independent".

Informally, the above definition states that at every round, with constant probability, all non-faulty nodes agree on a common random bit.

DEFINITION 2.8. *An algorithm $\mathcal{C}$ is said to be a self-stabilizing probabilistic coin-flipping algorithm if $\mathcal{C}$ has the following properties:*
*(***model***): $\mathcal{C}$ operates in a self-stabilizing synchronous network (i.e. with a global beat system), communicates only via message passing, and is resilient to $f < \frac{n}{3}$* Byzantine *nodes;*
*(***convergence***): Starting from any arbitrary state, $\mathcal{C}$ converges within $\Delta_{\mathcal{C}}$ beats to be a pipelined probabilistic coin-flipping algorithm.*

Given an algorithm $\mathcal{A}$ that is a probabilistic coin-flipping algorithm, one can construct an algorithm $\mathcal{C}$ that is a self-stabilizing probabilistic coin-flipping algorithm. See Figure 1.

Algorithm SS-BYZ-2-CLOCK /* executed at node u each beat */
/* $\mathcal{C}$ is self-stabilizing probabilistic coin-flipping algorithm */

On beat (signal from global beat system):

1. **broadcast**[a] $u.clock$; /* $u.clock \in \{0,1,\perp\}$ */
2. **execute** a single beat of $\mathcal{C}$, and set $rand$ to be the output of $\mathcal{C}$;
3. **consider** each message with "$\perp$" as carrying the value $rand$; /* $rand \in \{0,1\}$ */
4. **set** maj to be the value that appeared the most, and $\#maj$ the number of times it appeared; /* $maj \in \{0,1\}$ */
5. **if** $\#maj \geq n - f$ then $u.clock := 1 - maj$;
6. **else** $u.clock := \perp$;

[a]In the context of this paper, "broadcast" means "send the message to all nodes". (We **do not** assume broadcast channels.)

Figure 2: An algorithm that solves the 2-Clock problem.

LEMMA 1. *Given a probabilistic coin-flipping algorithm $\mathcal{A}$, the algorithm* SS-BYZ-COIN-FLIP *is a self-stabilizing coin-flipping algorithm, with convergence time $\Delta_{\text{SS-BYZ-COIN-FLIP}} = \Delta_{\mathcal{A}}$.*

PROOF. Consider a system that has been coherent for $\Delta_{\mathcal{A}}$ beats, and a set of $n-f$ non-faulty nodes $\mathcal{G}$, where each node has been non-faulty for $\Delta_{\mathcal{A}}$ beats. The nodes in $\mathcal{G}$, when executing Line 2, output the value of a probabilistic coin-flipping algorithm that has been initialized and executed properly for $\Delta_{\mathcal{A}}$ rounds, and therefore its properties hold. This situation continues to hold for as long as the nodes in $\mathcal{G}$ are not subject to transient faults. SS-BYZ-COIN-FLIP therefore converges, within $\Delta_{\mathcal{A}}$ beats, to become a pipelined probabilistic coin-flipping algorithm. □

THEOREM 1. *There exists a self-stabilizing probabilistic coin-flipping algorithm, with constant stabilization time.*

PROOF. Denote the coin-flipping algorithm in [12] by $\mathcal{OC}$; $\mathcal{OC}$ has the properties of a probabilistic coin-flipping algorithm, as defined in Definition 2.6 (see Observation 2.1). Thus, the algorithm SS-BYZ-COIN-FLIP (when executed with $\mathcal{A} := \mathcal{OC}$) is a self-stabilizing probabilistic coin-flipping algorithm, according to Lemma 1. In addition, $\Delta_{\mathcal{OC}}$ is constant, leading to a constant stabilization time of SS-BYZ-COIN-FLIP, as required. □

3. THE DIGITAL CLOCK SYNCHRONIZATION PROBLEM

In the digital clock synchronization problem each node u has an integer variable $u.clock$ representing the node's clock-value. The goal is to synchronize all correct nodes' *clock* variables.

DEFINITION 3.1. *A system is* **clock-synched** *at beat r with value $Clock(r)$, if at the end of beat r, all correct nodes have the same clock-value, and it is equal to $Clock(r)$.*

DEFINITION 3.2. *The $k-Clock$ problem consists of the following:*
(convergence) starting from any state, eventually (at some beat r) the system becomes clock-synched with value $Clock(r)$;
(closure) from this point on the system stays clock-synched s.t. at beat $r+i$ it is clock-synched with value $Clock(r)+i$ mod k.

3.1 Overview of the Solution

The first step is to construct a 2-Clock algorithm SS-BYZ-2-CLOCK using the self-stabilizing coin-flipping algorithm SS-BYZ-COIN-FLIP. Then, by using 2 instances of SS-BYZ-2-CLOCK, a 4-clock algorithm SS-BYZ-4-CLOCK is built. Using SS-BYZ-4-CLOCK one can have four send-and-receive "phases" before a wrap-around of the clock value occurs. Using SS-BYZ-4-CLOCK, SS-BYZ-CLOCK-SYNC is constructed, which runs SS-BYZ-4-CLOCK, and sends messages each time SS-BYZ-4-CLOCK's clock value changes. Thus, between two wraparounds of SS-BYZ-4-CLOCK's clock it is possible to try to achieve an agreement on the clock value of SS-BYZ-CLOCK-SYNC, with a constant probability of success. This allows SS-BYZ-CLOCK-SYNC to solve the k-Clock problem for any k, in an expected constant number of beats.

Observe that each algorithm uses the previous algorithms as building blocks. On a beat received from the global-beat-system, each algorithm performs a step in each of the appropriate building blocks. We call such a step "execution of a single beat" of the relative algorithm.

3.2 Solving the 2-Clock Problem

Let $\mathcal{C}$ be a self-stabilizing probabilistic coin-flipping algorithm. At each beat, $\mathcal{C}$ produces some random bit. In SS-BYZ-2-CLOCK (see Figure 2), $\mathcal{C}$ is executed in the background, and each beat $rand$ holds the random output bit of $\mathcal{C}$. The algorithm SS-BYZ-2-CLOCK requires that $\Delta_{node} \geq \Delta_{\mathcal{C}}$.

REMARK 3.1. *Consider some beat r. Notice that in the algorithm of Figure 2 the value of* rand *at beat r is used to "replace" $\perp$ values sent in beat $r-1$. One may try to use the value of* rand *at beat $r-1$, and have each node send "*rand*" instead of "$\perp$" (during beat $r-1$). The problem with such a solution is that the* Byzantine *nodes can "decide" which value they send at beat $r-1$ according to the result of* rand *at beat $r-1$. This way we lose the power of randomization, since the* Byzantine *nodes' action can depend on the value of the random bit.*

To avoid this, rand *of beat r is used only after all nodes (including the* Byzantine *ones) sent their messages in beat $r-1$. That is,* rand *is used only after the* Byzantine *nodes are committed to the values they sent. Thus,* rand *is independent of the clock values sent by* Byzantine *nodes.*

OBSERVATION 3.1. *Consider two vectors $\vec{A}, \vec{B}$ of length n that differ in at most f entries, where $n > 3f$. If $\vec{A}$ contains*

$n-f$ copies of v_A, and $\vec{B}$ contains $n-f$ copies of v_B, then $v_A = v_B$.

DEFINITION 3.3. *Let T be a coherent beat interval. For any $r \in T$:*
$clocks_r^{start}$ is the set of all clock values of correct nodes at the beginning of beat r.
$clocks_r^{end}$ is the set of all clock values of correct nodes at the end of beat r.

Let $\mathcal{G}$ be the set of correct nodes during beat $r \in T$; recall that $\mathcal{G}$ does not change throughout T (see Definition 2.5). $clocks_r^{start} := \{u.clock \mid u \in \mathcal{G}\}$ before the execution of Line 1, and $clocks_r^{end} := \{u.clock \mid u \in \mathcal{G}\}$ after the execution of Line 6. Notice that $clocks_r^{end} = clocks_{r+1}^{start}$. Note also that the system is clock-synched at beat r with value $v \in \{0,1\}$ if (and only if) $clocks_r^{end} = \{v\}$.

LEMMA 2. *Let T be a coherent interval. If at some beat $r \in T$, $clocks_r^{start} = \{v\}$, (where $v \in \{0,1\}$), then $clocks_r^{end} = \{1-v\}$.*

PROOF. If $clocks_r^{start} = \{v\}$, then there are $n-f$ correct nodes at the beginning of beat r with $clock = v$; when they execute Line 1, they all send the same value v. Since $v \in \{0,1\}$, each correct node receives at least $n-f$ messages with the same value v (see Observation 3.1), therefore $maj = v$ and $\#maj \geq n-f$. Thus, in Line 5, all correct nodes set $clock := 1 - maj = 1 - v$. □

DEFINITION 3.4. *A beat r is called "safe" if all correct nodes have the same value of rand during r.*

LEMMA 3. *Let T be a coherent interval. If $r \in T$ is a safe beat, then $clocks_r^{end} \subset \{v, \perp\}$ for $v \in \{0,1\}$.*

PROOF. Correct nodes set *clock* either in Line 5 or in Line 6. Those that set *clock* in Line 6 set it to "$\perp$". Consider all correct nodes that set *clock* at Line 5; we show that they all set *clock* to the same value v. r is a safe beat, therefore, all correct nodes that sent "$\perp$" in Line 1 will be considered to have sent the same value by all correct nodes. Thus, two correct nodes can differ by at most f values when setting maj and $\#maj$ in Line 4. By Observation 3.1, all nodes that have $\#maj \geq n-f$ have the same value for maj. Thus, all nodes that update *clock* in Line 5 update it to the same value. □

LEMMA 4. *Let T be a coherent interval. If $r \in T$ is a safe beat in which $clocks_r^{start} \subset \{v, \perp\}$ for $v \in \{0,1\}$, then with probability at least $min\{p_0, p_1\}$, $clocks_r^{end} = \{v'\}$ for $v' \in \{0,1\}$.*

PROOF. If $clocks_r^{start} = \{v\}$ for $v \in \{0,1\}$, then, by Lemma 2 we are done. Otherwise assume that $clocks_r^{start} \neq \{v\}$ for $v \in \{0,1\}$.

r is a safe beat, therefore, all correct nodes have the same value of *rand*. Consider two cases: $clocks_r^{start} = \{\perp\}$ and $clocks_r^{start} \neq \{\perp\}$. In the first case, all nodes consider (in Line 3) to have received at least $n-f$ messages with value *rand*; thus they set $\#maj \geq n-f$ and $maj = rand$, and therefore, (after Line 5) $clocks_r^{end} = \{1 - rand\}$.

In the second case, $clocks_r^{start} \neq \{\perp\}$; thus, under the lemma's assumption we have that $clocks_r^{start} = \{v, \perp\}$ for $v \in \{0,1\}$. Recall that $clocks_{r-1}^{end} = clocks_r^{start}$, thus, the values of $clocks_r^{start}$ have been determined in beat $r-1$. The "unpredictability" property implies that *rand* is independent of "what happened" during beat $r-1$ (see Remark 3.1). We thus conclude that with probability at least $min\{p_0, p_1\}$, $rand = v$. In this case, all correct nodes have $\#maj \geq n-f$ and $maj = v$, thus all correct nodes have (after Line 5) $clocks_r^{end} = \{1-v\}$.

Thus, with probability of at least $min\{p_0, p_1\}$ we have that $clocks_r^{end} = \{v'\}$ for $v' \in \{0,1\}$. □

LEMMA 5. *Let T be a coherent interval. Any beat $r \in T$ is safe with probability $p_0 + p_1$.*

PROOF. Consider some beat $r \in T$; since T is coherent, there is a set of $n-f$ correct nodes during beat r. Since $\Delta_{node} \geq \Delta_{\mathcal{C}}$, they have all executed $\mathcal{C}$ for $\Delta_{\mathcal{C}}$ beats ($\mathcal{C}$'s required convergence time). Thus, properties "event E_0" and "event E_1" hold; which means that with probability p_0 all correct nodes have $rand = 0$ and with probability p_1 all correct nodes have $rand = 1$. Thus, with probability $p_0 + p_1$ the beat is safe. □

THEOREM 2. *SS-BYZ-2-CLOCK solves the 2-Clock problem with expected constant convergence time.*

REMARK 3.2. *When talking about the expected convergence time of SS-BYZ-2-CLOCK, it is convenient to think of an infinitely long coherent interval $T = [r, \infty]$. However, the above theorem holds also for short finite intervals, but would require saying: "T is of length of at least l, where at any beat after $r + l$ there is a constant probability that the algorithm converges". Instead, we simply say that T is "long enough".*

PROOF. Let $T = [r_1, r_2]$ be a "long enough" coherent interval. By Lemma 5, for each beat $r \in T$ there is a constant probability $c_1 := p_0 + p_1$ that r is a safe beat. By the "unpredictability" property, the probability of the event E that two consecutive beats $r, r+1$ are safe is at least c_1^2. From Lemma 3 and Lemma 4, given that E occurred, there is a probability of $c_2 = \min\{p_0, p_1\}$, and thus all correct nodes have the same *clock* value at the end of beat $r+1$, and by Lemma 2, they continue to agree on it at the end of any beat $r' \geq r+1, r' \in T$.

Thus, during each beat r, $r \in [r_1 + 1, r_2]$, there is a constant probability of $c_2 \cdot c_1^2$ that r is safe, and that $r-1$ is safe, and that all correct nodes have the same *clock* value by the end of r. Therefore, after an expected constant number of beats (starting from $r_1 + 1$), all correct nodes agree on the *clock* value, and by Lemma 2, they all continue to agree on the *clock* value and change it from "1" to "0" and vice versa each beat. Hence, SS-BYZ-2-CLOCK solves the 2-Clock problem and has expected constant convergence time. □

Theorem 2 states that SS-BYZ-2-CLOCK converges with expected constant time. However, as can be seen by the proof of Theorem 2, the result is actually much stronger: if at some beat the algorithm has not yet converged, then it has a constant probability of converging in the next beat. Thus, denote by $\Delta_{\text{SS-BYZ-2-CLOCK}}$ the expected convergence time of SS-BYZ-2-CLOCK; the probability that SS-BYZ-2-CLOCK does not converge within $l \cdot \Delta_{\text{SS-BYZ-2-CLOCK}}$ beats decreases exponentially with l. Therefore, not only does SS-BYZ-2-CLOCK converge in expected constant time, it also does so with high probability.

Algorithm SS-BYZ-4-CLOCK /* executed at node u each beat */
/* $\mathcal{A}_1, \mathcal{A}_2$ are instances of SS-BYZ-2-CLOCK */

On beat (signal from global beat system):

1. **execute** a single beat of $\mathcal{A}_1$;
2. **if** $u.clock(\mathcal{A}_1) = 0$ then
   execute a single beat of $\mathcal{A}_2$;
3. **set** $u.clock := 2 \cdot u.clock(\mathcal{A}_2) + u.clock(\mathcal{A}_1)$[a];

[a]To differentiate between the output *clock* value of SS-BYZ-4-CLOCK and that of SS-BYZ-2-CLOCK, consider $u.clock$ to be the output of SS-BYZ-4-CLOCK, $u.clock(\mathcal{A}_1)$ is the output of $\mathcal{A}_1$ and $u.clock(\mathcal{A}_2)$ is the output of $\mathcal{A}_2$.

Figure 3: An algorithm that solves the 4-Clock problem.

4. SOLVING THE 4-CLOCK PROBLEM

The previous section solved the 2-Clock problem. The following describes how to solve the 4-Clock problem, using 2 instances of SS-BYZ-2-CLOCK, $\mathcal{A}_1$,and $\mathcal{A}_2$. The presented solution requires that $\Delta_{node} \geq \max\{\Delta_{\mathcal{A}_1}, 2 \cdot \Delta_{\mathcal{A}_2}\}$; since $\mathcal{A}_1, \mathcal{A}_2$ are both instances of SS-BYZ-2-CLOCK, Δ_{node} is set to be $2 \cdot \Delta_{\text{SS-BYZ-2-CLOCK}}$.

THEOREM 3. SS-BYZ-4-CLOCK *(Fig. Figure 3) solves the 4-Clock problem with expected constant convergence time.*

PROOF. Let $T = [r_1, r_2]$ be a "long enough"[6] coherent interval. A single beat of $\mathcal{A}_1$ is executed for every beat $r \in T$; thus $\mathcal{A}_1$ is executed properly by all correct nodes during T, and all lemmata regarding $\mathcal{A}_1$ hold. Therefore, $\mathcal{A}_1$ converges with expected constant time.

Let $r_{\mathcal{A}_1} \in T$ be the beat at which $\mathcal{A}_1$ has converged. Thus, for any beat $r \geq r_{\mathcal{A}_1}, r \in T$ all correct nodes alternate between executing a single beat of $\mathcal{A}_2$ and not executing $\mathcal{A}_2$. Thus, a single beat of $\mathcal{A}_2$ is executed every other beat in $[r_{\mathcal{A}_1}, r_2]$ by all correct nodes. Therefore, due to Theorem 2, $\mathcal{A}_2$ converges in expected constant time.

Let $r_{\mathcal{A}_2}$, $r_{\mathcal{A}_2} \geq r_{\mathcal{A}_1}$ denote the beat at which $\mathcal{A}_2$ has converged. During the interval $[r_{\mathcal{A}_2}, r_2]$, $\mathcal{A}_1$ and $\mathcal{A}_2$ have the following pattern: $(clock(\mathcal{A}_1) = 0, clock(\mathcal{A}_2) = 0), (clock(\mathcal{A}_1) = 1, clock(\mathcal{A}_2) = 0), (clock(\mathcal{A}_1) = 0, clock(\mathcal{A}_2) = 1), (clock(\mathcal{A}_1) = 1, clock(\mathcal{A}_2) = 1)$; this pattern continues until beat r_2. Thus, during the interval $[r_{\mathcal{A}_2}, r_2]$ the *clock* variable (that is updated in Line 3) has the following pattern: 0, 1, 2, 3; and this pattern continues until beat r_2.

To conclude, the linearity of expectations implies that $r_{\mathcal{A}_2} - r_1$ is constant in expectation. That is, SS-BYZ-4-CLOCK converges in expected constant time. □

REMARK 4.1. SS-BYZ-4-CLOCK *uses two different pipelines of coin-flipping. Actually, it could be improved to use a single coin-flipping pipeline, and reduce both the message complexity and expected convergence time. However, these improvements are only by a constant factor, and therefore are omitted for the sake of clarity.*

5. SOLVING THE K-CLOCK PROBLEM (FOR ANY K)

The construction of SS-BYZ-4-CLOCK can be similarly used to solve the 8-Clock problem from $\mathcal{A}_1$ that solves the 4-Clock problem and $\mathcal{A}_2$ that solves that 2-Clock problem.

[6]See Remark 3.2.

In general, any 2^{k+1}-Clock problem can be solved with $\mathcal{A}_1$ that solves 2^k-Clock and $\mathcal{A}_2$ that solves the 2-Clock problem. Even better, any $2^{2^{k+1}}$-Clock problem can be solved with $\mathcal{A}_1, \mathcal{A}_2$ that solve the 2^{2^k}-Clock problem. Thus, the k-Clock problem can be solved for infinitely many values. However, such constructions have $\log n$ overhead (or $\log \log n$, depending on which of the above schemas is used) in their message complexity and at least the same overhead in their expected convergence time.

The following SS-BYZ-CLOCK-SYNC (see Figure 4) algorithm has a constant overhead both in message complexity and in its expected convergence time. It uses a 4-Clock algorithm to construct a k-Clock algorithm for any value of k. The schema used is similar to the algorithm of Turpin and Coan (see [18]) when combined with the algorithm of Rabin (see [17]). More specifically, SS-BYZ-CLOCK-SYNC is constructed of 4 "phases", each executed in a consecutive beat. The first phase sends the clock value to everyone. In the second phase, each node votes on the majority clock value it received, or $\perp$ if no such value exists. The third phase determines whether enough nodes voted on a value $\neq\perp$, thus ensuring that those nodes that have voted, voted on the same value in phase 2. Lastly, in the fourth phase the new clock value is set either to be the majority clock value of phase 2, or (if there were not enough votes) is randomly selected using the output of the coin-flipping algorithm.

In the context of SS-BYZ-CLOCK-SYNC, Δ_{node} is the same as in SS-BYZ-4-CLOCK.

$\mathcal{A}$ is an instance of SS-BYZ-4-CLOCK. The following discussion assumes that all correct nodes have the same value of $clock(\mathcal{A})$; it takes expected constant time until this happens. By this assumption, all correct nodes perform the same portion of the code on each beat. That is, they all perform Block 3.a on some beat, then on the next beat they perform Block 3.b, and so on. In the following lemmata, we assume that correct nodes operate in the following cycle: they all perform Block 3.a, the following beat they perform Block 3.b, then Block 3.c, then Block 3.d, and then they go back to performing Block 3.a.

DEFINITION 5.1. *Let T be a coherent interval; For any beat $r \in T$:*
$full\_clocks_r^{start}$ is the set of all full_clock values of correct nodes at the beginning of beat r.
$full\_clocks_r^{end}$ is the set of all full_clock values of correct nodes at the end of beat r.

Notice that $full\_clocks_r^{start}, full\_clocks_r^{end} \subset \{0, 1, \ldots, k-1\}$ for all r.

Algorithm SS-BYZ-CLOCK-SYNC /* executed at node u each beat */
/* $\mathcal{A}$ solves the 4-Clock problem */

On beat (signal from global beat system):

1. **execute** a single beat of $\mathcal{A}$;
2. **set** $u.full\_clock := u.full\_clock + 1 \mod k$;
3. **case** $u.clock(\mathcal{A})$ of: /* consider $u.clock(\mathcal{A})$ at the beginning of the beat */

(a) $u.clock(\mathcal{A}) = 0$: **broadcast**[a] $u.full\_clock$ to all;

(b) $u.clock(\mathcal{A}) = 1$: **if** received $n - f$ same value v in previous beat
set $propose := v$;
else
set $propose := \perp$;
fi
broadcast $propose$ to all;

(c) $u.clock(\mathcal{A}) = 2$: **let** $save$ be the majority value $\neq\perp$ of the received values;
(if all values are $\perp$, set $save := \perp$)
if $save \neq \perp$ appeared $n - f$ times
set $bit := 1$;
else
set $bit := 0$;
fi
broadcast bit to all;
if $save = \perp$
set $save := 0$;

(d) u.$clock(\mathcal{A}) = 3$: **if** received $n - f$ "1" in previous beat
set $u.full\_clock := save + 3$;
else if received $n - f$ "0" in previous beat
set $u.full\_clock := 0$;
else if $rand = 1$
set $u.full\_clock := save + 3$;
else if $rand = 0$
set $u.full\_clock := 0$;
fi

[a] In the context of this paper, "broadcast" means "send the message to all nodes".

Figure 4: An algorithm that solves the k-Clock problem for any k.

LEMMA 6. *Let T be a coherent interval and let $r \in T$ be a beat at which $clock(\mathcal{A}) = 3$; if $full\_clocks_r^{end} = \{v\}$ then for every beat $r' > r, r' \in T$ it holds that*

$$full\_clocks_{r'}^{start} = \{v + (r' - r - 1) \mod k\} .$$

PROOF. Assume the lemma holds for any beat r', $r \leq r' \leq r + 5$. Recall that $full\_clocks_{r+4}^{end} = full\_clocks_{r+5}^{start}$. The assumption on r' implies that $full\_clocks_{r+4}^{end} = \{v + 4 \mod k\}$. In addition, notice that at beat $r+4$, it holds that $clock(\mathcal{A}) = 3$. Now, repeatedly applying the above assumption leads to $full\_clocks_{r+i}^{end} = \{v+i \mod k\}$ (for $i \geq 0, r+i \in T$). In other words, $full\_clocks_{r'}^{end} = \{v + (r' - r) \mod k\}$ and $full\_clocks_{r'}^{start} = \{v + (r' - r - 1) \mod k\}$. It is left to prove that the lemma holds for any beat r', $r \leq r' \leq r + 5$.

For $r' = r + 1$ the claim holds immediately; additionally, up until beat $r+5$, the correct nodes update $full\_clock$ only in Line 2. Therefore, they all update $full\_clock$ in the same way. Thus, $full\_clocks_{r+1}^{start} = \{v\}$; $full\_clocks_{r+2}^{start} = \{v + 1\}$; $full\_clocks_{r+3}^{start} = \{v + 2\}$; $full\_clocks_{r+4}^{start} = \{v + 3\}$, (where "+" is modulo k).

It remains to show that $full\_clocks_{r+5}^{start} = \{v + 4\}$. We will do this by proving an equivalent claim, namely, that: $full\_clocks_{r+4}^{end} = \{v + 4\}$. To show this, consider the messages sent during beats $r + 1, \ldots, r + 4$.

At beat $r+1$ all correct nodes send $v+1$ to everyone, and so all correct nodes receive at least $n - f$ copies of $v + 1$. At beat $r + 2$ all correct nodes set $propose := v + 1$. At beat $r + 3$ all correct nodes have $save = v + 1$ and they set $bit := 1$ and therefore all of them receive $n - f$ copies of "1". This implies that at beat $r + 4$ all correct nodes set $full\_clock := save + 3 = v + 1 + 3 = v + 4$. That is, $full\_clocks_{r+4}^{end} = \{v + 4\}$. □

LEMMA 7. *Let T be a coherent interval; At most one value $v \neq \perp$ can be sent in Block 3.b by correct nodes at some beat $r \in T$.*

PROOF. Immediate from Observation 3.1. □

LEMMA 8. *Let T be a coherent interval; If $r \in T$ is a safe beat at which $clock(\mathcal{A}) = 3$, then with probability at least $\min\{p_0, p_1\}$ all correct nodes have the same full_clock value.*

PROOF. First, consider the case in which no correct node sees $n - f$ copies of the same value. In this case, all correct nodes set $full\_clock := 0$, with probability p_0.

If some correct node p receives $n-f$ copies of some value v, then all correct nodes that receive $n-f$ copies of some value, receive the same value v (by Observation 3.1). Notice that v is calculated according to messages determined at beat $r - 1$, and $rand$ was chosen at beat r. Therefore, due to

"*unpredictability*", *rand* and v are independent of each other. Thus, with probability at least $\min\{p_0, p_1\}$, all correct nodes update *full_clock* in the same manner: either *full_clock* $:= 0$ or *full_clock* $:=$ *save* $+3$. If $v = 0$ then we are done. If $v = 1$ then we are left to show that all correct nodes have the same value of *save*.

Since p has received $n - f$ copies of "1" it means that some correct node q has sent "1" in beat $r - 1$. Thus, q has received at least $n - f$ copies of $save_q \neq \perp$ at beat $r - 2$. Thus, all other correct nodes have received at least $n - 2f$ copies of $save_q$. By Lemma 7, correct nodes either sent $\perp$ or $save_q$ in beat $r - 2$. Thus, correct nodes can receive at most $f < n - 2f$ values that are not $\perp$ and are not $save_q$. Hence, all correct nodes have $save = save_q$. □

THEOREM 4. ss-Byz-Clock-Sync *solves the k-Clock problem for any value of k, and converges in expected constant time.*

PROOF. The proof is very similar to the proof of Theorem 2 and Theorem 3. By Lemma 8, after an expected constant number of beats all correct nodes have the same *full_clock* value. By Lemma 6, the correct nodes continue to agree on their *full_clock* value and increase it by "1" at each beat (modulo k).

Therefore, ss-Byz-Clock-Sync solves the k-Clock problem for any value of k; and converges in expected constant time. □

6. DISCUSSION

6.1 Self-stabilizing Coin-flipping

The main result in this paper is the expected constant time digital clock synchronization algorithm. However, to reach this result an important tool has been developed: the self-stabilizing probabilistic coin-flipping algorithm, which provides a stream of common random bits to all correct nodes. This tool can be useful in developing randomized self-stabilizing solutions to various problems, since it provides a self stabilizing access to a stream of shared coins.

For example, the algorithm in [9] could be adapted to use ss-Byz-Coin-Flip as a self-stabilizing coin-flipping building block. Such a change would lead to an exponential reduction in the convergence time of [9]. However, [9]'s convergence time is dependent upon the wraparound clock value, and therefore would still not be constant.

Notice that the random bit produced at beat r is "independent" of events occurring up to beat $r - 1$. However, the adversary can "see" the result of the coin at beat r and take it into consideration when sending messages at beat r. Thus, one must be careful when using the stream of random bits; specifically, one must ensure that the states to choose from (using the random bit) have been decided in the previous beat, and not in the current beat. The technique presented in this paper can be adapted in dealing with such situations.

6.2 Self-stabilizing Pipelining

To the best of our knowledge, pipelining as means of transforming non-self-stabilizing *Byzantine* tolerant algorithms into self-stabilizing *Byzantine* tolerant algorithms, was first suggested in [15]. The current work is another example of employing the "pipeline concept" in a self-stabilizing and *Byzantine* tolerant protocol. It is interesting to classify the class of problems that can be solved using this technique.

6.3 Future Research

We consider two main points for future research; the first, regards the bounded-delay model, which assumes there is a bound on messages' delivery time (replacing the global-beat-system assumption). Previously, clock synchronization in the bounded-delay model was solved using an underlaying pulsing algorithm with linear convergence time. Can the ideas in the current paper be transported to the bounded-delay model, and reduce the convergence time to expected constant? This can be done either by directly solving the clock synchronization problem, or by reducing the convergence time of the underlying pulsing algorithm. If so, what extra overhead will be required? Notice that automatic translators from the global-beat-system model to the bounded-delay model exist, but they require linear running time. Therefore, they cannot efficiently transport the current ideas into the bounded-delay model.

The second point regards asynchronous systems. Without probability, it is impossible to solve the clock synchronization problem in an asynchronous network with *Byzantine* nodes. However, once probability is introduced, such a solution might be feasible. It is interesting to see what form the clock synchronization problem takes in an asynchronous setting, and what kind of probabilistic solutions apply.

7. REFERENCES

[1] A. Arora, S. Dolev, and M.G. Gouda. Maintaining digital clocks in step. *Parallel Processing Letters*, 1:11–18, 1991.

[2] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30, New York, NY, USA, 1983. ACM.

[3] G. Bracha. An O(log n) expected rounds randomized byzantine generals protocol. *J. ACM*, 34(4):910–920, 1987.

[4] R. Canetti and T. Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 42–51, New York, NY, USA, 1993. ACM.

[5] A. Daliot, D. Dolev, and H. Parnas. Linear time byzantine self-stabilizing clock synchronization. In *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.

[6] D. Dolev and E. N. Hoch. Byzantine self-stabilizing pulse in a bounded-delay model. In *Proc. of 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'07)*, Paris, France, Nov 2007.

[7] D. Dolev and E. N. Hoch. On self-stabilizing synchronous actions despite byzantine attacks. In *Proc. the 21st Int. Symposium on Distributed Computing (DISC'07)*, Lemesos, Cyprus, Sep. 2007.

[8] S. Dolev. Possible and impossible self-stabilizing digital clock synchronization in general graphs. *Journal of Real-Time Systems*, 12(1):95–107, 1997.

[9] S. Dolev and J. L. Welch. Self-stabilizing clock

synchronization in the presence of byzantine faults. *Journal of the ACM*, 51(5):780–799, 2004.

[10] S. Dolev and J. L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults (abstract). In *PODC*, page 256, 1995.

[11] C. Dwork, D. Shmoys, and L. Stockmeyer. Flipping persuasively in constant time. *SIAM Journal on Computing*, 19(3):472–499, 1990.

[12] P. Feldman and S. Micali. An optimal probabilistic algorithm for synchronous byzantine agreement. In *ICALP '89: Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, pages 341–378, London, UK, 1989. Springer-Verlag.

[13] M.J. Fischer and N.A. Lynch. A Lower Bound for the Time to Assure Interactive Consistency. *Information Processing Letters*, 14:183–186, 1982.

[14] T. Herman. Phase clocks for transient fault repair. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1048–1057, 2000.

[15] E. N. Hoch, D. Dolev, and A. Daliot. Self-stabilizing byzantine digital clock synchronization. In *Proc. of 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'06)*, Dallas, Texas, Nov 2006.

[16] M. Papatriantafilou and P. Tsigas. On self-stabilizing wait-free clock synchronization. *Parallel Processing Letters*, 7(3):321–328, 1997.

[17] M. Rabin. Randomized Byzantine generals. *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 403–409, 1983.

[18] R. Turpin and B. Coan. Extending binary Byzantine agreement to multivalued Byzantine agreement. *INFO. PROC. LETT.*, 18(2):73–76, 1984.

Chapter 4

OCD: Obsessive Consensus Disorder (or Repetitive Consensus)

Danny Dolev and Ezra N. Hoch, Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing (PODC '08), Toronto, Canada, Pages: 395-404, Aug. 2008.

OCD: Obsessive Consensus Disorder (or Repetitive Consensus)

Danny Dolev*
Hebrew University
dolev@cs.huji.ac.il

Ezra N. Hoch
Hebrew University
ezraho@cs.huji.ac.il

ABSTRACT

Consider a distributed system S of sensors, where the goal is to continuously output an agreed reading. The input readings of non-faulty sensors may change over time; and some of the sensors may be faulty (*Byzantine*). Thus, the system is required to repeatedly perform consensus on the input values.

This paper investigates the following question: assuming the input values of all the non-faulty sensors remain unchanged for a long period of time, what can be said about the agreed-upon output reading of the entire system? We prove that no system's output is stable, i.e. the faulty sensors can force a change of the output value at least once.

We show that any system with binary input values can avoid changing its output more than once, thus matching the lower bound. For systems with multi-value inputs, we show that the output may change at most twice; when $n = 3f + 1$ this solution is shown to be tight. Moreover, the solutions we present are self-stabilizing.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Algorithms, Reliability, Theory

Keywords

Distributed computing, fault tolerance, self-stabilization, Byzantine failures, repetitive consensus, long-lived consensus.

*This work was funded in part by ISF.

PODC'08, August 18–21, 2008, Toronto, Ontario, Canada.

1. INTRODUCTION

Consensus is one of the fundamental problems in distributed algorithms, and a vast body of literature exists on the subject (see [8], [10] and [1]). The "common" consensus problem consists of nodes with static input values that are required to agree on some output value within a finite time. Different extensions of the consensus problem study different directions; in this paper we consider the case where a sequence of multiple consensuses takes place.

Executing multiple consensus comes in different flavors; *Continuous Consensus* aims at continuously agreeing on the history of inputs of all nodes in the system (see [12]). *Multi-consensus* efficiently executes multiple consensuses sequentially (see [2]), *Committee Decision* executes multiple consensuses concurrently (see [9]); and others, such as [13]. We are interested in a different aspect of executing multiple consensuses.

When executing m consensuses, there are m agreed-upon output values. This raises the following question: What is the connection between the different outputs of the different consensuses? Is there any way to guarantee that consecutive consensuses output the same value - i.e. that the output of the different consensuses is stable? None of the above papers has investigated the stability of the output of the different consensuses in the sequence. The first paper to discuss the stability of "long-lived" consensus is [7] (from which the term "long-lived" is taken). [7] (and later, [4]) also defines measurements to estimate solutions of the "long-lived" consensus problem; in the current work, we consider similar measurements (see Section 2).

Consider a system consisting of n nodes, where each node receives an input value from the set V, $V = \{0, \ldots, v-1\}$; these input values can change over time, and hence the nodes need to repeatedly agree on their collective output value. For example, consider a network of sensors that needs to agree on a single output: assume some of the nodes are malfunctioning (*Byzantine*), and consider the following question: if the input of the nodes changes overtime and eventually stabilizes, is it possible to ensure that the output will also stabilize eventually? As will be shown below, the answer is "No". However: if the input does not change, then (under certain restrictions) the output can change at most once. For example, assume a system with $V = \{0, 1\}$ in which at some time the input values of all non-faulty nodes have stopped changing. If no faulty nodes exist, the output will eventually stop changing. However, we show that even a single *Byzantine* node can delay this transition indefinitely.

The above impossibility result is true for any system with

Byzantine presence, regardless of the communication model (synchronous, asynchronous, etc). However, our solution to reducing output changes operates in a highly-fault tolerant manner: it is self stabilizing[1] and *Byzantine* tolerant. That is, starting from any initial state, in the presence of permanent f *Byzantine* nodes, the number of output changes is bounded.

Related work: Overcoming faults in sensors by averaging the input values has been done in [11]; however the averaging is done by one node connected to all sensors, and has low fault resiliency.

Quantifying faulty nodes' effect on the system's output was investigated in [7] and [4]. These papers consider a *geodesic path*, which is a list of input vectors (representing the inputs to each node) in which each node changes its input value at most once. The two papers concentrate on the stability assurances that can be guaranteed for geodesic paths. However, they do not consider the *Byzantine* case, in which a small set of faulty nodes repeatedly face changes in their input values.

Recently, [15] has extended [7]'s worst-case scenario results to the average case, as well as showing the importance of memory for the output's stability. [15] considers paths that are randomly selected, and thus examines the average case scenario. However, [15] does not incorporate *Byzantine* behavior, which coordinates its failures in a malicious (and non-random) fashion.

[4] and [7] show that for memory-less consensus functions, even with a single faulty node, the output may change infinitely many times. However, the behavior of *Byzantine*-tolerant consensus functions with memory has not been investigated yet. In addition, the results of [7], [4] and [15] have not been shown to be self-stabilizing.

Contribution: We define an *oblivious path*, which is a list of input vectors in which at most f nodes change their input values (arbitrarily many times). We then quantify a system output's stability for such paths, giving both lower and upper bounds. The solution is shown to be self-stabilizing.

2. COMPUTATIONAL MODEL

Following [4] and [7], we begin this paper with an analysis that ignores the communication and timing model of the system. Let $V, V = \{0, \ldots, v-1\}$ be the set of allowed input values, and let $\mathcal{F} : V^n \to V \cup \{\perp\}$ be a function on an input vector $\vec{x} \in V^n$. Given a list of input vectors $\vec{x_0}, \vec{x_1}, \ldots, \vec{x_l}$, we are interested in the behavior of $\mathcal{F}(\vec{x_0}), \mathcal{F}(\vec{x_1}), \ldots, \mathcal{F}(\vec{x_l})$. To allow $\mathcal{F}$ to have memory (adding states to the system) we consider $\mathcal{F} : V^n \times M \to V \cup \{\perp\}$, and a state transition function $\tau : V^n \times M \to M$.

Intuitively, an input vector $\vec{x} \in V^n$ represents the inputs of each node in the distributed system, while the state transition function τ represents the change in the state of the distributed system (sending/reciving messages and their effects on each node's internal memory). Lastly, $\mathcal{F}$ represents the output of all non-faulty nodes in the system. The formal definitions are as follows (based on those of [4]):

DEFINITION 2.1. *A* **system** *is defined by a 6-tuple* $S = \langle n, v, f, M, \tau, \mathcal{F} \rangle$ *, where* n *is the number of nodes,* v *is the size of the input space* $V = \{0, \ldots, v-1\}$*,* f *is a bound on the number of faulty nodes,* M *is a set of memory states,* τ *is a state transition function* $\tau : V^n \times M \to M$ *and* $\mathcal{F}$ *is the output function* $\mathcal{F} : V^n \times M \to V \cup \{\perp\}$*.*

For the above definition to be "interesting" a "*validity*" requirement on $\mathcal{F}$ is added: the output of $\mathcal{F}$ must be the input of some correct node. Since $\mathcal{F}$ does not "know" which nodes are faulty and which are not, it can output a value ν only if it has seen at least $f+1$ copies of ν in the input vector; otherwise it must output $\perp$.

We also require that if some value ν' appears $n-f$ (or more) times in the input vector, then the output of $\mathcal{F}$ is ν'. More formally, let $\#\nu(\vec{x})$ be the number of occurrences of ν in $\vec{x}$: if $\mathcal{F}(\vec{x}) = \nu \in V$ then $\#\nu(\vec{x}) \geq f+1$; if $\mathcal{F}(\vec{x}) = \perp$ then $\forall_{\nu \in V} \#\nu(\vec{x}) < n-f$.

The above *validity* is considered throughout the paper. In Section 6 a different *validity* measure is presented.

DEFINITION 2.2. *An* **input path** *(path for short)* P *(of some system* S*) is a list of input vectors* $\vec{x}_0, \vec{x}_1, \ldots, \vec{x}_{l-1} \in V^n$*. Component* k *is said to* **change** *on path* P *if there is some input vector* $\vec{x}_i$ *such that* $\vec{x}_i[k] \neq \vec{x}_{i+1}[k]$*, where* $\vec{x}[k]$ *is the* k*th entry in the vector* $\vec{x}$*.*

DEFINITION 2.3. *A* **run** *of a system* S *from memory state* m_0 *on some path* P *is the sequence* $m_1 = \tau(\vec{x}_0, m_0), m_2 = \tau(\vec{x}_1, m_1), \ldots, m_l = \tau(\vec{x}_{l-1}, m_{l-1})$*;*

The **decision output** *of the above run is the sequence* $\mathcal{F}(\vec{x}_0, m_0), \mathcal{F}(\vec{x}_1, m_1), \ldots, \mathcal{F}(\vec{x}_{l-1}, m_{l-1})$*. Denote by* $O_i := \mathcal{F}(\vec{x}_i, m_i)$*. The decision output changes at index* $i > 0$ *if* $O_{i-1} \neq O_i$*.*

[4] and [7] define a **geodesic input path** as an input path in which each component changes at most once. In addition, they define the **instability** of a system S as the maximal number of decision output changes for any geodesic input path, starting from S's initial state. In the current work we are interested in a somewhat different stability measurement, so we henceforth refer to the instability of [4] and [7] as **geodesic-instability**, in order to differentiate it from our usage of the term "instability".

Note that *Byzantine* nodes may "pretend" that their inputs are consistently changing. It is therefore important to design systems that are robust in the face of such behavior, *i.e.* , that do not change their output if the correct nodes' inputs do not change. In other words, the target should be a system that is oblivious to changes in *Byzantine* nodes' input values (as long as the other nodes have stable input values). Hence, we define the following:

DEFINITION 2.4. *An* **f-oblivious input path** *(oblivious path, for short) of a system* S *is an input path* P *in which at most* f *components change.*

Notice that a geodesic path's length is bounded by n, since each of its components can change at most once. However, an oblivious path can be arbitrarily long, since f components can repeatedly change. Faulty nodes can behave arbitrarily at any point during the execution of S. Moreover, the input values of non-faulty nodes may change due to external readings. Thus, we aim at expressing the robustness with respect to all memory states and to all oblivious paths.

DEFINITION 2.5. *Let* $CT(S, m, P)$ *be the (possibly infinite) number of times the decision output of* S *changes when*

[1]For more on self-stabilizing see [14].

running on oblivious path P from memory state m. Let $MaxCT(S,m) := max_P\{CT(S,m,P)\}$.

Similarly, $CL(S,m,P)$ is the (possibly infinite) largest index for which the decision output of S changes when running on oblivious path P from memory state m.
Let $MaxCL(S,m) := max_P\{CL(S,m,P)\}$.

DEFINITION 2.6. *The* **count-instability** *of a system S is the maximal number of decision output changes during a run from any memory state m and for any oblivious input path P. Formally, we can define count-instability as: $max_m\{MaxCT(S,m)\}$.*

The **length-instability** *of a system S is the smallest index after which no change occurs to the decision output during a run from any memory state m and for any oblivious input path P. Formally, length-instability can be expressed as: $max_m\{MaxCL(S,m)\}$.*

It has been shown (in [4], [7]) that in memory-less systems S (systems in which $M = \{m_0\}$) in which some component may change its input infinitely many times, a path can be constructed to cause any number of decision output changes. In other words, for memory-less S with $f \geq 1$, the count-instability of S is ∞. This is the reason that [4] and [7] discuss only geodesic-instability, but do not discuss count-instability or length-instability at all. An interesting extension to the above oblivious-path definition (and therefore to the count/length instability definitions) that is not discussed in the current paper is one which considers changes to input values of non-faulty nodes. Notice that the lower bounds of Section 3 hold for this extension as well.

Table 1 presents the lower and upper bounds appearing in the following sections.

Table 1: Summary of results

parameters	count-instability	theorem#
$f > 0$	≥ 1	Theorem 1
$n \geq 3f+1$	≤ 2	Theorem 4
$n = 3f+1, \lvert V\rvert > 2$	≥ 2	Theorem 3
$n \geq 3f+1, \lvert V\rvert = 2$	1	Theorem 5

2.1 A General System Model

The computational model presented above captures common requirements of any distributed system that tries to repeatedly reach consensus on changing input values. Therefore, any lower bound presented in this computational model holds for any distributed system.

A distributed system T may have richer semantics than the proposed computational model. However, the properties of the computational model encompass the very basic behavior of repeatedly reaching consensus; therefore, it should be possible to "embed" the computational model in T's semantics. That is, in certain scenarios (such as when restricting the behavior of T) there is a reduction from T to the computational model.

For example, suppose that T is a sparsely connected network of nodes that gathers inputs from different nodes, waits until they do not change (to avoid fluctuations in the output), propagates them throughout the system, and only then decides on an output. In any system, on a run of T, at some time t_{-1}, the input will change to $\vec{x}_{-1}$ and stay there. Eventually, T outputs some value O_{-1} at time t_0. Consider m_0 to be the state of the system at t_0. At some time $t'_0 \geq t_0$ the input changes to $\vec{x}_0$ and eventually - at time t_1 - the system outputs O_0. Mark by m_1 the state of the system at t_1. In general, at time t_i the system is at some state m_i, at time $t'_i \geq t_i$ the input changes to $\vec{x}_i$ and at time $t_{i+1} \geq t'_i$ the system outputs a value O_i; the run continues in this manner.

Consider the scenario where $t'_i = t_i$ for all i; that is, the input changes immediately after the output has been determined. T must also operate correctly in the described scenario. However, in such a setting the following reduction can be used: $m_0, m_1, \ldots$ will be the states of the system; $\vec{x}_0, \vec{x}_1, \ldots$ will be the input vectors; τ will denote the system's state change between 2 consecutive outputs; and $\mathcal{F}$ will denote the system's output, given the previous system's state and the new input. Notice that T might change its state more than once between t_i and t_{i+1}; however, we are only interested in the change of state from the state at t_i (m_i) to the state at t_{i+1} (m_{i+1}).

The above example demonstrates how a system that produces outputs every so often, in accordance to inputs that change over time, has scenarios in which it can be abstracted using the computational model. This observation is the motivation behind using the computational model and the lower bounds proven in this paper.

3. IMPOSSIBILITY RESULTS

THEOREM 1. *For any system S with $f \geq 1$ the count-instability of S is at least 1.*

PROOF. Consider the path P : $\vec{x}_0 := 0^{n-f}1^f, \vec{x}_1 := 0^{n-f-1}1^{f+1}, \ldots, \vec{x}_i := 0^{n-f-i}1^{f+i}, \ldots, \vec{x}_{n-2f} := 0^f1^{n-f}$. Let m_0 be some state of S and consider S's run on P from m_0. Denote the decision output of S's run from m_0 on P as $O_0 := \mathcal{F}(\vec{x}_0, m_0), O_1 := \mathcal{F}(\vec{x}_1, \tau(m_0, \vec{x}_0))$, etc. Denote by m_i the ith memory state of the above run; that is $m_{i+1} = \tau(\vec{x}_i, m_i)$.

By *validity*, since $\#0(\vec{x}_0) \geq n - f$ it holds that $O_0 = 0$, and for similar reasons $O_{n-2f} = 1$. Let j be the first O_j that is 1; more formally let j be $j = min\{i|\mathcal{F}(\vec{x}_i, m_i) = 1\}$ (note that $j \geq 1$). Consider the memory state m_{j-1}: when S is at state m_{j-1}, given the input vector $\vec{x}_{j-1}$, S's output is 0 and it moves to state m_j. Then, for $\vec{x}_j$ as input, S's output is 1. Thus, the path $P := \vec{x}_{j-1}, \vec{x}_j$ is a path that causes S's run starting from m_{j-1} to change once. Notice that P is an oblivious path (for $f > 0$), since $\vec{x}_{j-1}, \vec{x}_j$ differ by exactly one component.

This implies that for any system S (with $f > 0$) there exists an oblivious path P and a state m such that S's run starting from m on P changes its decision output at least once. And so,S's count-instability is ≥ 1. □

Theorem 1 states that any system that is designed to be tolerant of even a single *Byzantine* node, must have states from which the *Byzantine* nodes can change the decision output value. The following theorem shows that the *Byzantine* node(s) can delay this output change indefinitely.

THEOREM 2. *For any system S with $f \geq 1$ the length-instability of S is ∞.*

PROOF. Let $\vec{x}_i := 0^{n-f-i}1^{f+i}$ and consider the path

$$P^k := \underbrace{\vec{x}_0, \ldots, \vec{x}_0}_{k}, \ldots, \underbrace{\vec{x}_i, \ldots, \vec{x}_i}_{k}, \ldots, \underbrace{\vec{x}_{n-2f}, \ldots, \vec{x}_{n-2f}}_{k}.$$

P^k is a path consisting of $n-2f+1$ "sections", each of length k. Each such section consists of a single input vector, where the ith section consists of k repetitions of $\vec{x}_i$. P^k is $k\cdot(n-2f+1)$ vectors long. Denote by $\vec{x}^j$ the jth vector in P^k ($j \in [0, k\cdot(n-2f+1)]$); note that $\vec{x}^j = \vec{x}_{\lfloor \frac{j}{k} \rfloor}$. For a given j, let m_j be the memory state before the input vector $\vec{x}^j$ is processed, and let O_j be the decision output of $\mathcal{F}(m_j, \vec{x}^j)$.

Due to *validity*, since $\#0(\vec{x}_0) \geq n-f$ it holds that $O_0 = 0$ and $O_j = 0$ for $j < k$. For similar reasons $O_{k\cdot(n-2f)} = 1$. Therefore, $O_j \neq O_{j-1}$ for some $j \geq k$. Consider the first such j. Denote the path $P := \vec{x}^{j-k}, \vec{x}^{j-k+1}, \ldots, \vec{x}^j$; P is an oblivious path, since at most one component changes in P. When running S from memory state m_{j-k} on the path P the output starts as "0" and changes to "1" after k input vectors; that is, $CL(S, m_{j-k}, P) \geq k$.

Notice that the above proof is independent of k's value. Thus for any k, there exists a memory state m and an oblivious path P such that $CL(S, m, P) \geq k$. That is, the length-instability of S is ∞. □

Theorem 2 implies that even a single *Byzantine* node can cause a decision output change in a system "whenever it wants". More specifically, there is a state of the system such that even if the inputs of all non-*Byzantine* nodes do not change, there is no bound on when a *Byzantine* node can cause the decision output of the system to change.

THEOREM 3. *For any system S with $n = 3f+1$ and $|V| \geq 3$ the count-instability of S is at least 2.*

PROOF. Let m_0 be some initial state and let $\vec{x} := 0^{2f+1}1^f$. Due to *validity*, $\mathcal{F}(\vec{x}, m_0) = 0$. Now, consider the vector $\vec{y} := 0^{f+1}1^{f+1}2^{f-1}$ and $m_1 := \tau(\vec{x}, m_0)$. $\mathcal{F}(\vec{y}, m_1)$ can be "0", "1" or $\perp$.

If it is "1" or $\perp$, then the path $P := \vec{x}, \vec{y}, \vec{x}$ causes S to change its output twice.

Similarly, assume that $\mathcal{F}(\vec{y}, m_1) = 0$ and let $m_2 := \tau(\vec{y}, m_1)$. Let $\vec{z} := 0^f 1^{f+1} 2^f$. Now, $\mathcal{F}(\vec{z}, m_2)$ can be either "1" or $\perp$. If it is $\perp$ then the path $P := \vec{y}, \vec{z}, 0^f 1^{2f+1}$ causes the system S to change its output twice. On the other hand, if $\mathcal{F}(\vec{z}, m_2) = 1$ then the path $P := \vec{y}, \vec{z}, 0^f 1^f 2^{f+1}$ changes the output twice. Thus, there exists an f-oblivious path that causes S to change its output twice. □

REMARK 3.1. *The $|V| \geq 3$ limit (in Theorem 3) is required. Moreover, if $|V| = 2$ then it is possible to construct a system S that has count-instability of 1. In Theorem 5 we show how to achieve exactly that.*

4. REPETITIVE CONSENSUS

In the above sections a formal analysis of the computational model was given, together with lower bounds regarding the *count-instability* of any distributed system. In the rest of the paper we concentrate on a more "practical" approach; that is, the repetitive consensus problem is defined and solved in a "synchronous" network in a self-stabilizing and *Byzantine* tolerant manner.

We start by defining the Global-Beat-System (a self-stabilizing equivalent of the classical synchronous network); then we present and solve the repetitive consensus problem as applied to this model.

4.1 Model and Definitions

The "Global Beat System" model (GBS for short) consists of n nodes that can communicate via message passing, where the sender of each message can be identified. Nodes have access to a common "global beat system", which produces signals/beats at regular intervals, such that a message sent by any node p to any node q upon receiving some beat, reaches q before the following beat. All nodes receive a signal/beat at the same time, and can perform computations and/or send messages upon its receipt. The beats are spaced in such a way as to allow a correct node to send one message to each correct node (and process such messages) in the time span between two consecutive beats.

In non-self-stabilizing synchronous systems, usually there is a common counter that all correct nodes are aware of (the current round number). This is not the case in the GBS model; however, for the sake of clarity we will refer to an "external" beat/round number r, that the nodes are not aware of. In the rest of this paper, "rounds" and "beats" will be used interchangeably. These terms are not to be confused with "pulses", which refer to the output of pulsing algorithms.

While the system is "unstable" (due to transient faults), any number of nodes may behave in a *Byzantine* manner and the communication network may behave arbitrarily. However, once the system "stabilizes", there will be at most f *Byzantine* nodes, the global beat system will produce beats regularly and the communication network will deliver messages on time (before the following beat).

DEFINITION 4.1. *A node is* **non-faulty** *when it follows the given protocol*[2]*, and* **faulty** *otherwise.*

REMARK 4.1. *In the current work processing time of incoming messages is ignored. This assumption does not weaken the result, it only simplifies the exposition.*

DEFINITION 4.2. *The communication network is* **non-faulty** *when the following conditions hold:*

1. *A message by a correct node p sent upon a receipt of a beat from the global beat system, arrives (and is processed) at its destination before the following beat is issued by the global beat system;*
2. *The sender's identity and the message context of any message received are not tampered with.*
3. *A message received by p was sent by some node no more than one beat ago. That is, "phantom" messages are not delivered.*

In real-world networks, it may take some time for the communication network to overcome transient faults. Specifically, the communication networks' buffers may contain messages that were not sent by any node, and the network may eventually deliver them. We consider the communication network to be *non-faulty* only after all of these "phantom" messages have been delivered or cleared away.

According to the above definition, once the network is non-faulty, it adheres to the GBS model. Which means that

[2]Notice that a non-faulty node p may exhibit unwanted behavior, due to its arbitrary state. However, given p's state, its behavior is determined by the protocol.

messages cannot be lost and old messages cannot be stored for an arbitrarily long time.

The transition from being faulty to becoming a "valid" participant of the protocol can't be instantaneous. Therefore, a continuous period of non-faulty behavior is required before the system or a node can be considered correct.

DEFINITION 4.3. *A node is* **correct** *following Δ_{node} beats of continuous non-faulty behavior in which the communication network is non-faulty.*

DEFINITION 4.4. *The system is* **coherent** *when the communication network is non-faulty and there are $n-f$ correct nodes.*

The values of n and f are fixed constants and are considered part of the "code" of the protocols and thus non-faulty nodes cannot initialize with arbitrary values for these constants.

4.2 The Repetitive Consensus Problem

Let I_p^r and O_p^r be the input and output value of p at beat r. Input values are from some finite set V, and output values are from $V \cup \{\perp\}$. Let $\mathcal{G}^r$ be the set of non-faulty nodes at beat r; (we will use $\mathcal{G}$ when r is clear from the context.)

DEFINITION 4.5. *The inputs (of non-faulty nodes) are* **stable** *during $[r_1, r_2]$ if for every node $p \in \mathcal{G}$ the value of I_p^r does not change, for all $r \in [r_1, r_2]$. This condition can be expressed formally as: $\forall_{p\in\mathcal{G}} \forall_{r_1 \le r \le r_2} [I_p^r = I_p^{r_1}]$.*

Similarly, we say that the outputs are stable during $[r_1, r_2]$ if $\forall_{p\in\mathcal{G}} \forall_{r_1 \le r \le r_2} [O_p^r = O_p^{r_1}]$.

Let I^r denote the vector of inputs of all nodes $I^r := (I_{p_1}^r, \ldots, I_{p_n}^r)$ and let O^r denote the vector of outputs at beat r. When talking about outputs we consider only outputs of non-faulty nodes, since no requirements can be given on outputs of *Byzantine* nodes. Likewise, only non-faulty nodes' inputs are considered, as a *Byzantine* node can have any input it wishes.

DEFINITION 4.6. *The outputs are* **in agreement** *at some beat r if $\forall_{p,p'\in\mathcal{G}} [O_p^r = O_{p'}^r]$; denote by $\mathcal{V}^r$ the agreement value and let $\mathcal{V}^r := O_p^r$ for some $p \in \mathcal{G}$.*

The outputs are **decisive** *during beat interval $R = [r_1, r_2]$ if the outputs are stable during R and the outputs are in agreement during beat r_1. Denote by $\mathcal{V}^{[r_1,r_2]}$ the agreement value and let $\mathcal{V}^{[r_1,r_2]} := \mathcal{V}^{r_1}$.*

For the outputs to become stable, the inputs must not change for a "long-enough" period of time, leading to the following definition:

DEFINITION 4.7. *A beat interval $R = [r_1, r_2]$ is* **Δ-applicable** *if $r_2 - r_1 \ge \Delta$ and the inputs are stable during R. We use the notation $R_{|\Delta} := [r_1 + \Delta, r_2]$.*

Hopefully, there exists some Δ, s.t. for any Δ-applicable interval R, the outputs are decisive in $R_{|\Delta}$. That is, if the inputs do not change for long enough, then as long as they continue not to change, the outputs are in agreement and do not change. However, due to Theorem 1 and Theorem 2 this is impossible; we therefore add the following definitions:

DEFINITION 4.8. *The outputs are k-***decisive** *during beat interval $R = [r_1, r_2]$ if the outputs are in agreement during R; and R consists of k disjoint intervals $R_1, \ldots, R_k$, such that: a. $\bigcup R_i = [r_1, r_2]$; b. for each interval R_i the outputs are stable.*

DEFINITION 4.9. *A system k-***changes its mind** *in the interval $[r_1, r_2]$ if k is the minimal value such that the outputs are $(k+1)$-decisive. Alternatively, we say that the system changes its mind k times.*

Notice that "decisive" as defined in Definition 4.6 is the same as "0-changes its mind" as defined in Definition 4.9.

The system is considered to uphold (k, Δ)-repetitive consensus behavior, if (for Δ-applicable intervals) the non-faulty nodes agree on their output, "*validity*" holds, and the output does not change more than k times. The following is a formal definition:

DEFINITION 4.10. *A system upholds a* **(k, Δ)-repetitive consensus behavior** *during interval R, if for any Δ-applicable interval $R' \subset R$:*

1. **Agreement:** *The outputs are in agreement in $R'_{|\Delta}$;*
2. **Validity:** *For any beat $r \in R'_{|\Delta}$, if $\mathcal{V}^r \neq \perp$ then some non-faulty node has $\mathcal{V}^r$ as its input value (during R'); if all non-faulty nodes have the same input value ν (during R'), then $\mathcal{V}^r = \nu$;*
3. **Bounded changes:** *The system changes its mind during $R'_{|\Delta}$ at most k times.*

Note that the above definition is applicable only for R's larger than Δ. Definition 4.10 defines "desired behavior", in the context of the repetitive consensus problem. It refers to any subinterval $R' = [r_1, r_2]$ of length at least Δ, saying that if the inputs are stable throughout R' then the following holds: "*agreement*" states that all outputs in the interval $[r_1 + \Delta, r_2]$ are the same at all non-faulty nodes. "*validity*" extends the definition from Section 2 to the distributed synchronous model at hand. Lastly, "*bounded changes*" states that the output of the system (in the interval $[r_1 + \Delta, r_2]$) may change at most k times. The motivation behind such a definition is straightforward: if the inputs are stable for long enough, then the system agrees on its output, the output is related (in a reasonable way) to the input, and the output does not change more than k times.

DEFINITION 4.11. *The* **k-repetitive consensus problem** *(k-RC for short) is solved by an algorithm $\mathcal{A}$ if there exists a constant Δ such that for any interval R the system upholds a (k, Δ)-repetitive consensus behavior.*

DEFINITION 4.12. *The* **self-stabilizing k-repetitive consensus problem** *(k-SSRC for short) states that: there exist constants $\Delta, \Delta_{stabilize}$, s.t. for any interval $[r_1, r_2]$ with no transient faults, where $r_2 - r_1 \ge \Delta_{stabilize}$, the system upholds the (k, Δ)-repetitive consensus behavior in the interval $[r_1 + \Delta_{stabilize}, r_2]$. $\Delta_{stabilize}$ is called the* **convergence time**.

The distributed model presented above is a weaker model (has less assumptions) than the computational model of

Algorithm SS-REP-CONS /* executed at node p */
/* $\mathcal{P}$ is a self-stabilizing pulse with $Cycle > \Delta_{agree}$[a] */

On beat: (signal from global beat system)

1. **Execute** a single beat of $\mathcal{P}$;
2. **If** $\mathcal{P}$ invoked a PULSE at the current beat:
   /* "vals" represents the agreed input vector */
   (a) **If** *vals* contains (at least) $n - f$ same value ν then set $Output_p := \nu$;
   Else if $Output_p$ does not appear $f + 1$ times in *vals*, set $Output_p := \perp$;
   (b) **Start** executing *Byzantine* agreement on I_p; /* I_p is p's current input */
   (denoted by BYZ-AGREE$_p$)
   (c) **Start** executing *Byzantine* consensus on $Output_p$;
   (denoted by BYZ-CONS)
   /* vals[q] holds q's input value */
3. **Once** a q's BYZ-AGREE$_q$ agreement terminates with v, update $vals[q] := v$;
4. **Once** the BYZ-CONS consensus terminates with v', update $Output_p := v'$;
5. **Set** $O_p := Output_p$; /* O_p is p's current output */

[a] Δ_{agree} is the number of rounds to execute *Byzantine* agreement.

Figure 1: An algorithm that solves the 2-SSRC problem in the GBS model.

Section 2. Therefore, the lower bounds of Section 3 hold for the distributed model as well. [3]

COROLLARY 1. *The 0-RC problem cannot be solved; The 0-SSRC problem cannot be solved, for any value of* $\Delta_{stabilize}$.

PROOF. Immediate from Theorem 1, Theorem 2 and the discussion in Section 2.1. □

REMARK 4.2. *Notice that the first part of the above corollary holds for both self-stabilizing and non-self-stabilizing models. That is, the 0-RC problem cannot be solved even in a non-self-stabilizing synchronous model in the presence of even a single* Byzantine *fault.*

5. SOLVING SS-REPETITIVE CONSENSUS

It is impossible to solve the 0-SSRC problem (or even the 0-RC problem) for any value of $f > 0$; in addition, for $n = 3f+1, |V| \geq 3$ it is impossible to solve the 1-SSRC (and the 1-RC) problem. In the following section we present a solution for the 2-SSRC problem for any value of $n \geq 3f+1$ (see Figure 1).

The SS-REP-CONS algorithm (see Figure 1) has three main goals: to have all correct nodes agree on the same vector of input values *vals*, to agree on $Output_p$ and to have $Output_p$ change as little as possible. The first goal is achieved by executing *Byzantine* agreements on each node q's input value (Line 2.b) and by storing the result in $vals[q]$ (Line 3). This ensures that, all correct nodes have a vector, *vals*, of agreed input values.

The second and third goals are achieved by the update rule in Line 2.a; however, this update is dependent on the previous value of $Output_p$. Thus, all correct nodes are also required to agree on the previous value of $Output_p$; which is done by executing a *Byzantine* consensus BYZ-CONS (Line 2.c). The feedback update rule used in Line 2.a (specifically, the second line) is essential to the stabilizing nature of SS-REP-CONS. If there was a "static" update rule instead (one which ignores the previous value of $Output_p$) it would incur more output changes than required by the present value.

To facilitate the above method of operation, the BYZ-CONS instance and the different BYZ-AGREE$_p$ instances must start their execution at all correct nodes at the same round, and be executed properly by all correct nodes (BYZ-CONS, and BYZ-AGREE$_p$ are not self-stabilizing). Thus, a self-stabilizing pulsing algorithm $\mathcal{P}$ is used as a building block[4].

$\mathcal{P}$'s convergence time is $\Delta_{\mathcal{P}}$ and $\mathcal{P}$'s *Cycle* is set to be long enough to execute BYZ-CONS and BYZ-AGREE between two consecutive pulses. Once $\mathcal{P}$ stabilizes, each time it pulses the correct nodes start executing a new instance of BYZ-CONS and the different BYZ-AGREE$_p$s. These algorithms terminate before the next pulse of $\mathcal{P}$ and thus all correct nodes have an agreed view of *vals* and of $Output_p$. From this point on, all correct nodes continue to agree on *vals* and on the previous value of $Output_p$ and thus also on the new value of $Output_p$ ($Output_p$ is dependent only on the value of *vals* and on the previous value of $Output_p$).

REMARK 5.1. *In the context of* SS-REP-CONS Δ_{node} *is defined to be equal to* $\Delta_{\mathcal{P}}$.

5.1 Correctness Proof

LEMMA 1. *If the system has been coherent for* $\Delta_{\mathcal{P}}$ *beats, then for as long as the system stays coherent: all correct nodes enter Line 2 once every Cycle beats and do so in unison.*

PROOF. Follows immediately from the properties of a self-stabilizing *Byzantine* tolerant pulsing algorithm. See [5] for more information on pulsing algorithms. □

[3] For a detailed discussion on the applicability of the computational model's lower bounds, see Section 2.1.

[4] A pulsing algorithm invokes "pulses" every *Cycle* rounds at all correct nodes simultaneously (see [5]).

$\mathcal{P}$ is a self-stabilizing pulsing algorithm. Thus, after $\Delta_{\mathcal{P}}$ rounds, $\mathcal{P}$ stabilizes and starts pulsing in a regular pattern. Denote the round at which $\mathcal{P}$ has stabilized by $r_{stabilize}$. Consider r_{start} to be the first round at which $\mathcal{P}$ invokes a pulse after $r_{stabilize}$; in other words, r_{start} is the first round in which a pulse is invoked after $\mathcal{P}$ has stabilized.

LEMMA 2. *If the system is coherent for Cycle rounds after r_{start}, then for as long as the system stays coherent: all correct nodes have the same value of vals and the same value of $Output_p$.*

PROOF. Starting at round r_{start}, all correct nodes start executing BYZ-AGREE$_p$ for each node p. Since $\mathcal{P}$'s *Cycle* is long enough to allow a *Byzantine* agreement to terminate, all correct nodes terminate all the BYZ-AGREE$_p$ instances (even for a *Byzantine* node p) with agreed output values. Thus, when updating the values of the vector *vals* at Line 3, all correct nodes update it in the same manner. Since Line 3 is the only line in which *vals* is updated, we have that starting from round $r_{start} + Cycle$, all correct nodes have the same value for *vals*.

Notice that $Output_p$ is updated in two locations: Line 1.a and Line 4. For the same reason as in the above paragraph, all correct nodes update $Output_p$ in the same manner in Line 4. In addition, since all correct nodes have the same view of *vals* and $Output_p$ (starting from round $r_{start} + Cycle$), they all update $Output_p$ in the same way also in Line 1.a. And we have that starting from round $r_{start} + Cycle$, all correct nodes have the same view of *vals* and of $Output_p$. □

Lemma 2 implies that the "*agreement*" property holds from round $r_{start} + Cycle$ and onwards. This is because the output O_p of node p is determined by Line 5 - which sets $O_p := Output_p$ - and from the above lemma all correct nodes have the same value of $Output_p = O_p$.

Let r_{start} be as defined above, and let $r_{end}, r_{end} \geq r_{start} + 3 \cdot Cycle$ be any round such that the system is coherent in the interval $[r_{start}, r_{end}]$.

LEMMA 3. *"Validity" of the $(k, 2 \cdot Cycle)$-repetitive consensus behavior holds for* SS-REP-CONS *during the interval $[r_{start} + Cycle, r_{end}]$ (for any k).*

PROOF. According to the previous lemma, all correct nodes have the same view of *vals* in the interval $[r_{start}+Cycle, r_{end}]$. Let $[r_1, r_2] \subset [r_{start} + Cycle, r_{end}]$ be the round interval in which all non-faulty nodes have stable input values. Within *Cycle* rounds of r_1, a pulse will be invoked by $\mathcal{P}$, causing all non-faulty nodes to start executing BYZ-AGREE$_p$; all the BYZ-AGREE$_p$ instances terminate before the next pulse is invoked (no later then $r_1 + 2 \cdot Cycle$). Thus, during the interval $[r_1 + 2 \cdot Cycle, r_2]$ all correct nodes have the same view of *vals*, which reflects the "real" input values of each correct node. Since correct nodes enforce $Output_p$'s adherence to the validity requirement (see Line 2.a), then during the interval $[r_1 + 2 \cdot Cycle, r_2]$ the output of SS-REP-CONS conforms to *validity*. □

LEMMA 4. *The "Bounded changes" property of the $(2, 2 \cdot Cycle)$-repetitive consensus behavior holds for* SS-REP-CONS *during the interval $T = [r_{start} + Cycle, r_{end}]$.*

PROOF. Let $[r_1, r_2] \subset T$ (where $r_2 - r_1 \geq 2 \cdot Cycle$) be an interval in which the inputs are stable. Recall that all nodes see the same value of *vals*, and that *vals* reflects the input values of the correct nodes. Let C_v be the number of non-faulty nodes with input value v. Clearly, $\sum_v C_v = n - f$. Assume by contradiction that two different $C_v, C_{v'} \geq n-2f$; therefore $C_v + C_{v'} \geq 2n - 4f$; and since $3f < n$ we have that $C_v + C_{v'} > n - f$. But this contradicts the fact that $C_v + C_{v'} \leq \sum_v C_v = n - f$. Thus, $C_v \geq n - 2f$ holds for at most one C_v.

First, consider the case in which no C_v is $\geq n - 2f$. In this case, when executing Line 2.a, the value of $Output_p$ is either unchanged or changes to $\perp$. That is, it changes at most once.

Now consider the case in which some C_v is $\geq n - 2f$. In this case, when executing Line 2.a, the value of $Output_p$ is either unchanged, v, or $\perp$. In addition, notice that once $Output_p = v$ it cannot change to some other value unless a correct node has changed its value. Thus, $Output_p$ changes at most twice (to $\perp$ and then to v). □

THEOREM 4. SS-REP-CONS *solves the 2-SSRC problem, for $\Delta := 2 \cdot Cycle$ (with $\Delta_{stabilize} := \Delta_{\mathcal{P}} + 4 \cdot Cycle$).*

PROOF. From Lemma 1, Lemma 2, Lemma 3 and Lemma 4 we have that if the system has been coherent for a period of $\Delta_{\mathcal{P}} + 4 \cdot Cycle$, then the following holds:

1. $\mathcal{P}$ "stabilizes" after Δ_p rounds; denote by $r_{stabilize}$ the round at which $\mathcal{P}$ "stabilizes".
2. $\mathcal{P}$ will invoke a pulse at some round in the interval $[r_{stabilize}, r_{stabilize}+Cycle]$; let r_{start} denote this round.
3. Let r_{end} be the maximal round such that during the interval $[r_{start} + Cycle, r_{end}]$ no transient faults occur. The properties of the $(2, 2 \cdot Cycle)$-repetitive consensus behavior hold during $[r_{start} + Cycle, r_{end}]$.

From the above, SS-REP-CONS solves the 2-SSRC problem, for $\Delta := 2 \cdot Cycle$, and $\Delta_{stabilize} := \Delta_{\mathcal{P}} + 4 \cdot Cycle$. □

In the binary setting (the input value range contains 2 values) SS-REP-CONS matches the lower bound.

THEOREM 5. SS-REP-CONS *solves the 1-SSRC problem when $|V| = 2$.*

PROOF. The proof is the same as above, with the following observation: when $|V| = 2$, the output can change only once.

We can thus follow the lines of the proof of Lemma 4, with a single change: in the case where some $C_v \geq n - 2f$, executing Line 2.a can change $Output_p$ only to v: if $Output_p = v$ then $Output_p$ never changes. If $Output_p = 1 - v$, then it cannot change to $\perp$, since if $Output_p$ does not appear $f + 1$ times in *vals* then $1 - Output_p = v$ appears $n - f$ times in *vals*. Thus, $Output_p$ is changed to v.

With the above modification, Lemma 4 proves that when $|V| = 2$ there is at most a single change in output. Thus, together with the rest of the lemmata, SS-REP-CONS solves the 1-SSRC problem. □

6. RANGE VALIDITY

In the above sections, the "exact value" (EV for short) *validity* was discussed. It is interesting to consider a different *validity* requirement: the "range value" validity (RV for

short)[5], which states that the output of the function $\mathcal{F}$ is in the range of input values of correct nodes. Notice that EV may output $\perp$, while RV must always output a value $v \in V$. Formally, RV is defined as follows: if $\mathcal{F}(\vec{x}) = \nu$ then $\sum_{\nu' \leq \nu} \#\nu'(\vec{x}) \geq f+1$ and $\sum_{\nu' \geq \nu} \#\nu'(\vec{x}) \geq f+1$.

In the next subsections, we show that 0-SSRC cannot be solved for RV-*validity*, for any value of $f > 0$. In addition, we also show how to solve the 1-SSRC problem for RV-*validity*, when $n > 4f$, thus matching the lower bound. Table 2 presents the lower and upper bounds regarding RV-*validity*.

Table 2: Summary of results for RV-*validity*

parameters	count-instability	theorem#
$f > 0$	≥ 1	Theorem 6
$v \geq n, n + \lfloor \frac{n-1}{2} \rfloor < 5f$	≥ 2	Theorem 7
$n \geq 4f+1$	1	Theorem 8

We start with the impossibility results followed by upper bounds.

6.1 Impossibility Results: Lower Bounds

Both Theorem 1 and Theorem 2 hold for RV-*validity*, as Theorem 6 states. Theorem 7 extends the bounds for RV-*validity* only.

THEOREM 6. *For any RV system S with $f \geq 1$ the count-instability of S is at least 1, and the length-instability of S is ∞.*

PROOF. Same proof as for EV-*validity*. □

THEOREM 7. *For any RV system S with $v \geq n$ and $n + \lfloor \frac{n-1}{2} \rfloor < 5f$ the count-instability of S is at least 2.*

PROOF. Let m_0 be any state and $\vec{x_0} = (0, 1, \ldots, n-1)$. Due to RV-validity, $f \leq \mathcal{F}(\vec{x}_0, m_0) \leq n - f - 1$. There are 2 cases: $\mathcal{F}(\vec{x}_0, m_0) \leq \lfloor \frac{n-1}{2} \rfloor$ and $\mathcal{F}(\vec{x}_0, m_0) > \lfloor \frac{n-1}{2} \rfloor$; for symmetry reasons they are equivalent.

We can thus consider only the case $\mathcal{F}(\vec{x}_0, m_0) \leq \lfloor \frac{n-1}{2} \rfloor$. Mark by $k := \mathcal{F}(\vec{x}_0, m_0)$, thus $f \leq k \leq \lfloor \frac{n-1}{2} \rfloor < 2f$. Let

$$\vec{x}_1 = (\underbrace{n, n, \ldots, n}_{k-f+1}, \underbrace{k-f+1, k-f+2, \ldots, n-1}_{n-k+f-1}) .$$

Consider $\mathcal{F}(\vec{x}_1, m_1)$, where $m_1 := \tau(\vec{x}_0, m_0)$. Notice that $\vec{x}_1$ contains $k-f+1$ values of "n", thus (due to RV-validity) $f - (k-f+1) = 2f-k-1$ additional values are "to be ignored". Therefor, the output must be $\leq n - 1 - (2f - k - 1) = n+k-2f$. That is, due to RV-validity we have that $k+1 \leq \mathcal{F}(\vec{x}_1, m_1) \leq n+k-2f$. Since $k = \mathcal{F}(\vec{x}_0, m_0)$ it holds that $\mathcal{F}(\vec{x}_0, m_0) \neq \mathcal{F}(\vec{x}_1, m_1)$.

Define $k' = \mathcal{F}(\vec{x}_1, m_1)$. If $k' < 2f$ then define

$$\vec{x}_2 := (\underbrace{n, n, \ldots, n}_{f}, \underbrace{f, f+1, \ldots, n-1}_{n-f}) .$$

In this case, $2f \leq \mathcal{F}(\vec{x}_2, m_2) \leq n-1$ (where $m_2 := \tau(\vec{x}_1, m_1)$). And we have that $\mathcal{F}(\vec{x}_1, m_1) \neq \mathcal{F}(\vec{x}_2, m_2)$. Therefore, the run from m_0 on the path $\vec{x}_0, \vec{x}_1, \vec{x}_2$ has 2 output changes.

On the other hand, consider the case where $k' \geq 2f$. Let

$$\vec{x}_2 := (\underbrace{0, .., 0}_{k-f+1}, \underbrace{k-f+1, k-f+2, .., n+k-2f}_{n-f}, \underbrace{0, .., 0}_{2f-k-1}) .$$

[5]RV's definition is taken from [7]. Note that the current paper's EV definition differs from the EV in [7].

Notice that $\vec{x}_2$ contains exactly f "0"s. Moreover, $\vec{x}_2$ contains exactly f values that are $> n+k-3f$; thus, due to RV-validity we have that $\mathcal{F}(\vec{x}_2, m_2) \leq n+k-3f$. Since $k \leq \lfloor \frac{n-1}{2} \rfloor$, it holds that that $\mathcal{F}(\vec{x}_2, m_2) \leq n + \lfloor \frac{n-1}{2} \rfloor - 3f$.

We assumed $k' \geq 2f$ and since $n + \lfloor \frac{n-1}{2} \rfloor < 5f$, thus $\mathcal{F}(\vec{x}_2, m_2) < k' = \mathcal{F}(\vec{x}_1, m_1)$. That is, there were 2 output changes. □

6.2 Solving 1-SSRC for RV-validity

6.2.1 Definitions

Prior to solving the 1-SSRC problem, a definition of the problem in the context of RV-*validity* is required. The only change in the definition from EV-*validity* is in the "Validity" property of the (k, Δ)-repetitive consensus behavior definition. We thus adapt this definition in the following manner:

DEFINITION 6.1. *For RV-*validity *let the "Validity" property of the (k, Δ)-repetitive consensus behavior be:*
For any beat $r \in R'_{|\Delta}$, if $\mathcal{V}^r$ is the output value then some non-faulty node has input value $\nu_1 \leq \mathcal{V}^r$ and some non-faulty node has input value $\nu_2 \geq \mathcal{V}^r$.

COROLLARY 2. *If $v \geq n$ and $n + \lfloor \frac{n-1}{2} \rfloor < 5f$, the 1-RC (and 1-SSRC) problem cannot be solved for RV-validity.*

PROOF. Follows immediately from Theorem 7. □

6.2.2 Algorithm

The algorithm SS-REP-CONS also solves the 1-SSRC problem - for $n \geq 4f+1$ - in the RV-*validity* setting, provided that we replace Line 2.a with the following:

- **Remove** f lowest and f highest values from *vals*. Let *high* be the new highest value in *vals*, and *low* be the new lowest value (after the removal);
- **If** $Output_p \notin [low, high]$ then set $Output_p$ to be the median value in *vals*.

This change incorporates the difference between EV-*validity* and RV-*validity*.

6.2.3 Proofs

THEOREM 8. SS-REP-CONS *(with the above changes) solves the $1-SSRC$ problem for $n \geq 4f+1$.*

PROOF. Notice that Lemma 1, Lemma 2 and Lemma 3 all hold for RV-*validity*. Thus, it is left to show that once the system has stabilized, it does not change its output more than once. Consider v' to be the value of $Output_p$ at all nodes after SS-REP-CONS has stabilized (v' is well defined due to Lemma 2).

Consider the first beat in which $Output_p$ changes to $v \neq v'$; Sort the input values of the correct nodes, and let v_{low} be the $f+1$st input value from the bottom, and v_{high} be the $f+1$st from the top. The median value v has at least $2f+1$ values that are lower or equal to it. There are at most f faulty nodes, and thus there are at least $f+1$ correct nodes lower or equal to v. Thus, $v_{low} \leq v$. For the same reason $v_{high} \geq v$. Thus, the calculated median is in the range $[v_{low}, v_{high}]$; in other words, $v \in [v_{low}, v_{high}]$.

In any future beat, when values are removed from *vals*, the values v_{low} and v_{high} are not discarded, since only the f

lowest and f highest values are removed. Thus, when calculating *low* and *high* (see algorithm in previous subsection) the following holds: $low \leq v_{low}$ and $high \geq v_{high}$. Thus, $Output_p \in [low, high]$, which means that $Output_p$ is not updated; *i.e.* , $Output_p = v$. Thus, the output value of the correct nodes changes at most once after SS-REP-CONS has converged. □

7. DISCUSSION

7.1 Related Problems

There are two problems that seem to be related to repetitive consensus: self-stabilizing *Byzantine* agreement, and continuous consensus. However, both problems differ from repetitive consensus on the same major issue.

Self-stabilizing *Byzantine* Agreement (SSBA): This problem consists of having a *Byzantine* agreement that is self-stabilizing (see [3] for more information). That is, starting from an arbitrary memory state, any node p can initiate an agreement procedure on its value. Due to the self-stabilizing property, p can repeatedly initiate agreement procedures on its value; thus, in a sense, SSBA can be seen as a repetitive *Byzantine* agreement. The main difference between repetitive consensus and SSBA lies in the difference in their *validity* property. SSBA requires that if the leader is non-faulty, then all non-faulty nodes agree on the leader's value. Since SSBA requires an agreement on a single (leader) node's value, if this value changes then the agreement should change with it. However, in repetitive consensus, we require a consensus of all nodes' values: thus, if one node's value changes, and this node is *Byzantine*, then the output value should not change. Therefore, in repetitive consensus, a single node changing its input value must not lead to a change in the output, where as in SSBA a single node's change of input value should lead (if it is the leader) to the change of the output value.

Continuous Consensus (CC): The CC problem involves continuously agreeing on all the inputs of all the nodes in the system. That is, each node should have a list of all the inputs that each node in the system has had until now (see [12] for more information). However, the property that all nodes agree on the input values of all other nodes during the entire execution is still not sufficient to solve the 0-RC problem.

As in SSBA, the essence of this impossibility lies in the requirement that if a *Byzantine* node changes its input value, the output value of the repetitive consensus does not change. Thus, in CC, even if all non-faulty nodes have all the input values of all nodes, they cannot differentiate between a non-faulty node that has changed its value, and a *Byzantine* node mimicking a new input value.

REMARK 7.1. *The above comparison leads to the conclusion that the impossibility result of Corollary 1 stems from the requirements of the repetitive consensus problem, and not from the ability to gather information from the entire system. In other words, it is not missing data that prohibits the repetitive consensus output; there simply does not exist a function on the nodes' inputs that satisfies the requirements of the 0-RC problem.*

7.2 Pseudo Self-stabilization

In the context of self-stabilization, algorithms that converge to a "safe" state and leave the safe states at most a constant number of times are called "pseudo self-stabilizing" algorithms (see [6]). Depending on how one defines a *safe configuration* in the context of *Byzantine* faults, the algorithms presented in the current paper may be considered as pseudo self-stabilizing, where the constant number of times they may leave "safe" states is 1 or 2.

7.3 Bounded-delay Network

In the full paper the results herein are extended to the bounded-delay model, in which there is no global-beat-system; instead, a bound on messages' delivery time is assumed. The main idea behind this extension is the usage of a "transformer" from the global-beat-system model to the bounded-delay model, which conserves the properties of SS-REP-CONS.

7.4 Open Questions

We have shown that for any system with $f > 0$ the count-instability is at least 1, and that this is tight for $|V| = 2$. In addition, for systems with $n = 3f + 1$ (where $|V| \geq 3$) we have shown that the count-instability is at least 2, and SS-REP-CONS reaches this bound. This raises the following question: what is the exact relation between $|V|, n$ and f regarding count-instability and what algorithm can achieve it?

When considering the RV-*validity* scenario it becomes even more interesting: we have shown how to solve the 1-SSRC problem for any $n \geq 4f + 1$. What happens for smaller n? For $n \leq 4f$ we have only given lower bounds. Can upper bounds be given for $n \leq 4f$?

Acknowledgements

We would like to thank Yoni Peleg for contributing Theorem 5.

8. REFERENCES

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

[2] A. Bar-Noy, X .Deng, J. Garay, and T. Kameda. Optimal amortized distributed consensus. *Information and Computation*, 120(1):93–100, 1995.

[3] A. Daliot and 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.

[4] L. Davidovitch, S. Dolev, and S. Rajsbaum. Stability of multivalued continuous consensus. *SIAM Journal on Computing*, 37(4):1057–1076, 2007.

[5] D. Dolev and E. N. Hoch. On self-stabilizing synchronous actions despite byzantine attacks. In *Proc. the 21st Int. Symposium on Distributed Computing (DISC'07)*, Lemesos, Cyprus, Sep. 2007.

[6] S. Dolev. *Self-Stabilization*. The MIT Press, 2000.

[7] S. Dolev and S. Rajsbaum. Stability of long-lived consensus. *J. Comput. Syst. Sci.*, 67(1):26–45, 2003.

[8] Michael J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In Marek Karpinski, editor, *FCT*, volume 158 of *Lecture Notes in Computer Science*, pages 127–140. Springer, 1983.

[9] Eli Gafni, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. The committee decision problem. In José R. Correa, Alejandro Hevia, and Marcos A. Kiwi, editors, *LATIN*, volume 3887 of *Lecture Notes in Computer Science*, pages 502–514. Springer, 2006.

[10] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[11] Keith Marzullo. Tolerating failures of continuous-valued sensors. *ACM Trans. Comput. Syst.*, 8(4):284–304, 1990.

[12] Tal Mizrahi and Yoram Moses. Continuous consensus via common knowledge. In *TARK '05: Proceedings of the 10th conference on Theoretical aspects of rationality and knowledge*, pages 236–252, Singapore, Singapore, 2005. National University of Singapore.

[13] Roberto De Prisco, Butler Lampson, and Nancy Lynch. Revisiting the PAXOS algorithm. *Theoretical Computer Science*, 243(1–2):35–91, 2000.

[14] Marco Schneider. Self-stabilization. *ACM Comput. Surv.*, 25(1):45–67, 1993.

[15] F. Becker and S. Rajsbaum and I. Rapaport and E. Re'mila. Average binary long-lived Consensus: quantifying the stabilization role played by memory. In *Proc. 15th International Colloquium on Structural Information and Communication Complexity (SIROCCO'08)*, Switzerland, June. 2008.

Chapter 5

Constant-space Localized Byzantine Consensus

Danny Dolev and Ezra N. Hoch, Proceedings of the 22nd international symposium on Distributed Computing (DISC '08), Arcachon, France, Pages: 167 - 181, Sep. 2008.

Constant-Space Localized Byzantine Consensus

Danny Dolev* and Ezra N. Hoch

School of Engineering and Computer Science,
The Hebrew University of Jerusalem, Israel
{dolev,ezraho}@cs.huji.ac.il

Abstract. Adding *Byzantine* tolerance to large scale distributed systems is considered non-practical. The time, message and space requirements are very high. Recently, researches have investigated the *broadcast problem* in the presence of a f_ℓ-local *Byzantine* adversary. The local adversary cannot control more than f_ℓ neighbors of any given node. This paper proves sufficient conditions as to when the synchronous *Byzantine consensus problem* can be solved in the presence of a f_ℓ-local adversary.

Moreover, we show that for a family of graphs, the *Byzantine* consensus problem can be solved using a relatively small number of messages, and with time complexity proportional to the diameter of the network. Specifically, for a family of bounded-degree graphs with logarithmic diameter, $O(\log n)$ time and $O(n \log n)$ messages. Furthermore, our proposed solution requires constant memory space at each node.

1 Introduction

Fault tolerance of a distributed system is highly desirable, and has been the subject of intensive research. *Byzantine* faults have been used to model the most general and severe failures. Classic *Byzantine*-tolerant research has concentrated on an "all mighty" adversary, which can choose up to f "pawns" from the n available nodes (usually, $f < \frac{n}{3}$). These *Byzantine* nodes have unlimited computational power and can behave arbitrarily, even colluding to "bring the system down". Much has been published relating to this model (for example, [11], [12], [2]), and many lower bounds have been proven.

One of the major drawbacks to the classic *Byzantine* adversarial model is its heavy performance cost. The running time required to reach agreement is linear in f (which usually means it is also linear in n), and the message complexity is typically at least $O(n^2)$ (when $f = O(n)$, see [7]). This drawback stems from the "global" nature of the classic *Byzantine* consensus problem - i.e., the *Byzantine* adversary has no restrictions on the spread of faulty nodes. Thus, the communication graph must have high connectivity (see [6] for exact bounds) - which leads to a high message load. Moreover, the global nature of the adversary requires every node to agree with every other node, leading to linear termination time and to a high out degree for each node (at least $2f + 1$ outgoing connections,

* Supported in part by ISF.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 167–181, 2008.

see [6]). Such algorithms cannot scale; thus - as computer networks grow - it becomes infeasible to address global *Byzantine* adversaries.

To overcome this limitation, some research has assumed computationally-bounded adversaries, for example [5]. Alternatively, recent work has considered a f_ℓ-local *Byzantine* adversary. The f_ℓ-local adversary is restricted in the nodes it can control. For every node p, the adversary can control at most f_ℓ neighboring nodes of p. We call this stronger model "local" and the classic model "global". [10] has considered the *Byzantine* broadcast problem, which consists of all non-*Byzantine* nodes accepting a message sent by a given node. [13] classifies the graphs in which the broadcast problem can be solved, in the presence of a f_ℓ-local adversary. In the current work we consider the *Byzantine* consensus problem, which consists of all non-*Byzantine* nodes agreeing on the same output value, which must be in the range of the input values of the non-*Byzantine* nodes. Sufficient conditions are given to classify graphs on which the problem can be solved, in the presence of a f_ℓ-local adversary.

Solving the *Byzantine* consensus problem when facing a global adversary requires $O(n)$ memory space. When considering a local adversary, the memory space requirements can be reduced. This raises the question of possible tradeoff between fault tolerance and memory space requirement, which has been previously investigated in the area of self-stabilizing (see [3], [8], [9]). The current work provides a new tradeoff: for a family of graphs, consensus can be solved using constant memory space, provided that the *Byzantine* adversary is f_ℓ-local.

Contribution: This work solves the *Byzantine* consensus problem in the local adversary model. For a large range of networks (a family of graphs with constant degree and logarithmic diameter), *Byzantine* consensus is solved within $O(\log n)$ rounds and with message complexity of $O(n \cdot \log n)$, while requiring $O(1)$ space at each node[1]. These results improve exponentially upon the classic setting which requires linear time, $O(n^2)$ messages, and (at least) linear space for reaching a consensus. We also present two additional results which are of special interest while using constant memory: first, we show a means of retaining identities in a local area of the network (see Section 4.1); second, we present a technique for waiting $\log n$ rounds using $O(1)$ memory space at each node (see Section 4.2).

2 Model and Problem Definition

Consider a synchronous distributed network of n nodes, $\{p_0, \ldots, p_{n-1}\} = \mathcal{P}$, represented by an undirected graph $G = (E, V)$, where $V = \mathcal{P}$ and $(p, p') \in E$ if p, p' are connected. Let $\Gamma(p)$ be the set of neighbors of p, including p. The communication network is assumed to be synchronous, and communication is done via message passing; each communication link is bi-directional.

Byzantine nodes have unlimited computational power and can communicate among themselves. The *Byzantine* adversary may control some portion of the nodes; however, the adversary is limited in that each node may not have more

[1] Formally, we show that each node uses space polynomial in its degree; for constant degree it is $O(1)$.

than f_ℓ *Byzantine* neighbors, where f_ℓ is a system-wide constant. Formally, a subset $S \subset \mathcal{P}$ is f_ℓ-local if for any node $p \in \mathcal{P}$ it holds that $|S \bigcap \Gamma(p)| \leq f_\ell$. A *Byzantine* adversary is said to be f_ℓ-local if (in any run) the set of *Byzantine* nodes is f_ℓ-local.

2.1 Problem Definition

The following is a formal definition of the *Byzantine* consensus problem, followed by a memory-bounded variant of it.

Definition 1. *The* ***Byzantine*** **consensus problem** *consists of the following: Each node p has an input value v_p from an ordered set $\mathcal{V}$; all nodes agree on the same output $V \in \mathcal{V}$ within a finite number of rounds (agreement), such that $v_p \leq V \leq v_q$ for some correct nodes p, q (validity).*

The goal of the current work is to show that for f_ℓ-local adversaries, efficient deterministic solutions to the *Byzantine* consensus problem exist; efficient with respect to running time, message complexity and space complexity. A node p's memory space depends solely on the local network topology - i.e. p's degree or the degree of p's neighbors.

Denote by ℓ-MaxDeg(p) the maximal degree of all nodes that are up to ℓ hops away from p. Notice that 0-MaxDeg(p) $= |\Gamma(p)|$.

Definition 2. *The* **ℓ-hop local *Byzantine*** *consensus problem is the* Byzantine *consensus problem with the additional constraint that each correct node p can use memory space that is polynomially bounded by ℓ-*MaxDeg(p).

Informally, the above definition states that a node cannot "know" about all the nodes in the system, even though it can access all its local neighbors. That is, consensus must be reached with "local knowledge" only.

Remark 1. Definition 2 can be replaced by a requirement that the out degree of each node as well as the memory space are constant. We choose the above definition instead, as it generalizes the constant-space requirement.

3 *Byzantine* Consensus

In this section sufficient conditions for solving the *Byzantine* consensus problem for a f_ℓ-local adversary are presented. First we discuss solving the *Byzantine* consensus problem in a network that is fully connected. We later give some definitions that allow us to specify sufficient conditions to ensure that *Byzantine* consensus can be solved in networks that are not fully connected.

3.1 Fully Connected *Byzantine* Consensus

We now define ByzCon, which solves the *Byzantine* consensus problem in a fully connected network. Take any *Byzantine* agreement[2] algorithm (for example, see

[2] *Byzantine* agreement is sometimes called *Byzantine* broadcast. This problem consists of a single leader broadcasting some value v using point-to-point channels.

[14]), and denote it by BA. ByzCon executes n instance of BA, one for each node p's input value, thus agreeing on the input value v_p. As a result, all correct nodes have an agreed vector of n input values (notice that this vector may contain "$\perp$" values when BA happens to return such a value for a *Byzantine* node). The output value of ByzCon is the median of the values of that vector, where "$\perp$" is considered as the lowest value.

Claim. ByzCon solves the *Byzantine* consensus problem in a fully-connected network.

Definition 3. *Given an algorithm $\mathcal{A}$ and a set of nodes $S \subset \mathcal{P}$,* Valid*$(\mathcal{A}, S)$ = true if the set S upholds the connectivity requirements of $\mathcal{A}$.* Faulty*$(\mathcal{A}, S)$ denotes the maximal number of faulty nodes that $\mathcal{A}$ can "sustain" when executed on S (for example,* Faulty$(BA, S) = \lfloor \frac{|S|-1}{3} \rfloor$*).*

In a similar manner, Time$(\mathcal{A}, S)$ *is the maximal number of rounds it takes $\mathcal{A}$ to terminate when executed on S, and* Msg$(\mathcal{A}, S)$ *is the maximal number of messages sent during the execution of $\mathcal{A}$ on S.*

Notice that for all S: Valid(ByzCon, S)= Valid(BA, S), Faulty(ByzCon, S)= Faulty(BA, S), Time(ByzCon, S) = Time(BA, S) Msg(ByzCon, S) = $|S| \cdot$ Msg(BA, S). That is, ByzCon's connectivity requirements, fault tolerance ratio and running time, are the same as in the *Byzantine* agreement algorithm of [14]; *i.e.,* ByzCon requires a fully connected graph among the nodes participating in ByzCon and it supports up to a third of them being *Byzantine*.

3.2 Sparsely Connected *Byzantine* Consensus

Consider a given algorithm that solves the *Byzantine* consensus problem when executed on a set of nodes S. ByzCon, defined in the previous section, requires S to be fully-connected. The following discussion assumes ByzCon's existence and correctness.

The following definitions hold with respect to any f_ℓ-local adversary.

Definition 4. *Given a subset $S \subset \mathcal{P}$, denote by f_ℓ-$Byz(S)$ the maximal number of nodes from S that may be* Byzantine *for a f_ℓ-local adversary.*

Definition 5. *A non-empty subset S, $\emptyset \neq S \subset \mathcal{P}$, is a* **$f_\ell$-decision group** *if* Valid*(*ByzCon*, S) = true and* Faulty(ByzCon, S) $\geq f_\ell$-$Byz(S)$.

When considering ByzCon as constructed in Section 3.1, the above definition states that S must be fully connected, and $|S| > 3 \cdot f_\ell$. Since 0-local adversaries are of no interest, the minimal size of S satisfies, $|S| \geq 4$.

Definition 6. *A non-empty subset $S' \subseteq S$ of a f_ℓ-decision group is a* **f_ℓ-common** *source if f_ℓ-$Byz(S') + |S - S'| \leq$* Faulty(ByzCon, S).

Claim. If $S' \subseteq S$ is a common source and all correct nodes in S' have the same initial value, ν, then the output value of ByzCon when executed on S, is ν.

Proof. Denote by W the set of correct nodes in S' and by Y the set of all nodes in S that are not in W; *i.e.,* $Y := S - W$. Since S' is a common source, $|Y| \leq \text{FAULTY}(\text{BYZCON}, S)$. Assume by way of contradiction that all nodes in W have the same initial value ν, and the output of BYZCON (when executed on S) is not ν; denote this execution by $\mathcal{R}$. If all the nodes in Y are *Byzantine* and they simulate their part of $\mathcal{R}$, then it holds that all correct nodes in S have the same initial value ν, the number of *Byzantine* nodes is less than $\text{FAULTY}(\text{BYZCON}, S)$, and yet the output is not ν. In other words, the "validity" of BYZCON does not hold. Therefore, if all nodes in $W = S - Y$ are correct and have the same initial value ν, that must be the output value of BYZCON when executed on S. □

Definition 7. *Two subsets $S_1, S_2 \subset \mathcal{P}$ are f_ℓ-***connected** *if S_1, S_2 are f_ℓ-decision groups, and if $S_1 \cap S_2$ is a f_ℓ-common source for both S_1 and S_2.*

Definition 8. *A list $C = S_1, S_2, \ldots, S_l$ is a f_ℓ-***value-chain** *between S_1 and S_l if the subsets S_i, S_{i+1} are f_ℓ-connected, for all $1 \leq i \leq l-1$. The* **length** *of the value-chain C is $l-1$.*

Definition 9. *Let $\mathcal{G}$ be a set of f_ℓ-decision groups,* i.e., *$\mathcal{G} \subseteq 2^{\mathcal{P}}$. $\mathcal{G}$ is an f_ℓ-***entwined structure** *if for any two subsets $g, g' \in \mathcal{G}$ there is a f_ℓ-value-chain $C = g_1, g_2, \ldots, g_l \in \mathcal{G}$ between g, g'. The* **distance** *between g, g' is the minimal length among all f_ℓ-value-chains between g, g'.*

Definition 10. *The* **diameter** *$\mathcal{D}$ of an f_ℓ-entwined structure $\mathcal{G}$ is the maximal distance between any two f_ℓ-decision groups $g, g' \in \mathcal{G}$.*

Definition 11. *An f_ℓ-entwined structure $\mathcal{G}$ is said to* **cover** *graph G if for any node p in G there is a subset $g \in \mathcal{G}$ such that $p \in g$. Formally: $\bigcup_{g \in \mathcal{G}} \{g\} = \mathcal{P}$.*

Remark 2. All entwined structures in the rest of this paper are assumed to cover their respective graphs. We will therefore sometimes say "an entwined structure $\mathcal{G}$" instead of "an entwined structure $\mathcal{G}$ that covers graph G".

Some of the above definitions were parameterized by f_ℓ. When f_ℓ is clear from the context, we will remove the prefix f_ℓ. (*i.e.,* "decision group" instead of "f_ℓ-decision group").

Definition 12. *Let G be a graph. Denote by $\Phi(G)$ the maximal value f_ℓ s.t. there is a f_ℓ-entwined structure that covers G.*

The following theorem states the first result of the paper.

Theorem 1. *The* Byzantine *consensus problem can be solved on graph G for any $\Phi(G)$-local adversary.*

Section 3.3 contains the algorithm LOCALBYZCON that given a f_ℓ-entwined structure $\mathcal{G}$, solves the *Byzantine* consensus problem for any f_ℓ-local adversary. Section 3.4 proves the correctness of LOCALBYZCON, thus completing the proof of Theorem 1.

Algorithm LocalByzCon /* *executed at node* p */
/* $\mathcal{BC}_i$ *is an instance of* ByzCon */

Initialization:

1. **set** $v_p := p$'s initial value;
2. **set** $Output_p := \emptyset$; /* $\mathcal{G}_p := \{g \in \mathcal{G} | p \in g\}$ */
3. **for** all $g_i \in \mathcal{G}_p$ start executing $\mathcal{BC}_i$ with initial value v_p;

For $\Delta_{max} \cdot (2\mathcal{D} + 1)$ **rounds:** /* $\Delta_{max} := max_i\{\text{Time}(\text{ByzCon}, g_i)\}$ */

1. **execute** a single round of each $\mathcal{BC}_i$ that p participates in;
2. **for each** $\mathcal{BC}_i$ that terminated in the current round with value V:
   set $Output_p := Output_p \bigcup \{V\}$;
3. **for each** $\mathcal{BC}_i$ that terminated in the current round:
   start executing $\mathcal{BC}_i$ with initial value $\min\{Output_p\}$;

Return value:

return output as $\min\{Output_p\}$;

Fig. 1. Solving the *Byzantine* consensus problem for a f_ℓ-local adversary

3.3 Algorithm LocalByzCon

Figure 1 introduces the algorithm LocalByzCon that solves the *Byzantine* consensus problem for f_ℓ-local adversaries, given an f_ℓ-entwined structure $\mathcal{G}$. The main idea behind LocalByzCon is to execute ByzCon locally in each decision group. Each node takes the minimal agreement value among the decision groups it participated in. The fact that any two decision groups in $\mathcal{G}$ have a value-chain between them ensures that the minimal agreement value among the different invocations of ByzCon will propagate throughout $\mathcal{G}$. Since $\mathcal{G}$ covers G, all nodes will eventually receive the same minimal value.

Consider $\mathcal{G}$ to be a f_ℓ-entwined structure, and $g_1, g_2, \ldots, g_m \in \mathcal{G}$ to be all the decision groups (of $\mathcal{P}$) in $\mathcal{G}$. For a node p, $\mathcal{G}_p$ is the set of all decision groups that p is a member of; that is, $\mathcal{G}_p := \{g \in \mathcal{G} | p \in g\}$. Each node p participates in repeated concurrent executions of ByzCon instances, where for each $g_i \in \mathcal{G}_p$, node p will execute a ByzCon instance, and once that instance terminates p will execute another instance, etc. For each $g_i \in \mathcal{G}_p$ denote by $\mathcal{BC}_i^1$ the first execution of ByzCon by all nodes in g_i; $\mathcal{BC}_i^2$ denotes the second instance executed by all nodes in g_i, and so on.

According to Definition 1 all nodes participating in ByzCon terminate within some finite time Δ. Furthermore, each node can wait until Δ rounds elapse and terminate, thus achieving simultaneous termination. Therefore, in LocalByzCon, all nodes that participate in $\mathcal{BC}_i^j$ terminate it at the same round and start executing $\mathcal{BC}_i^{j+1}$ together at the following round.

3.4 Correctness Proof

For every $g_i \in \mathcal{G}$ denote by $r_i(1)$ the round at which the first consecutive instance of ByzCon executed on g_i has terminated. Denote by $r(1) := \max\{r_i(1)\}$. Let $Output_p^r$ denote the value of $Output_p$ at the end of round r. Notice that $Output_p^r \subseteq Output_p^{r+1}$ for all correct p and all r. Denote $Output^r := \bigcup\{Output_p^r\}$,

the union of all $Output_p$ (for correct p) at the end of some round r. Using this notation, $Output^{r(1)}$ represents all the values in any $Output_p$ (for correct p) after at least one instance of ByzCon has terminated for each $g \in \mathcal{G}$. Consider some instance of ByzCon that terminates after round $r(1)$: it must be (at least) a second execution of that instance, thus all correct nodes that participated in it had their input values chosen from $Output^{r(1)}$. Thus, due to *validity*, the output value is in the range of $[\min\{Output^{r(1)}\}, \max\{Output^{r(1)}\}]$. Hence, we conclude that $\min\{Output^{r}\} \geq \min\{Output^{r(1)}\}$ for any $r \geq r(1)$. Denote by $v_{min} := \min\{Output^{r(1)}\}$; clearly no correct node p will hold a lower value (in $Output_p$) for any $r \geq r(1)$.

Lemma 1. *If a correct node p has $\min\{Output_p^r\} = v_{min}$ then it will never have a lower value in $Output_p$ for any round $r' \geq r$.*

Lemma 2. *At round $r(1)$, there exists $g_i \in \mathcal{G}$ such that for every correct node $p \in g_i$ it holds that $\min\{Output_p\} = v_{min}$.*

Proof. By definition, $v_{min} \in Output_p^{r(1)}$. Thus, v_{min} was the output of some $\mathcal{BC}_i$ instance (on decision group g_i) at some round $r \leq r(1)$. Consider the nodes in g_i, they have all added v_{min} to their $Output_p$. Thus, $v_{min} \in Output_p^{r(1)}$ and by definition it is the lowest value in $Output_p$ at round $r(1)$. Thus, at round $r(1)$, all correct nodes in g_i have $\min\{Output_p\} = v_{min}$. □

Lemma 3. *If* LocalByzCon *has been executed for at least $\Delta_{max} \cdot (2\mathcal{D} + 1)$ rounds, then all correct nodes have $\min\{Output_p\} = v_{min}$.*

Proof. Divide the execution of LocalByzCon into "stages" of Δ_{max} rounds each. Clearly there are at least $2\mathcal{D} + 1$ stages, and in each stage each $\mathcal{BC}_i$ is started at least once. From the above lemma, for some decision group g_i, all correct nodes $p \in g_i$ have $\min\{Output_p\} = v_{min}$ at the end of the first stage.

Let g' be some decision group, and let $g_i = g_1, g_2, \ldots, g_l = g'$ be a value-chain between g_i and g'; there exists such a value-chain because $\mathcal{G}$ is an entwined structure, and its length is $\leq \mathcal{D}$.

Consider the second stage. Since g_1, g_2 are connected, and since all nodes in g_1 have the same initial value (v_{min}) during the entire stage 2, then in $g_1 \cap g_2$ all nodes have v_{min} as their initial value during stage 2. Since $g_1 \cap g_2$ is a common source of g_2, it holds that instance $\mathcal{BC}_2$ that is started in stage 2 is executed with all nodes in $g_1 \cap g_2$ having initial value v_{min}. Thus, when $\mathcal{BC}_2$ terminates (no later than the end of stage 3), it terminates with the value v_{min}, thus all nodes in g_2 also have $v_{min} \in Output_p$. By Lemma 1, all nodes in g_2 choose v_{min} as their initial value for any instance of ByzCon started during stage 4 and above. Repeating this line of proof leads to the conclusion that after an additional $2\mathcal{D}$ stages all correct nodes in g' have $v_{min} \in Output_p$.

Since any decision group g' has a value-chain of no more than $\mathcal{D}$ length to g_i, we have that after $2\mathcal{D} + 1$ stages, all correct nodes in all decision groups have $v_{min} \in Output_p$. Since $\mathcal{G}$ covers G, each correct node is a member of some decision group, thus all correct nodes in G have $v_{min} \in Output_p$. Since v_{min} is the lowest possible value, $\min\{Output_p\} = v_{min}$ for all $p \in \mathcal{P}$. □

Remark 3. Consider the value-chain $g_1, g_2, \ldots, g_l$ in the proof above. The proof bounds the time (in rounds) it takes v_{min} to "propagate" from g_1 to g_l. The given bound $(2 \cdot l \cdot \Delta_{max})$ is not tight. In fact, instead of assuming $2 \cdot \Delta_{max}$ rounds for each "hop along the chain", one can accumulate $2 \cdot \textsc{Time}(\textsc{ByzCon}, g_i)$ when moving from g_{i-1} to g_i. Thus, define $\textsc{Time}_{dist}(g, g')$ to be the shortest such sum on any value-chain between g, g', and $\mathcal{D}_{\textsc{time}}$ as the maximal $\textsc{Time}_{dist}(g, g')$ on any $g, g' \in \mathcal{G}$; and we can conclude that it is enough to run LocalByzCon for $\mathcal{D}_{\textsc{time}}$ rounds. Notice that this analysis it tight up to a factor of 2.

Lemma 4. *Given an f_ℓ-entwined structure $\mathcal{G}$ that covers G,* LocalByzCon *solves the* Byzantine *consensus problem, for a f_ℓ-local adversary.*

Proof. From the above lemmas, after $\Delta_{max} \cdot (2\mathcal{D} + 1)$ rounds, all correct nodes terminate with the same value, v_{min}. Notice that v_{min} is the output of some ByzCon instance. Thus, there are two correct nodes p, q such that $v_p \leq v_{min} \leq v_q$. Therefore, both "agreement" and "validity" hold. □

3.5 Complexity Analysis

The time complexity of LocalByzCon is $(2\mathcal{D} + 1) \cdot \Delta_{max}$. In other words, let g_{max} be the largest decision group in $\mathcal{G}$, that is $g_{max} := argmax_{g_i \in \mathcal{G}}\{|g_i|\}$; using this terminology we have that, $\textsc{Time}(\textsc{LocalByzCon}, \mathcal{P}) = (2\mathcal{D} + 1) \cdot O(g_{max})$.

Similarly, the message complexity of LocalByzCon per round is the sum of all messages of all ByzCon instances each round, which is bounded by $\sum_{g_i} |g_i|^3 \leq |\mathcal{G}| \cdot |g_{max}|^3$. Thus, $\textsc{Msg}(\textsc{LocalByzCon}, \mathcal{P}) \leq (2\mathcal{D} + 1) \cdot |\mathcal{G}| \cdot O(|g_{max}|^4)$ (messages per round times rounds).

4 Constant-Space *Byzantine* Consensus

Section 3 proves a sufficient condition for solving the *Byzantine* consensus problem on a given graph G for a f_ℓ-local adversary. The current section gives a sufficient condition for solving the ℓ-hop local *Byzantine* consensus problem.

Definition 13. *An f_ℓ-entwined structure $\mathcal{G}$ is called* **ℓ-hop local** *if for every $g \in \mathcal{G}$, ℓ bounds the distance between any two nodes in g.*

For ByzCon constructed in Section 3.1, any entwined structure $\mathcal{G}$ is 1-hop local, since for every decision group $g \in \mathcal{G}$, it holds that $\textsc{Valid}(\textsc{ByzCon}, g) = true$, thus g is fully connected. However, the following discussion holds for any algorithm that solves the *Byzantine* consensus problem, even if it does not require decision groups to be fully connected.

Definition 14. *An f_ℓ-entwined structure $\mathcal{G}$ is called* **ℓ-lightweight** *if $\mathcal{G}$ is ℓ-hop local and for every $p \in \mathcal{P}$, $|\mathcal{G}_p|$ and $|g|$ (for all $g \in \mathcal{G}_p$) are polynomial in $|\Gamma(p)|$.*

Definition 15. *Let G be a graph. Denote by ℓ-$\Psi(G)$ the maximal value f_ℓ s.t. there is a f_ℓ-entwined structure that is ℓ-lightweight and covers G.*

The following theorem states the second contribution of this paper.

Theorem 2. *The ℓ-hop local* Byzantine *consensus problem can be solved on graph G for any $[\ell\text{-}\Psi(G)]$-local adversary.*

By Theorem 1, any f_ℓ-entwined structure $\mathcal{G}$ that covers graph G can be used to solve the *Byzantine* consensus problem on G, for a f_ℓ-local adversary. To prove Theorem 2 we show that when LocalByzCon is executed using an ℓ-lightweight f_ℓ-entwined structure, each node p's space requirements are polynomial in ℓ-MaxDeg(p). There are 3 points to consider: first, we show that the memory footprint of the different ByzCon instances that p participates in is "small". Second, ByzCon assumes unique identifiers, which usually require $\log n$ space. We need to create identifiers that are locally unique (thus requiring space that is independent of n), such that ByzCon can be executed properly on each decision group. Third, the main loop of LocalByzCon requires to count at least up to $\mathcal{D}$, which requires $\log \mathcal{D}$ bits, possibly requiring space that is dependent on n.

The second point is discussed in Section 4.1 and the third in Section 4.2. To solve the first point notice that $\mathcal{G}$ is ℓ-lightweight, thus each node participates in no more than $poly(|\Gamma(p)|)$ ByzCon instances concurrently, and each such instance contains $poly(|\Gamma(p)|)$ nodes. Assuming that the identifiers used by ByzCon require at most $polylog(\ell\text{-MaxDeg}(p))$ bits (see Section 4.1), p requires at most $poly(\ell\text{-MaxDeg}(p))$ space to execute the ByzCon instances of LocalByzCon.

4.1 Locally Unique Identifiers

Consider the algorithm LocalByzCon in Figure 1 and an ℓ-lightweight entwined structure $\mathcal{G}$. Different nodes communicate only within decision groups, *i.e.,* node p sends or receives messages from node q only if $p, q \in g$ for some $g \in \mathcal{G}$. Thus, the identifiers used can be locally unique.

To achieve this goal, node p numbers each node q in each decision group $g \in \mathcal{G}_p$, sequentially. Notice that the same node q might "receive" different numbers in different decision groups that p participates in. Thus, we can define Num(p, g, q) to be the number p assigns to q for the decision group $g \in \mathcal{G}_p$. Notice that for an ℓ-lightweight entwined structure, Num$(p, *, *)$ requires polynomial space in $|\Gamma(p)|$. Each node z holds Num$(p, g, *)$ for all $g \in \mathcal{G}_z \bigcap \mathcal{G}_p$, along with a mapping between Num(p, g, q) and Num(z, g, q). The memory footprint of this mapping is again polynomial in $|\Gamma(z)|$ (for ℓ-lightweight entwined structures).

In addition, each node p numbers the decision groups it is a member of: let Indx(p, g) be the "number" of $g \in \mathcal{G}_p$ according to p's numbering. Any node z (such that $\mathcal{G}_p \bigcap \mathcal{G}_z \neq \phi$) holds a mapping between Indx(p, g) and Indx(z, g), for all $g \in \mathcal{G}_p \bigcap \mathcal{G}_z$. Notice that Indx$(p, *)$ is polynomial in $|\Gamma(p)|$, and the mapping requires memory space of size polynomial in $max\{|\Gamma(p)|, |\Gamma(z)|\}$. For ℓ-lightweight entwined structures, the distance between p and z is $\leq \ell$, thus $max\{|\Gamma(p)|, |\Gamma(z)|\} \leq \ell$-MaxDeg($p$), resulting in a memory space footprint (of all the above structures) that is polynomial in ℓ-MaxDeg(p) for any node p.

When node p wants to send node q's identifier to z regarding decision group g (notice that $p, q, z \in g$ and $g \in \mathcal{G}_p, \mathcal{G}_q, \mathcal{G}_z$), it sends "$(\textsc{Num}(p, g, q), \textsc{Indx}(p, g))$" and node z uses its mapping to calculate $\textsc{Indx}(z, g)$, from which node z can discern what g is, and use its $\textsc{Num}$ mapping to calculate $\textsc{Num}(z, g, q)$. Therefore, nodes can identify any node in their decision groups and can communicate these identities among themselves. Thus, "identities" can be uniquely used locally, with a low memory footprint. *i.e.,* the required memory space is polynomial in ℓ-$\textsc{MaxDeg}(p)$. Notice that the above structures are constructed before executing LocalByzCon, once the system designer knows the structure of $\mathcal{G}$.

4.2 Memory-Efficient Termination Detection

LocalByzCon as given in Figure 1 loops for $\Delta_{max} \cdot (2\mathcal{D}+1)$ rounds. Therefore, for $\mathcal{D}$ that depends on n, the counter of the loop will require too much memory. From the analysis of the execution of LocalByzCon it is clear that any node terminating after it ran for more than $\Delta_{max} \cdot (2\mathcal{D}+1)$ rounds, terminates with the same value. Thus, it is only important that all nodes eventually terminate, and that they do so after at least $\Delta_{max} \cdot (2\mathcal{D}+1)$ rounds; how can this be done without counting rounds? The following is an example of a solution requiring constant memory, using a simpler model.

Consider a synchronous network without any *Byzantine* nodes, where each node p has $poly(|\Gamma(p)|)$ memory. Given that $\mathcal{D}$ (the diameter of the network) is not constant, how can one count until $\mathcal{D}$ in such a network? Mark two nodes u, v as "special" nodes, such that the distance between u and v is $\mathcal{D}$ (clearly there exist u, v that satisfy this condition). When the algorithm starts, u floods the network with a *"start"* message. When v receives this message, it floods the network with an *"end"* message. When any node receives the *"end"* message, it knows that at least $\mathcal{D}$ rounds have passed, and it can therefore terminate.

Using the above example, we come back to our setting of entwined structures and f_ℓ-local *Byzantine* adversaries: consider two "special" decision groups g_1, g_2 from $\mathcal{G}$, such that the $\textsc{Time}_{dist}$ between g_1 and g_2 is $\mathcal{D}_{\text{TIME}}$ (see Remark 3). Each node p, in addition to its initial value v_p, has two more initial values v_p^1, v_p^2 which are both set to "1". All nodes in g_1 set v_p^1 :="0". Instead of executing a single LocalByzCon, each node executes 3 copies of LocalByzCon: one on v_p, one on v_p^1 (denoted $\textsc{LocalByzCon}_1$) and one on v_p^2 (denoted $\textsc{LocalByzCon}_2$). Only nodes in g_2 perform the following rule: once g_2's output in $\textsc{LocalByzCon}_1$ is "0", set v_p^2 := "0". Lastly, all nodes terminate one repetition after $Output_p$ of $\textsc{LocalByzCon}_2$ contains "0".

The analysis of this addition is simple: the value "0" in $\textsc{LocalByzCon}_1$ propagates throughout the network until it reaches g_2. Once it reaches g_2, the value "0" of $\textsc{LocalByzCon}_2$ propagates throughout the network, causing all nodes to terminate. Notice that before g_2 updates its input value of $\textsc{LocalByzCon}_2$, no correct node will have a value of "0" for $\textsc{LocalByzCon}_2$. Lastly, notice that at least $\frac{1}{2}\mathcal{D}_{\text{TIME}}$ rounds must pass before g_2 changes the input value to the third LocalByzCon (see Remark 3). Thus, if the second and third LocalByzCon are

executed "at half speed" (every round, the nodes wait one round), then all correct nodes terminate not before $\mathcal{D}_{\text{TIME}}$ rounds pass, and no later than $2\mathcal{D}_{\text{TIME}}$ rounds.

The above schema requires the same memory space as the "original" LocalByzCon (i.e. independent of n), up to a constant factor, while providing termination detection, as required.

The decision groups g_1, g_2 must be selected prior to LocalByzCon's execution. An alternative option is to select g_1 using some leader election algorithm, and then use an MST algorithm to find g_2. However, in addition to *Byzantine* tolerance, these algorithms' memory requirements must not depend on n, which means that global identifiers cannot be used. There is a plethora of research regarding leader election / spanning trees and their relation to memory space (see [1], [3], [4], [15]). However, as far as we know, there are no lower or upper bounds regarding the exact model this work operates in (e.g. local identities, but no global identities). Thus, it is an open question whether it is required to choose g_1, g_2 a priori, or if they can be chosen at runtime.

4.3 Complexity Analysis

Consider graph G with maximal degree d_{max}, and an ℓ-lightweight entwined structure $\mathcal{G}$ (with diameter $\mathcal{D}$) that covers G. Denote $g_{max} := argmax_{g_i \in \mathcal{G}}\{|g_i|\}$, since $\mathcal{G}$ is ℓ-lightweight, g_{max} is polynomial in d_{max}, and $|\mathcal{G}| = n \cdot poly(d_{max})$. Therefore, by Section 3.5, LocalByzCon's time complexity is $O(\mathcal{D}) \cdot poly(d_{max})$, and its message complexity is $O(\mathcal{D}) \cdot n \cdot poly(d_{max})$.

From the above, if G's maximal degree is independent of n, and if $\mathcal{D} = O(\log n)$, then the running time of LocalByzCon is $O(\log n)$ and its message complexity is $O(n \log n)$, while using $O(1)$ space. (Section 5 shows entwined structures that reach these values). This achieves an exponential improvement on the time and message complexity of *Byzantine* consensus in the "global" model, which are $O(n)$ and $O(n^2)$ respectively. [3]

5 Constructing 1-Lightweight Entwined Structures

In this section we present a family of graphs for which 1-lightweight entwined structures exist. The family of graphs is given in a constructive way: for each graph $G' = (V', E')$ (where $|V'| = n'$) and for any value of f_ℓ ($f_\ell << n'$) we construct a graph $G = (V, E)$ ($|V| = n$) with a 1-lightweight f_ℓ-entwined structure $\mathcal{G}$. Our construction achieves $|\mathcal{G}| = O(n)$ and $|g|$ is small for all $g \in \mathcal{G}$, thus ensuring that $\mathcal{G}$ is indeed 1-lightweight. Furthermore, G's diameter and maximal degree are a function of those of G's, thus ensuring that for G' with bounded degree and logarithmic diameter, G has similar properties.

First we show how to construct an entwined structure, given a graph with "ring topology". Then we show how "to combine" two graphs with ring topologies. Lastly, for every graph G', we create a graph G that "blows up" each node p

[3] In fact, known algorithms that use the transformation in [6] to operate in not-fully-connected graphs might even require $O(n^3)$ messages.

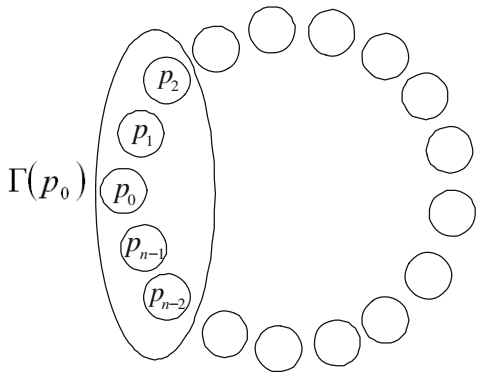

Fig. 2. p_0's neighbors in an extended ring topology of order 2

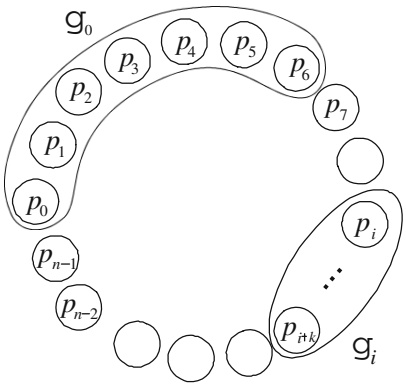

Fig. 3. g_0 for $f_\ell = 1$ and $k = 3(f_\ell + 1) = 6$

in G' to be a ring (containing $O(|\Gamma(p)|)$ nodes), and then combines the different rings of G'.

5.1 A Simple "Ring" Entwined Structure

This section presents a simple construction of an entwined structure $\mathcal{G}$ for a ring topology network. The constructed $\mathcal{G}$ has $|g_{max}|$ constant (independent of n), but a diameter of $O(n)$.

Definition 16. *A graph $G = (V, E)$ is said to have an* **extended ring topology** *of order k, if $E = \bigcup_{p_i \in V} \bigcup_{1 \leq j \leq k} \{(p_i, p_{i+j})\}$ (addition is done modulo n).*

Informally, an extended ring topology of order k means that each node is connected to the k neighbors "ahead" of it in the ring; since G is bi-directional, each node is also connected to the k neighbors "behind" it (see Figure 2 for an example). Notice that a "regular" ring topology is actually an extended ring topology of order 1.

Consider a graph $G = (V, E)$ with extended ring topology of order $k = 3 \cdot (f_\ell + 1)$. Define $\mathcal{G}$ as follows: $g_i := \{p_i, p_{i+1}, p_{i+2}, \ldots, p_{i+k}\}$ (see Figure 3).

Claim. Any $g \in \mathcal{G}$ is af_ℓ-decision group.

Claim. For any i, g_i and g_{i+1} are f_ℓ-connected.

Lemma 5. *$\mathcal{G}$ is an f_ℓ-entwined structure with diameter at most n, and $|g| = 3 \cdot f_\ell + 4$ for any $g \in \mathcal{G}$.*

Proof. By definition of $g_i \in \mathcal{G}$, $|g_i| = k + 1 = 3 \cdot f_\ell + 4$. Let $g_i, g_j \in \mathcal{G}$ be some decision groups. $0 \leq i, j \leq n - 1$ and either $i \geq j$ or $j \geq i$. Assume w.l.o.g. that $j \geq i$. Consider the list $C := g_i, g_{i+1}, \ldots, g_j$. By the previous claim, g_i and g_{i+1} are f_ℓ-connected, and therefore C is a value chain. Thus, there exists a f_ℓ-value chain between any two decision groups in $\mathcal{G}$ of length at most n. Thus, $\mathcal{G}$ is an f_ℓ-entwined structure with diameter at most n. □

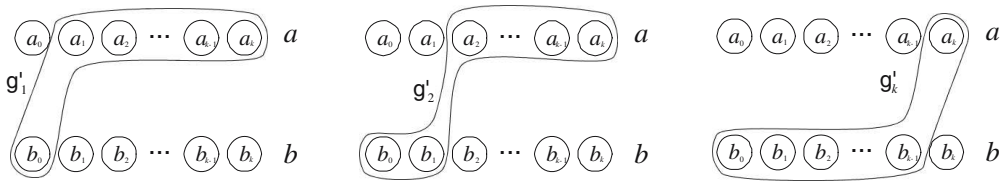

Fig. 4. The original decision groups a, b and the constructed decision groups g'_1, g'_2 and g'_k

5.2 Connecting Rings Together

Consider two extended ring topology graphs G_1, G_2 and their respective f_ℓ-entwined structures $\mathcal{G}_1, \mathcal{G}_2$ (as built in the previous subsection). Let $a \in \mathcal{G}_1$ and $b \in \mathcal{G}_2$ be 2 decision groups that are fully connected and are of the same size, e.g. $|a| = |b| = k + 1$.

Mark by a_i the nodes in a and by b_i the nodes in b (for $0 \leq i \leq k$). Define $g'_j := \{a_j, a_{j+1}, \ldots, a_k, b_0, \ldots, b_{j-1}\}$ for $1 \leq j \leq k$; see Figure 4 for an example. Add edges such that each g'_j is fully connected. Notice that $|g'_j| = k+1$ and that for $1 \leq j < k$, it holds that g'_j and g'_{j+1} are f_ℓ-connected. In addition a and g'_0 are f_ℓ-connected, as well as b and g'_k.

Take $\mathcal{G}' = \mathcal{G}_1 \bigcup \mathcal{G}_2 \bigcup \{g'_j\}$. $\mathcal{G}'$ is a f_ℓ-entwined structure over the graph that is the union of G_1, G_2 with edges that are induced by the different g'_js. Notice that the diameter of $\mathcal{G}'$ is the diameter of $\mathcal{G}_1$ plus the diameter of $\mathcal{G}_2$ plus $k + 1$ (the length of the distance between a and b). Furthermore, $|\mathcal{G}'| = |\mathcal{G}_1| + |\mathcal{G}_2| + k$. In addition every node $p \in a \bigcup b$ participates in no more than k additional decision groups.

5.3 General Entwined Structure Construction

Let G' be any graph on n' nodes, and let d_v be the degree of node v; denote $d_{max} := max_{v \in G'} d_v$. Consider a graph G_v per node v with $d_v \cdot k$ nodes, such that G_v has an extended ring topology of order k. Consider the "ring" entwined structure constructed in the previous sections $\mathcal{G}_v$ with decision groups denoted as $g_1(v), g_2(v), \cdots$. Notice that $g_i(v) \cap g_{i+k}(v) = \phi$ (see Figure 5). For each node $v \in G'$, consider the list of its neighbors $\Gamma(v) = v_1, v_2, \ldots, v_{d_v}$, and define a function $N(u, v)$ that returns the index of v as a neighbor of u; *i.e.,* $N(v, v_i) = i$.

Using the operation defined in the previous subsection: for any edge (u, v) in G', "connect" the rings in the following way: $g_{N(u,v)*k}(u)$ with $g_{N(v,u)*k}(v)$. That is, let v be the ith neighbor of u ($N(u, v) = i$ or $u_i = v$), and u be the jth neighbor of v ($N(v, u) = j$, $v_j = u$) (see Figure 6); the following two decision groups are to be combined: $g_{i*k}(u)$ with $g_{j*k}(v)$. Denote the new decision groups created due to each such "connection" as $\mathcal{G}_{u,v}$.

Consider G to be the union of all G_v (both nodes and edges), and let $\mathcal{G}$ be the union of the entwined structures. That is, $\mathcal{G} = (\bigcup_{v \in G'} \mathcal{G}_v) \cup (\bigcup_{u,v \in G'} \mathcal{G}_{u,v})$.

Claim. Let $g \in \mathcal{G}_v$ and $g' \in \mathcal{G}_u$ such that (v, u) is an edge in G', then there is a f_ℓ-value-chain between g and g' of length $\leq (d_{max} + 1) \cdot k$.

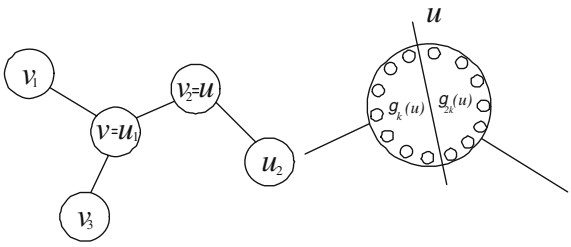

Fig. 5. A graph with 5 nodes, and node u's "ring" $\mathcal{G}_u$ and 2 decision groups $g_k(u), g_{2k}(u)$

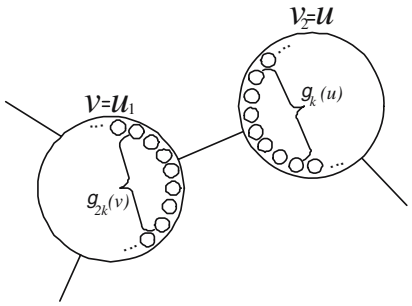

Fig. 6. v, u connecting, combining $g_{2k}(v), g_k(u)$

Claim. $\mathcal{G}$ is a f_ℓ-entwined structure with diameter $\mathcal{D} \leq D_{G'} \cdot (d_{max} + 1) \cdot k + 2k$, where $D_{G'}$ is the diameter of G'.

Theorem 3. *$\mathcal{G}$ is a 1-lightweight f_ℓ-entwined structure.*

5.4 Analysis

The above construction shows that for any graph G' there is a graph G and an f_ℓ-entwined structure $\mathcal{G}$ (covering G) s.t. the diameter of $\mathcal{G}$ is linear in the diameter of G', the maximal degree of G', and in k. Thus, by taking G' to be any graph with constant degree and logarithmic diameter, we have that the diameter of $\mathcal{G}$ is $O(D_{G'} \cdot k)$. In other words, $\mathcal{D} = O(k \cdot logn)$; by taking k to be constant as well (this means that the graph G has a constant degree), we have that $\mathcal{D} = O(logn)$. Therefore, the above construction produces graphs and their respective entwined structures, such that the diameter is logarithmic and the degree is constant, fulfilling the promise of Section 4.3.

6 Conclusion and Future Work

A sufficient condition for solving *Byzantine* consensus in the presence of a localized *Byzantine* adversary was given. Furthermore, for a family of graphs it was shown how to solve the *Byzantine* consensus problem using memory space that is constant (independent of the size of the network).

We consider few points for future research: first, find tighter bounds for when *Byzantine* consensus can be solved on a graph G. Second, allow for dynamic changes in the system, such as nodes joining or leaving or even constructing the entwined structure on-the-fly. Third, adapt the entwined structure-construction such that if some portion of the network does not uphold the required *Byzantine*-to-correct threshold, then the rest of the network may still reach consensus. Lastly, it would be interesting to find other constructions of entwined structures and see what additional types of graphs can solve *Byzantine* consensus.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments.

References

1. Afek, Y., Stupp, G.: Optimal time-space tradeoff for shared memory leader election. J. Algorithms 25(1), 95–117 (1997)
2. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics. John Wiley & Sons, Chichester (2004)
3. Beauquier, J., Gradinariu, M., Johnen, C.: Memory space requirements for self-stabilizing leader election protocols. In: PODC 1999: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing, pp. 199–207. ACM, New York (1999)
4. Beauquier, J., Gradinariu, M., Johnen, C.: Randomized self-stabilizing and space optimal leader election under arbitrary scheduler on rings. Distributed Computing 20(1), 75–93 (2007)
5. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. 20(4), 398–461 (2002)
6. Dolev, D.: The byzantine generals strike again. Journal of Algorithms 3, 14–30 (1982)
7. Dolev, D., Reischuk, R.: Bounds on information exchange for byzantine agreement. J. ACM 32(1), 191–204 (1985)
8. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
9. Dolev, S., Gouda, M.G., Schneider, M.: Memory requirements for silent stabilization. In: PODC 1996: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing, pp. 27–34. ACM, New York (1996)
10. Koo, C.-Y.: Broadcast in radio networks tolerating byzantine adversarial behavior. In: PODC 2004: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing, pp. 275–282. ACM, New York (2004)
11. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Transactions on Programming Languages and Systems 4(3), 301–382 (1982)
12. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
13. Pelc, A., Peleg, D.: Broadcasting with locally bounded byzantine faults. Inf. Process. Lett. 93(3), 109–115 (2005)
14. Toueg, S., Perry, K.J., Srikanth, T.K.: Fast distributed agreement. SIAM Journal on Computing 16(3), 445–457 (1987)
15. Yamashita, M., Kameda, T.: Computing on anonymous networks. i. characterizing the solvable cases. Parallel and Distributed Systems, IEEE Transactions 7(1), 69–89 (1996)

Chapter 6

Conclusions and Discussion

The above thesis shows several fault tolerant building blocks: self-stabilizing Byzantine tolerant clock synchronization, self-stabilizing Byzantine agreement and a self-stabilizing Byzantine tolerant shared coin. The importance of fault tolerant building blocks is self evident. For example, by constructing a self-stabilizing Byzantine tolerant shared coin, another tool is available for system designers that wish to add self-stabilization to randomized Byzantine protocols.

In order to make building blocks useful, it is important to ensure they are efficient and that they operate in as many models as possible. The first result (Chapter 2) solves the self-stabilizing Byzantine tolerant clock synchronization problem in a bounded-delay model. This solution extends ideas from synchronous solutions into the bounded-delay model, thus expanding the range of models in which a self-stabilizing Byzantine clock can be employed.

The second result (Chapter 3) improves the previous deterministic linear convergence time (of self-stabilizing Byzantine tolerant digital clock) to an expected constant convergence time. Thus making it much more efficient and accessible. It would be interesting to try and make the same transition from ideas in the synchronous model to the bounded delay model (as done in Chapter 2) regarding the randomized solution in Chapter 3. Such a transition would (hopefully) produce a self-stabilizing Byzantine tolerant clock algorithm for the bounded-delay model with expected constant convergence time. As part of the randomized solution of Chapter 3 another synchronous building is presented: a self-stabilizing Byzantine tolerant shared coin.

The first two results as well as the last one give upper bounds, *i.e.,* specific algorithms that solve different problems. However, the third result proves

different lower bounds. Specifically, it shows that when considering a stable agreement (*i.e.,* one in which the output does not change over time) the Byzantine adversary has the ability to delay the stability of the output indefinitely. That is, the adversary can disrupt the output stability at least once, and can delay the disruption for as long as it wants.

Lastly, in Chapter 5 the consensus problem is considered, against a local Byzantine adversary. One of the main motivations of the choice of adversary is to improve the message and round complexity. In the standard Byzantine model it is not possible to solve consensus within less than a linear number of rounds or in less than square number of messages. Under the local Byzantine adversary, Chapter 5 shows a solution improving both message and round bounds. The solution is non-self-stabilizing. As was previously shown, consensus and clock synchronization are closely related problems. It would be interesting to build upon the consensus solution to achieve a self-stabilizing Byzantine tolerant clock which operates against a local adversary. Such a solution would (hopefully) have low message complexity and low round complexity while solving clock synchronization in a self-stabilizing, Byzantine tolerant and realistic manner.

An important scheme used thoroughly in this thesis, is that of pipelining. Pipelining is a scheme used to take an algorithm $\mathcal{A}$ which is Byzantine tolerant but is not self-stabilizing, and construct an algorithm $\mathcal{B}$ that is both Byzantine tolerant and self-stabilizing. Specifically, in Chapter 3, pipelining is used to create a self-stabilizing Byzantine tolerant shared coin algorithm. Chapter 2 is based on a synchronous algorithm with heavily relies on pipelining, and thus the bounded-delay solution's intuition is based on pipelining. It is interesting to see if there are any other possible usages of pipelining in the world of self-stabilization and Byzantine tolerance.

Bibliography

[1] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30, New York, NY, USA, 1983. ACM.

[2] A. Daliot and D. Dolev. Self-stabilization of byzantine protocols. In *In Proc. of the 7th Symposium on Self-Stabilizing Systems (SSS'05)*, Barcelona, Spain, Oct 2005.

[3] A. Daliot and D. Dolev. Self-stabilizing byzantine pulse synchronization. Technical report, Cornell ArXiv, Aug 2005. url: http://arxiv.org/abs/cs.DC/0608092.

[4] A. Daliot and D. Dolev. Self-stabilizing byzantine agreement. In *In Proc. of the Twenty-fifth ACM Symposium on Principles of Distributed Computing (PODC'06)*, Denver, Colorado, Jul 2006.

[5] A. Daliot, D. Dolev, and H. Parnas. Linear time byzantine self-stabilizing clock synchronization. In *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.

[6] W. Dijkstra. Self-stabilization in spite of distributed control. *Commun. of the ACM*, 17:643–644, 1974.

[7] D. Dolev. The byzantine generals strike again. *Journal of Algorithms*, 3:14–30, 1982.

[8] D. Dolev and E. N. Hoch. On self-stabilizing synchronous actions despite byzantine attacks. In *In Proc. the 21st Int. Symposium on Distributed Computing (DISC'07)*, Lemesos, Cyprus, Sep. 2007.

[9] S. Dolev. *Self-Stabilization*. The MIT Press, 2000.

[10] S. Dolev and J. L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults. *Journal of the ACM*, 51(5):780–799, 2004.

[11] Paul Feldman and Silvio Micali. An optimal probabilistic algorithm for synchronous byzantine agreement. In *ICALP '89: Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, pages 341–378, London, UK, 1989. Springer-Verlag.

[12] M. J. Fischer, N. A. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1:26–39, 1986.

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

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

[15] Chiu-Yuen Koo. Broadcast in radio networks tolerating byzantine adversarial behavior. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 275–282, New York, NY, USA, 2004. ACM.

[16] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–301, 1982.

[17] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[18] T. Masuzawa and 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.

[19] Toshimitsu Masuzawa and Sébastien Tixeuil. Bounding the impact of unbounded attacks in stabilization. In *SSS*, pages 440–453, 2006.

[20] Michael B. Or, Elan Pavlov, and Vinod Vaikuntanathan. Byzantine agreement in the full-information model in o(log n) rounds. In *STOC '06: Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 179–186, New York, NY, USA, 2006. ACM.

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

[22] Andrzej Pelc and David Peleg. Broadcasting with locally bounded byzantine faults. *Inf. Process. Lett.*, 93(3):109–115, 2005.

[23] M. Rabin. Randomized Byzantine generals. *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 403–409, 1983.

[24] Yusuke Sakurai, Fukuhito Ooshita, and Toshimitsu Masuzawa. A self-stabilizing link-coloring protocol resilient to byzantine faults in tree networks. In *OPODIS*, 2004.