Self-stabilizing Byzantine Pulse and Clock Synchronization

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

by

Ezra N. Hoch

Supervised by

Prof. Danny Dolev

School of Engineering and Computer Science
The Hebrew University of Jerusalem
Israel

March 28, 2007

Acknowledgments

First, I would like to gratefully thank my advisor, Prof. Danny Dolev, for his supporting guidance, inspiring ideas, everlasting patience and for capturing my imagination.

Second, I would like to thank Ariel Daliot (now Dr. Ariel Daliot), for being there to pitch ideas at and for his enthusiasm, some of which he passed on to me.

Third, I would like to thank the DANSS team for their support and friendliness.

Lastly, I would like to thank my friends and classmates for staying my friends throughout the period of my study.

Abstract

This thesis presents a scheme that achieves self-stabilizing Byzantine digital clock synchronization assuming a "synchronous" system. This synchronous system is established by the assumption of a common external "beat" delivered with a regularity in the order of the network message delay, thus enabling the nodes to execute in lock-step. The system can be subjected to severe transient failures with a permanent presence of Byzantine nodes. The presented algorithms guarantee eventually synchronized digital clock counters, i.e. common increasing integer counters associated with each beat.

Two algorithms for achieving this synchronization are given. The first one, produces digital clock synchronization directly. The second one, produces an underlaying pulse which can be used to create clock synchronization, or can be used for other synchronization goals.

Using the digital clock synchronization, one can go on to achieve regular clock synchronization, progressing at real-time rate and with high granularity. In addition, a general Byzantine stabilizer is shown, based on the pulse synchronization algorithm.

This thesis shows the first algorithms to achieve deterministic linear convergence time, supporting $f < \frac{n}{3}$ Byzantine nodes. In addition, it does not require the use of local physical timers. Moreover, it is the first to show a self-stabilizing protocol that overcomes Byzantine faults and operates in a network that is not fully connected.

Table of Contents

List of Figures

Chapter 1

Introduction

Most distributed tasks require some sort of synchronization. Clock synchronization is a very straightforward and intuitive tool for supplying this. PULSE synchronization can be used as an underlying building block to achieve clock synchronization, as well as solving other synchronization problems; in a sense, PULSE synchronization is a more fundamental synchronization problem.

It thus makes sense to require such an underlying synchronization mechanism to be highly fault-tolerant.

1.1 Two Synchronization Mechanisms

This thesis presents two such mechanisms: digital clock synchronization and PULSE synchronization. Both algorithms are self-stabilizing and are tolerant to permanent presence of *Byzantine* faults. That is, they attain synchronization, once lost, while containing the influence of the permanent presence of faulty nodes.

Consider a system in which the nodes execute in lock-step by regularly receiving a common "pulse" or "tick" or "beat". The digital clock synchronization problem is to ensure that eventually all the correct nodes hold the same value of 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" following each beat. In this thesis, the terms "clock" and "digital clock" are used interchangeably.

The pulse synchronization problem is to agree on some "special beats" that are $Cycle$ beats apart. More specifically, the PULSE synchronization problem is to ensure that eventually all correct nodes PULSE together, and as long as enough nodes remain

correct, they continue to PULSE together *Cycle* beats apart. For example, given $Cycle = 7$ we would like all correct nodes, that may start at arbitrary initial states, to eventually PULSE together every 7 beats, and continue so as long as there are enough correct nodes.

We will use the "beat" notation for the "global" signal received, and "pulse" for the "special beats" agreed upon.

The motivation for such a PULSE synchronization algorithm is to allow for "long" algorithms enough time to execute. For example, setting *Cycle* to be the worst-case execution-time for terminating *Byzantine* agreement, one can start an agreement on the next clock value each PULSE[1]; and thus solve the digital clock synchronization problem.

In both problems, the global beat system provides some measure of synchronization. For example, given a global beat system with beat interval at least as long as the worst-case execution-time for terminating *Byzantine* agreement, the digital clock synchronization problem is solved by initiating a *Byzantine* agreement on the next clock value, each time a beat is received. Under the same assumption, the PULSE synchronization problem is solved by initiating a *Byzantine* agreement on the next time when the nodes should PULSE, each time a beat is received.

The crux of both problems is to achieve synchronization when it is not given by the global beat system; that is, when the beat interval length is in the order of communication's end-to-end delay. Since in that scenario the global beat system does not provide - by itself - enough synchronization, and a more complex algorithm is required to exert the required synchronization. The main contribution of the thesis is achieving exactly that.

1.2 Contribution

Two different synchronization problems are presented and solved herein. Both are highly fault tolerant. That is, self-stabilizing and tolerant to permanent presence of *Byzantine* faults.

The first, the digital clock synchronization problem, was presented before in [2]. In [2] a randomized solution was given, with expected exponential convergence time. Here we give a solution that converges deterministically in linear time. The main

[1] See [1] for such a self-stabilizing *Byzantine* clock synchronization algorithm, which executes on top of a self-stabilizing *Byzantine* pulse-synchronization primitive in a semi-synchronous network.

drawback of the presented solution is that it supports only $f < \frac{n}{4}$, as opposed to the solution of [2], which supports $f < \frac{n}{3}$.

To overcome this weakness, the PULSEing problem is defined and solved. Its solution is tolerant to up to $f < \frac{n}{3}$ *Byzantine* nodes, and converges deterministically in linear time. However, it does not solve the digital clock synchronization problem directly, but rather requires an additional step to convert it to a digital clock synchronization solution. Luckily, this additional step does not change the convergence time or the *Byzantine* fault tolerance level.

In addition, the PULSE solution algorithm is the first one in the presented model that does not require the nodes to be fully connected to each other; it only requires that there are $2 \cdot f + 1$ distinct routes between any two correct nodes.

1.3 Thesis Outline

The thesis is organized as follows: a short survey of related work is given in Chapter 2. Chapter 3 defines the model and both of the clock/pulse synchronization problems. Chapter 4 discusses in depth the digital clock synchronization problem, and provides a solution to it. Chapter 5 discusses in detail the PULSE synchronization problem, and provides a solution to it. Lastly, Chapter 6 contains conclusions and future work.

Chapter 2

Related Work

Digital clock synchronization and pulse synchronization are related problems; and in the model that this thesis operates in they are equivalent (see Section 3.4). When considering previous work, it can be divided into 3 categories: clock synchronization, pulse synchronization and combining *Byzantine* faults with self-stabilization. Pulse synchronization has been discussed only in the context of combining the two fault models, as opposed to clock synchronization which has been discussed in each of the fault models independently.

2.1 Digital Clock Synchronization

The presented self-stabilizing *Byzantine* clock synchronization algorithms assume that common beats are received synchronously (simultaneously) and in the order of the message delay apart. The clocks progress at real-time rate. Thus, when the clocks are synchronized, in-spite of permanent *Byzantine* faults, the clocks may accurately estimate real-time.[1] Following transient failures, and with on-going *Byzantine* faults, the clocks will synchronize within a finite time and will progress at real-time rate, although the actual clock-reading values might not be directly correlated to real-time. Many applications utilizing the synchronization of clocks do not really require the exact real-time notion (see [3]). In such applications, agreeing on a common clock reading is sufficient as long as the clocks progress within a linear envelope of any real-time interval. An additional advantage of the current solution is that it can be

[1] All the arguments apply also to the case where there is a small bounded drift among correct clocks.

implemented without the use of local physical timers at the nodes. Local timers are only needed to achieve a high granularity of the synchronized clocks.

Clock synchronization in a similar model has earlier been denoted as "digital clock synchronization" ([4, 5, 6, 7]) or "synchronization of phase-clocks" ([8]), in which the goal is to agree on continuously incrementing counters associated with the beats. The convergence time in those papers is not linear, whereas in the current solution it is linear.

The additional requirement of tolerating permanent *Byzantine* faults poses a special challenge for designing self-stabilizing distributed algorithms due to the capability of malicious nodes to hamper stabilization. This difficulty may be indicated by the remarkably few algorithms resilient to both fault models (see [9] for a short review). The digital clock synchronization algorithms in [2] are, to the best of our knowledge, the first self-stabilizing algorithms that are tolerant to *Byzantine* faults. The randomized algorithm, presented in [2], operating in the same model as in the current thesis, converges in expected exponential time.

In [1] it was previously presented a self-stabilizing *Byzantine* clock synchronization algorithm, which converges in linear time and does not assume a synchronous system. That algorithm executes on top of an internal pulse synchronization primitive with intervals that allow to execute *Byzantine* agreement in between. The solution presented in the current thesis also converges in linear time and only assumes that the (external synchronously received) beats are on the order of the message delay apart. The current solution is simpler, and takes advantage of the stronger model, as opposed to [1], which would not improve its precision if executed in the current model.

Remark 2.1.1. *Chapter 4 have been previously publish in [10].*

2.2 Pulse Synchronization

The first PULSEing algorithm was given in [11], in a self-stabilizing and *Byzantine* tolerant model. It was later shown that a PULSEing algorithm can be used as an underlying layer to achieve clock synchronization ([1]), token circulation ([12]) and a general stabilizer ([9]). All of these results are given in a self-stabilizing, *Byzantine* tolerant manner, in a model in which message delivery time is bounded (but there is no global beat system). This gives the motivation for producing robust and efficient

PULSEing algorithms, as they can be used to improve the robustness of a variety of applications.

In [11] and [13] the presented pulse synchronization procedures do not assume any sort of prior synchronization such as common beats. The former is biologically inspired and the latter utilizes a self-stabilizing Byzantine agreement algorithm developed in [14]. Both of these pulse synchronization algorithms are complicated and have complicated proofs, while the current pulse solution is achieved in a relatively straightforward manner and its proofs are simpler. Due to the relative simplicity of the algorithm, formal verification methods, as were used in [15], can potentially be used to formally verify the correctness of the proposed algorithms.

2.3 Self-stabilization and *Byzantine* Faults

Several fault tolerant stabilizers exist (see [16], [17] and [18]) with varying requirement and features (such as local containment of faults). In [9], it was shown that PULSE synchronization can be used to create a generalized stabilizer. However, in [9] the stabilizer is complex, and can stabilize a narrow class of algorithms. In Section 5.9.2 we show a simpler stabilizer, which can stabilize a wider range of algorithms.

Chapter 3

Model and Problem Statement

This chapter contains the formal definition of the model in which the different synchronization problems are defined and solved. In addition, the formal definitions of the Digital Clock and Pulse synchronization problems are given.

3.1 Model

Consider a fully connected network of n nodes. All the nodes are assumed to have access to a "global beat system" that provides "beats" with regular intervals. The communication network and all the nodes may be subject to severe transient failures, which might eventually leave the system in an arbitrary state.

We say that a node is *Byzantine* if it does not follow the instructed algorithm and *non-Byzantine* otherwise. Thus, any node whose failure does not allow it to exactly follow the algorithm as instructed is considered *Byzantine*, even if it does not behave fully maliciously. A non-*Byzantine* node will be called *non-faulty*. In the following discussion f will denote the upper bound on the number of permanent *Byzantine* nodes. The Digital Clock synchronization solution supports $f < \frac{n}{4}$; the Pulse synchronization solution supports $f < \frac{n}{3}$.

Assume that the network has bounded time on message delivery when it behaves coherently. Nodes are instructed to send their messages immediately after the occurrence of a beat from the global beat system. In addition, message delivery and the processing involved can be completed between two consecutive global beats, by any node that is non-faulty. More specifically, the time required for message delivery and message processing is called a *round*, and we assume that the time interval between

global beats is greater than and in the order of such a round. Due to transient faults, different nodes might not agree on the current beat/round number. Hence, we use a notion of an external beat number r, which the nodes are not aware of, but will simplify the presentations and discussions.

At times of transient failures there can be any number of concurrent *Byzantine* faulty nodes; the turnover rate between faulty and non-faulty behavior of nodes can be arbitrarily large and the communication network may also behave arbitrarily. Eventually the system behaves coherently again. At such case a non-faulty node may still find itself in an arbitrary state.

Definition 3.1.1. *The system is* coherent *if there are at most f Byzantine nodes, and each message sent at a beat to a non-faulty destination arrives and is processed at its destination before the next beat.*

Remark 3.1.2. *The definition of* coherent *depends on the value of f and hence is different for the different synchronization problems. It should be clear from the context which value of f is relevant.*
This remark holds for the following definitions as well.

Since a non-faulty node may find itself in an arbitrary state, there should be some time of continues non-faulty operation before it can be considered correct.

Definition 3.1.3. *A non-faulty node is considered* correct *only if it remains non-faulty for Δ_{node} rounds during which the system is coherent.*[1]

The algorithm parameters n, f, as well as the node's id are fixed constants and thus are considered part of the incorruptible correct code at the node. Thus, it is assumed that non-faulty nodes do not hold arbitrary values of these constants.

3.2 The Digital Clock Synchronization Problem

The Digital Clock synchronization problem consists of synchronizing integer values among the different nodes, in such a way that they all agree on the same value, and increase it by 1 each beat. More formally:

Denote by $DigiClock_p(r)$ the value of the digital clock at node p at beat r. We say that the system is in a *synchronized_state* if for all correct nodes the value of their $DigiClock$ is identical.

[1] The assumed bound on the value of Δ_{node}, for each problem, will be defined later.

Definition 3.2.1. The digital-clock synchronization problem
Convergence: *Starting from an arbitrary system state, the system reaches a synchronized_state after a finite time.*

Closure: *If at beat r the system is in a synchronized_state then for every r', $r' \geq r$,*

1. *the system is in a synchronized_state at beat r'; and*
2. *$DigiClock(r') = (DigiClock(r) + r' - r) \pmod{\text{max-clock}}$,[2] at each correct node.*

Remark 3.2.2. *The above problem statement implicitly requires that the precision of the clock synchronization is zero. That is, an algorithm that solves the above problem would have all correct nodes agree on the same value of DigiClock at the same time.*

Previous works (such as [1]) that operate in a different model (without a distributed external "beat" system) would not achieve precision zero, even if executed in the current model. Hence, the proposed algorithm herein fully utilized the strength of the "global beat system" model.

3.3 The Pulse Synchronization Problem

The Pulse synchronization problem consists of having all correct nodes "pulse" together every predefined period of time. The following is a formal definition.

We say that a system is $[\phi, \psi]$-PULSING if all correct nodes PULSE together in the following pattern: ϕ consecutive beats of PULSEs followed by ψ consecutive beats of non-PULSE. That is, the system has a *Cycle* of length $\phi + \psi$ beats, out of which only the first ϕ beats are PULSEs. More formally, denote by $pulsed_p(r) = True$ if p PULSEd on beat r and $pulsed_p(r) = False$, otherwise.

Definition 3.3.1. *A system is $[\phi, \psi]$-PULSING in the beat interval $[r_1, r_2]$ if there exists some $0 \leq k < \phi + \psi$, such that for every correct node p, and for every beat $r \in [r_1, r_2]$, it holds that:*

1. *$pulsed_p(r) = True$, in case $0 \leq r - k \pmod{\phi + \psi} < \phi$; and*
2. *$pulsed_p(r) = False$ in case $\phi \leq r - k \pmod{\phi + \psi} < \phi + \psi$.*

[2] "*max-clock*" is the wrap around of the variable *DigiClock*. All increments to *DigiClock* in the rest of the thesis are assumed to be (mod *max-clock*).

(k denotes the offset, from r_1, of the first PULSE *in the pattern.)*

For example, consider "1" to represent a beat in which all correct nodes PULSE, and "0" to represent a beat in which all correct nodes do not PULSE. Using this notation, the following is a PULSEing pattern of a $[\phi, \psi]$-PULSING system.

$$[\ldots, \overbrace{1,1,\ldots,1}^{\phi \text{ beats}}, \overbrace{0,0,\ldots,0}^{\psi \text{ beats}}, \overbrace{1,1,\ldots,1}^{\phi \text{ beats}}, \overbrace{0,0,\ldots,0}^{\psi \text{ beats}}, \ldots]$$
$$\underbrace{\qquad\qquad}_{Cycle \text{ beats}} \qquad \underbrace{\qquad\qquad}_{Cycle \text{ beats}}$$

Definition 3.3.2. The PULSEing problem
Convergence: *Starting from an arbitrary state, the system becomes* $[\phi, \psi]$-PULSING *after a finite number of beats.*
Closure: *If the system is* $[\phi, \psi]$-PULSING *in the beat interval* $[r_1, r_2]$ *it is also* $[\phi, \psi]$-PULSING *in the interval* $[r_1, r_2 + 1]$.

Definition 3.3.3. *a* $[\phi, \psi]$-PULSER *is an algorithm* $\mathcal{A}$*, such that once the system is coherent (and stays so), it solves the* PULSE*ing problem.*

The objective is to develop an algorithm that PULSEs only once every $Cycle$.

NOTATION 3.3.1. *We denote a* $[1, \psi]$-PULSER *as* $[\psi + 1]$-PULSER.

Using the above notation, "1" for PULSEing and "0" for non-PULSEing, a $[Cycle]$-PULSER looks as follows:

$$[\ldots, \underbrace{1,0,0,\ldots,0}_{Cycle \text{ beats}}, \underbrace{1,0,0,\ldots,0}_{Cycle \text{ beats}}, \ldots]$$

The goal is to build a $[Cycle]$-PULSER for any $Cycle > 0$. That is, a self-stabilizing, *Byzantine* tolerant algorithm that eventually PULSEs every $Cycle$ beats.

3.4 Digital Clock vs. Pulse

This section shows that the digital clock synchronization problem is equivalent to the PULSEing problem. One direction of the equivalence is: given an algorithm that solves the digital clock synchronization problem, simply PULSE every time the $DigiClock$ variable is divisible by $Cycle$. This produces a $[Cycle]$-PULSER algorithm.

The other direction is a bit more complicated. Given a $[f+2]$-PULSER algorithm, every PULSE execute a *Byzantine* agreement on what the *DigiClock* value will be in the next PULSE. In addition, each beat *DigiClock* is increased by 1, and when the *Byzantine* agreement terminates, the *DigiClock* is set to the agreement value. This way, all nodes agree on the value of *DigiClock* and increase it by one at each beat.

In the following chapters, a direct solution to the digital clock problem is provided, along with an indirect solution (via PULSEing). The indirect solution has better "properties". However, it is important to consider the direct solution as well, as it provides insight into some synchronization complexities. This leads to a better understanding of the requirements of a highly fault tolerant synchronization mechanism.

Chapter 4

Digital Clock Synchronization

This chapter details a solution for the Digital Clock synchronization problem. The algorithm given in Section 4.4 supports up to $\frac{n}{4}$ *Byzantine* nodes, and all references to f in this chapter assume that $f < \frac{n}{4}$.

4.1 Chapter Outline

Section 4.2 defines the requirements of the $\mathcal{BC}$ protocol which is used as a sub-routine for the digital clock synchronization algorithm. Section 4.3 describes an implementation of the $\mathcal{BC}$ protocol. In Section 4.4 our solution for the digital clock problem is presented. Section 4.5 contains the proof of correctness of the algorithm presented in Section 4.4. The rest of the chapter is dedicated to complexity analysis of the algorithm, and a discussion of the results.

4.2 The Byzantine Consensus Protocol

The digital clock synchronization algorithm utilizes a *Byzantine* consensus protocol as a sub-routine. Denote this protocol by $\mathcal{BC}$. We require the regular three conditions of Consensus from $\mathcal{BC}$, and one additional fourth requirement. That is, in $\mathcal{BC}$ the following holds:

1. Agreement: All non-faulty nodes terminate $\mathcal{BC}$ with the same output value.
2. Validity: If all non-faulty nodes have the same initial value v, then the output value of all non-faulty nodes is v.

3. Termination: All non-faulty nodes terminate $\mathcal{BC}$ within Δ rounds.

4. Solidarity. If the non-faulty nodes agree on a value v, such that $v \neq \perp$ (where $\perp$ denotes a non-value), then there are at least $n - 2 \cdot f$ non-faulty nodes with initial value v.

Remark 4.2.1. *Note that for $n > 4f$ the "solidarity" requirement implies that if the* Byzantine *consensus is started with at most $\frac{n}{2}$ non-faulty nodes with the same value, then all non-faulty nodes terminate with the value $\perp$.*

As we commented above, since $\mathcal{BC}$ requires the nodes to maintain a consistent state throughout the protocol, a non-faulty node that has recently recovered from a transient fault cannot be considered correct. In the context of this chapter, a non-faulty node is considered *correct* once it remains non-faulty for at least $\Delta_{node} = \Delta + 1$ beats and as long as it continues to be non-faulty.

In Section 4.3 we discuss how a typical synchronous *Byzantine* consensus protocol can be used as such a $\mathcal{BC}$ protocol. The specific example we discuss has two early stopping features: First, termination is achieved within $2f + 4$ of our rounds. If the number of actual *Byzantine* nodes is $f' \leq f$ then termination is within $2f' + 6$ rounds. Second, if all non-faulty nodes have the same initial value, then termination is within 4 rounds.

The symbol Δ denotes the bound on the number of rounds it takes $\mathcal{BC}$ to terminate at all correct nodes. That is, if $\mathcal{BC}$ has some early stopping feature, we still wait until Δ rounds pass. This means that the early stopping may improve the message complexity, but not the time complexity. By using the protocols in Section 4.3, we can set $\Delta := 2f + 4$ rounds.

Remark 4.2.2. *The symbol Δ is also used in Chapter 5. Its value will be clear from the context.*

4.3 *Byzantine* Consensus ($\mathcal{BC}$) Implementation

We present a $\mathcal{BC}$ protocol, denoted Byz-Consensus, that has the four properties required by our algorithm. This is done by converting the *Byzantine* agreement protocol in [19] into a *Byzantine* consensus protocol. In the following sections we show that the converted protocol guarantees *agreement, termination, validity* and *solidarity*.

We start by presenting our $\mathcal{BC}$ protocol in Figure 4.1. In the following subsection we claim that the *broadcast* primitive continues to hold its properties in the new protocol. The full proofs are in Appendix 7.

4.3.1 The $\mathcal{BC}$ Protocol

The difference between a *Byzantine* agreement protocol and a *Byzantine* consensus protocol is that in agreement there is a *general* G with some initial value v, and all correct nodes need to agree on G's value. In consensus, there is no general, and every correct node has its own initial value, and all correct nodes need to agree on an output value. *Byzantine* consensus "can be seen" as an agreement problem where the *general* might have sent different values to different nodes, and all correct nodes need to agree on what value G has sent.

We use the *Byzantine* agreement protocol in [19], and alter it in the following way. We consider the general to be a virtual node I_0, that sends its value to all nodes in the first round. I_0 does not participate in the protocol, except for "supposedly" having "sent" its value in the first round. Other than this change, our protocol is almost identical to the protocol in [19]. The added change transforms the protocol in [19] to a *Byzantine* consensus protocol, and also ensures that *solidarity* holds. Note that I_0 is not considered a *Byzantine* node or a correct node, it has a special status of a "virtual" node. That is, neither n (the number of nodes in the system) nor f count I_0.

We achieve the *solidarity* requirement by sending the initial value of every node to all other nodes. We say that two messages "(v)" are *distinct* if they were sent by different nodes. An *echo* message is sent only if a correct node received $n-f$ distinct "(v)" messages. Hence, there were at least $n-2\cdot f$ nodes with v as their initial value. Clearly, no correct node will accept if not even a single correct node has sent an *echo* message. Therefore, if a correct node accepted some value v', then an *echo* message was sent by a correct node, and that means that at least $n-2\cdot f$ correct nodes had v' as their initial value.

The protocol we present here is round based, and each round consists of two phases. In Appendix 7, Subsection 7.2, we prove the following: *The "Agreement", "Validity", "Termination" and "Solidarity" conditions hold for Byz-Consensus.*

Protocol **Byz-Consensus**/* *executed at node p where m is the initial value* */

1. $broadcasters := \phi; v := \perp$;
2. phase = 1:
   send(m) to all nodes;
3. phase = 2:
   (a) **if received** $n - f$ distinct (v') messages by end of phase 1 **then**
   send(echo, I_0, v', 1) to all nodes;
   (b) **if received** $n - f$ distinct (echo, I_0, v', 1) messages by the end of phase 2 **then** $v = v'$;
4. **round** r **for** $r = 2$ **to** $r = f + 2$ **do**:
   (a) **if** $v \neq \perp$ **then**
   invoke **broadcast**(p, v, r);
   stop and **return**(v);
   (b) by the end of round r:
   if in round $r' \leq r$ **accepted**$(I_0, v', 1)$ and (q_i, v', i) for all $i, 2 \leq i \leq r$,
   where all q_i distinct **then**
   $v := v'$;
   (c) **if** $|broadcasters| < r - 1$ **then**
   stop and **return**(v);
5. **stop** and **return**(v);

Figure 4.1: The Byzantine Consensus algorithm

4.3.2 The *b*roadcast Primitive

We use the same *broadcast* primitive as in [19]. The *broadcast* primitive is provided here for convenience. It is almost an exact copy of the *broadcast* primitive in [19], with minor changes for readability.

A minor difference needs to be addressed in the *broadcast* primitive. Instead of allowing the acceptance of messages only from the set of nodes P, we also allow accepting a message from a single virtual node, named I_0. That is, a node may accept a message $(I_0, v, 1)$. However, since I_0 is not an actual node, it does not participate in sending messages or receiving them.

When stating (and proving) the properties of the *broadcast* primitive, we consider $I_0 \notin P$. We consider $n = |P|$ and f to be the number of *Byzantine* nodes **not** including I_0.

Note that an *init* message for I_0 is never sent, because I_0 is a virtual node, and

Broadcast Primitive /* *rules for broadcasting and accepting* (p, m, k) */

Round k:
  Phase $2k-1$:
    node p sends $(init, p, m, k)$ to all nodes.
  Phase $2k$:
    if received $(init, p, m, k)$ from p in phase $2k-1$
      and received only one $(init, p, \_, \_)$ message in all previous phases
    then send $(echo, p, m, k)$ to all;
    if received $(echo, p, m, k)$ from $\geq n-f$ distinct nodes in phase $2k$
    then *accept* (p, m, k);
Round $k+1$:
  Phase $2k+1$:
    if received $(echo, p, m, k)$ from $\geq n-2f$ distinct nodes in phase $2k$
    then send $(init', p, m, k)$ to all;
    if received $(init', p, m, k)$ from $\geq n-2f$ distinct nodes in phase $2k+1$
    then $broadcastuers := brodcasters \cup \{p\}$;
  Phase $2k+2$:
    if received $(init', p, m, k)$ from $\geq n-f$ distinct nodes in phase $2k+1$
    then send $(echo', p, m, k)$ to all;
    if received $(echo', p, m, k)$ from $\geq n-f$ distinct nodes in phase $2k+2$
    then *accept* (p, m, k);
Round $r \geq k+2$:
  Phase $2r-1, 2r$:
    if received $(echo', p, m, k)$ from $\geq n-2f$ distinct nodes in previous phases,
      and not sent $(echo', p, m, k)$
    then send $(echo', p, m, k)$ to all;
    if received $(echo', p, m, k)$ from $\geq n-f$ distinct nodes in previous phases
    then *accept* (p, m, k);

Figure 4.2: The *broadcast* primitive

its *init* message is simulated by the first phase of the Byz-Consensus protocol.

Also note that the messages sent in line 3.a and received in line 3.b are of the same format as messages sent by the *broadcast* primitive. That is, nodes may accept a message $(I_0, v, 1)$, "as if" I_0 actually sent it. Hence, we cannot simply use the original *broadcast* primitive as a "black box", even though it has not been changed. We are required to reprove the properties shown in [19].

The properties that hold regarding the *broadcast* primitive used in our protocol, are:

1. *Correctness*: If a correct node $p \in P$ executes broadcast(p, m, k) in round k, then every correct node accepts (p, m, k) in the same round.

2. *Unforgeability*: If a correct node $p \in P$ does not execute broadcast(p, m, k), then no correct node ever accepts (p, m, k).

3. *Relay*: If a correct node accepts (p, m, k) in round r for some $p \in P \cup \{I_0\}$, then every other correct node accepts (p, m, k) in round $r + 1$ or earlier.

4. *Detection of broadcasters*: If a correct node accepts (p, m, k) in round k or later, for some $p \in P \cup \{I_0\}$, then every correct node has $p \in broadcasters$ at the end of round $k + 1$.

The proofs of the above properties are in Appendix 7, Subsection 7.1.

4.4 Digital Clock Synchronization Algorithm

The following digital clock synchronization algorithm tolerates up to $f < \frac{n}{4}$ concurrent *Byzantine* faults. The objective is to have the digital clocks increment by "1" every beat and to achieve synchronization of these digital clocks.

4.4.1 Intuition for the Algorithm

The idea behind our algorithm is that each node runs many simultaneous *Byzantine* consensus protocols. In each round of the algorithm it executes a single round in each of the *Byzantine* consensus protocols, but each *Byzantine* consensus protocol instance is executed with a different round number. That is, if $\mathcal{BC}$ takes Δ rounds to terminate, then each node runs Δ concurrent instances of it, where, for the first one it executes the first round, for the second it executes the second round, and in general for the i^{th} $\mathcal{BC}$ protocol it executes the i^{th} round. We index a $\mathcal{BC}$ protocol by the number of rounds passed from its invocation. When the Δ^{th} $\mathcal{BC}$ protocol is completed, a new instance of $\mathcal{BC}$ protocol is initiated. This mechanism, of executing concurrently Δ $\mathcal{BC}$ protocols, allows the non-faulty nodes to agree on the clock values as of Δ rounds ago. The nodes use the consistency of these values as of Δ rounds ago together with the exchange of their current values to "tune" the future clock values.

4.4.2 Preliminaries

Given a *Byzantine* consensus protocol $\mathcal{BC}$, each node maintains the following variables and data structures:

1. *DigiClock* holds the beat counter value at the node.

2. *ClockVec* holds a vector containing the value of *DigiClock* that each node sent in the current round.

3. $DigiClock_{most}$ holds the value that appears at least $\frac{n}{2}+1$ times in *ClockVec,* if one exists.

4. $Agree[i]$ is the memory space of the i^{th} instance of $\mathcal{BC}$ protocol (the one initialized i rounds ago).

5. v holds the agreed value of the currently terminating $\mathcal{BC}$.

6. v_{prev} holds the value of v one round ago.

Note that all the variables are reset or recomputed periodically, so even if a node begins with arbitrary values in its variables, it will acquire consistent values. Similarly, the memory space of $\mathcal{BC}$ is reset whenever a node initiates and starts to execute a new $\mathcal{BC}$ instance. The consistency of the variable values used for $\mathcal{BC}$ are taken care of within that protocol.

Figure 5.3 presents the digital clock synchronization algorithm.

Remark 4.4.1. *The model allows for only one message to be sent from node p to p' within one round (between two consecutive beats). The digital clock synchronization algorithm in Figure 5.3 requires sending two sets of messages in each round. Observe that the set of messages sent in Step 2 is not dependent on the operations taking place in Step 1, therefore, all messages sent by the algorithm during each round can be sent right after the beat and will arrive and processed before the next beat, meeting the model's assumptions.*

Note that a "simpler" solution, such as running consensus on previous *DigiClock*, incrementing it by $\Delta+1$ and setting that as the current *DigiClock* would not work, because for some specific initial values of *DigiClock* the *Byzantine* nodes can cause the non-faulty nodes to get "stuck" in an infinite loop of alternating values.

4.5 Lemmata and Proofs

All the lemmata, theorems, corollaries and definitions hold only as long as the system is coherent. We assume that all nodes may start in an arbitrary state, and that from some time on, at least $n-f$ of them are concurrently correct. We will denote by $\mathcal{G}$ a

Algorithm **Digital-SSByz-ClockSync** */* executed at each beat */*

1. for each $i \in \{1, .., \Delta\}$ do
   execute the i^{th} round of the $Agree[i]$ BC protocol;
2. send value of $DigiClock$ to all nodes and store the received clocks of other nodes in $ClockVec$;
3. set the following:
   (a) v:= the agreed value of $Agree[\Delta]$;
   (b) $DigiClock_{most}$:= the value appearing at least $\lfloor \frac{n}{2} \rfloor + 1$ times in $ClockVec$, and 0 otherwise;
4. (a) if $(v = 0)$ or $(v = v_{prev} + 1)$ then
   $DigiClock := DigiClock_{most} + 1 \pmod{max\text{-}clock}$;
   (b) else
   $DigiClock := 0$;
5. for each $i \in \{2, ..., \Delta\}$ do
   $Agree[i] := Agree[i-1]$;
6. initialize $Agree[1]$ by invoking $\mathcal{BC}(DigiClock)$.
7. $v_{prev} := v$.

Figure 4.3: The digital clock synchronization algorithm

group of nodes that behave according to the algorithm, and that are not subject to (for some pre-specified number of rounds) any new transient faults. If, $|\mathcal{G}| \geq n - f$ and remain non-faulty for a sufficiently long period of time ($\Omega(\Delta)$ global beats), then the system will converge.

For simplifying the notations, the proof refers to some "external" round number. The nodes do not maintain it, it is only used for the proofs.

Definition 4.5.1. *We say that the system is* $\textsc{Calm}(\alpha, \sigma)$, $\sigma > \alpha$, *if there is a set* $\mathcal{G}$, $|\mathcal{G}| = n - f$, *of nodes that are continuously non-faulty during all rounds in the interval* $[\alpha, \sigma - 1]$.

The notation $\textsc{Calm}(\alpha, \sigma \geq \beta)$ denotes that $\textsc{Calm}(\alpha, \sigma)$ and $\sigma \geq \beta$. Specifically, the notation implies that the system was calm for at least β rounds. Notice that all nodes in $\mathcal{G}$ are considered correct when the system is $\textsc{Calm}(\alpha, \sigma \geq \Delta)$.

Note that in typical self-stabilizing algorithms it is assumed that eventually all nodes behave correctly, and therefore there is no need to define $\textsc{Calm}()$. In our context, since some nodes may never behave correctly, and additionally some nodes may recover and some may fail we need a sufficiently large subset of the nodes to behave correctly for sufficiently long time in order for the system to converge.

In the following lemmata, $\mathcal{G}$ refers to the set implied by $\textsc{Calm}(\alpha, \sigma)$, without stating so specifically.

Lemma 4.5.2. *If the system is* $\textsc{Calm}(\alpha, \sigma \geq \Delta + 1)$*, then for any round* β*,* $\beta \in [\alpha + \Delta + 1, \sigma]$*, all nodes in* $\mathcal{G}$ *have identical values of* v *after executing Step 2 of Digital-SSByz-ClockSync.*

Proof. Irrespective of the initial states of the nodes in $\mathcal{G}$ at the beginning of round α (which is after the last transient fault in $\mathcal{G}$ occurred), the beats received from the global beat system will cause all nodes in $\mathcal{G}$ to perform the steps in synchrony. By the end of round α, all nodes in $\mathcal{G}$ reset $\mathcal{BC}$ protocol $Agree[1]$.

Note that at each round another $\mathcal{BC}$ protocol will be initialized and after Δ rounds from its initialization each such protocol returns the same value at all nodes in $\mathcal{G}$, since all of them are non-faulty and follow the protocol. Hence, After $\Delta + 1$ rounds, the values all nodes in $\mathcal{G}$ receive as outputs of $\mathcal{BC}$ protocols are identical. Therefore v is identical at all $g \in \mathcal{G}$, after executing Step 2 of that round.

Since this claim depends only on the last $\Delta + 1$ rounds being "calm", the claim will continue to hold as long as such a $\mathcal{G}$ set of nodes not experiencing transient faults exists. Thus, this holds for any round β, $\alpha + \Delta \leq \beta \leq \sigma$. □ □

Lemma 4.5.3. *If the system is* $\textsc{Calm}(\alpha, \sigma \geq \Delta + 2)$*, then for any round* $\beta \in [\alpha + \Delta + 1, \sigma]$*, either all nodes in* $\mathcal{G}$ *perform Step 4.a, or all of them perform Step 4.b.*

Proof. By Lemma 4.5.2, after the completion of Step 2 of round $\alpha + \Delta + 1$ the value of v is the same at all nodes of $\mathcal{G}$, hence after an additional round the value of v_{prev} is the same at all nodes of $\mathcal{G}$. Since the decision whether to perform Step 4.a or Step 4.b depends only on the values of v, and v_{prev}, all nodes in $\mathcal{G}$ perform the same line (either 4.a or 4.b). Moreover, because this claim depends on the last $\Delta + 2$ rounds being "calm", the claim will continue to hold as long as no node in $\mathcal{G}$ is subject to a fault. □ □

Denote $\Delta_1 := \Delta+2$. All the following lemmata will assume the system is $\textsc{Calm}(\alpha, \beta)$, for rounds $\beta \geq \Delta_1$. Therefore, in all the following lemmata, we will assume that in each round β, all nodes in $\mathcal{G}$ perform the same Step 4.x (according to Lemma 4.5.3).

Lemma 4.5.4. *If the system is* $\textsc{Calm}(\alpha, \sigma \geq \Delta_1)$*, and if at the end of some* $\beta \geq \alpha + \Delta_1 - 1$*, all nodes in* $\mathcal{G}$ *have the same value of DigiClock, then at the end of any* β'*,* $\beta \leq \beta' \leq \sigma$*, they will have the same value of DigiClock.*

Proof. Since we consider only $\beta \geq \alpha + \Delta_1 - 1$, by Lemma 4.5.3 all nodes in $\mathcal{G}$ perform the same step in Step 4. For round $\beta' = \beta + 1$, the value of *DigiClock* can be changed at Lines 4.a or 4.b. If it was changed at 4.b then all nodes in $\mathcal{G}$ have the value 0 for *DigiClock*. If it was changed by Step 4.a, then because we assume that at round β all nodes in $\mathcal{G}$ have the same *DigiClock* value, and because $|\mathcal{G}| = n - f \geq \lfloor \frac{n}{2} + 1 \rfloor$, the value of $DigiClock_{most}$ computed at round β' is the same for all nodes in $\mathcal{G}$, and therefore, executing Step 4.a will produce the same value for *DigiClock* in round β' for all nodes in $\mathcal{G}$.

By induction, for any $\beta \leq \beta' \leq \sigma$, all nodes in $\mathcal{G}$ continue to agree on the value of *DigiClock*. □ □

Denote $\Delta_2 := \Delta_1 + \Delta + 1$. All the following lemmata will assume the system is $\textsc{Calm}(\alpha, \sigma)$, for $\sigma \geq \Delta_2$.

Lemma 4.5.5. *If the system is* $\textsc{Calm}(\alpha, \sigma \geq \Delta_2)$*, then at the end of any round* $\beta, \beta \in [\alpha + \Delta_2 - 1, \sigma]$*, the value of DigiClock at all nodes in* $\mathcal{G}$ *is the same.*

Proof. Consider any round $\beta' \in [\alpha + \Delta_1 - 1, \alpha + \Delta_1 + \Delta - 1]$. If at the end of β' all nodes in $\mathcal{G}$ hold the same *DigiClock* value, then from Lemma 4.5.4 this condition holds for any β, $\beta \in [\alpha + \Delta_2 - 1, \sigma]$. Hence, we are left to consider the case where at the end of any such β' not all the nodes in $\mathcal{G}$ hold the same value of *DigiClock*. This implies that Step 4.b was not executed in any such round β'. Also, if Step 4.a was executed during any such round β', and there was some *DigiClock* value that was the same at more than $\frac{n}{2}$ nodes in $\mathcal{G}$, then after the execution of Step 4.a, all nodes would have had the same *DigiClock* value. Hence, we assume that for all β', only Step 4.a was executed, and that no more than $\frac{n}{2}$ from $\mathcal{G}$ had the same *DigiClock* value.

Consider round $\beta'' = \alpha + \Delta_1 + \Delta$. The above argument implies that at round $\beta'' - \Delta$, Step 4.a was executed, and there were no more than $\frac{n}{2}$ nodes in $\mathcal{G}$ with the

same *DigiClock* value. Since $\frac{n}{2} < n - 2 \cdot f$, the "solidarity" requirement of $\mathcal{BC}$ implies that the value entered into v at round β'' is $\perp$. Hence, at round β'' Step 4.b would be executed.

Therefore, during one of the rounds $\beta \in [\alpha + \Delta_1 - 1, \alpha + \Delta_1 + \Delta]$, all the nodes in $\mathcal{G}$ have the same value of *DigiClock*, and from Lemma 4.5.4 this condition holds for all rounds, until σ. □ □

Remark 4.5.6. *The requirement that $f < \frac{n}{4}$ stems from the proof above. That is because we require that $\frac{n}{2} < n - 2 \cdot f$ (to be able to use the "solidarity" requirement of $\mathcal{BC}$). We note that this is the only place that the requirement $f < \frac{n}{4}$ appears, and that it is a question for future research whether this can be improved to the known lower bound of $f < \frac{n}{3}$.*

Corollary 4.5.7. *If the system is* Calm$(\alpha, \sigma \geq \Delta_2)$*, then for every round β, $\beta \in [\alpha + \Delta_2 - 1, \sigma - 1]$, one of the following conditions holds:*

1. *The value of DigiClock at the end of round $\beta + 1$ is "0" at all nodes in $\mathcal{G}$.*
2. *The value of DigiClock at the end of round $\beta + 1$ is identical at all nodes in $\mathcal{G}$ and it is the value of DigiClock at the end of round β plus "1".*

Lemma 4.5.8. *If the system is* Calm$(\alpha, \sigma \geq \Delta_2 + \Delta)$*, then for every round $\beta \in [\alpha + \Delta_2 + \Delta - 1, \sigma]$, Step 4.b is not executed.*

Proof. By Corollary 4.5.7, for all rounds β, $\beta \in [\alpha + \Delta_2 - 1, \sigma - 1]$ one of the two conditions of the *DigiClock* values holds. Due to the "validity" property of $\mathcal{BC}$, after Δ rounds, the value entered into v is the same *DigiClock* value that was at the nodes in $\mathcal{G}$, Δ rounds ago. Therefore, after Δ rounds, the above conditions hold on the value of v, v_{prev}. Hence, for any round β, $\beta \in [\alpha + \Delta_2 + \Delta - 1, \sigma]$ one of the conditions holds on v, v_{prev}. Since for both of these conditions, Step 4.a is executed, Step 4.b is never executed for such a round β. □ □

Corollary 4.5.9. *If the system is* Calm$(\alpha, \sigma \geq \Delta_2 + \Delta)$*, then for every round β, $\beta \in [\alpha + \Delta_2 + \Delta - 1, \sigma]$, it holds that all nodes in $\mathcal{G}$ agree on the value of DigiClock and increase it by "1" at the end of each round.*

Corollary 4.5.9 implies, in a sense, the convergence and closure properties of algorithm Digital-SSByz-ClockSync.

Theorem 4.5.10. *From an arbitrary state, once the system stays coherent and there are $n-f$ nodes that are non-faulty for $3\Delta+3$ rounds, Digital-SSByz-ClockSync ensures convergence to a synchronized_state. Moreover, as long as there are at least $n-f$ correct nodes at each round the closure property also holds.*

Proof. The conditions of the theorem implies that the system satisfies the property $\textsc{Calm}(\alpha, \sigma \geq \Delta_2 + \Delta)$. Consider the system at the end of round $\Delta_2 + \Delta$ and denote by $\bar{\mathcal{G}}$ a set of $n-f$ correct nodes implied by $\textsc{Calm}(\alpha, \sigma \geq \Delta_2 + \Delta)$. Consider all the Δ instances of $\mathcal{BC}$ in their memory. Denote by $\mathcal{BC}_i$ the instance of $\mathcal{BC}$ initialized i $(0 \leq i \leq \Delta - 1)$ rounds ago. By Lemma 4.5.8, Step 4.b is not going to be executed (if the nodes in $\bar{\mathcal{G}}$ will continue to be non-faulty). Therefore, at the end of the current round,

1. the set of inputs to each $\mathcal{BC}_i$ contained at least $\lfloor \frac{n}{2} \rfloor + 1$ identical values from non-faulty nodes, when it was initialized (denote that value I_i);
2. for every i, $0 \leq i \leq \Delta - 1$, either $I_i = I_{i+1}$ or $I_i = 0$;
3. I_0 is the value that at least $\lfloor \frac{n}{2} \rfloor + 1$ non-faulty hold in their DigiClockat the end of the current round.

The first property holds because otherwise, by the "solidarity" property of $\mathcal{BC}$, the agreement in that $\mathcal{BC}$ will be on $\perp$ and Step 4.b will be executed. The second property holds because otherwise Step 4.b will be executed. The third property holds since this is the value they initialized the last $\mathcal{BC}$ with.

Observe, that each $\mathcal{BC}_i$ will terminate in $\Delta - i$ rounds with a consensus agreement on I_i, as long as there are $n-f$ non-faulty nodes that were non-faulty throughout its Δ rounds of execution. Thus, under such a condition, for that to happen some nodes from $\bar{\mathcal{G}}$ may fail and still the agreement will be reached. Therefore, Corollary 4.5.9 holds for each node that becomes correct, i.e., was non-faulty for Δ rounds, because it will compute the same values as all the other correct nodes.

By a simple induction we can prove that the three properties above will hold in any future round, as long as for each $\mathcal{BC}$ there are $n-f$ non-faulty nodes that executed it.

Thus, the three properties imply that the basic claim in Corollary 4.5.9 will continue to hold, which completes the proof of the convergence and closure properties of the system. □ □

Note that if the system is "synchronized" and the actual number of *Byzantine* faults f' is less than f, then Theorem 4.5.10 implies that any non-faulty node that is not in $\mathcal{G}$ (there are no more than $f - f'$ such nodes) synchronizes with the *DigiClock* value of nodes in $\mathcal{G}$ after at most Δ global beats from its last transient fault.

4.6 Complexity Analysis

The clock synchronization algorithm presented above converges in $3 \cdot \Delta + 3$ rounds. That is, it converges in $\Omega(f)$ rounds (since $\Delta = 2 \cdot f + 4$ for our $\mathcal{BC}$ of choice).

Once the system converges, and there are at least $|\mathcal{G}| = n - f$ correct nodes, the $\mathcal{BC}$ protocol will stop executing after 4 rounds for all nodes in $\mathcal{G}$ (due to the early stopping feature of $\mathcal{BC}$ we use). During each round of $\mathcal{BC}$, there are n^2 messages exchanged. Note that we execute Δ concurrent $\mathcal{BC}$ protocols; hence, over a period of Δ rounds, there are $\Delta \cdot 4 \cdot n^2$ messages sent. Therefore, the amortized message complexity per round is $O(n^2)$. Note that the early stopping of $\mathcal{BC}$ does not improve the convergence rate. It only improves the amortized message complexity.

4.7 Discussion

4.7.1 Additional Results

The digital clock synchronization algorithm presented here can be quickly transformed into a token circulation protocol in which the token is held in turn by any node for any pre-determined number of rounds and in a pre-determined order. The pre-determined variables are part of the required incorruptible code. E.g. if the token should be passed every k beats, then node p_i, $i = 1 + \lfloor \frac{DigiClock}{k} \rfloor \pmod{n}$ holds the token during rounds $[k \cdot (i-1)+1, k \cdot i]$. Similarly, it can also produce synchronized pulses which can then be used to produce the self-stabilizing counterpart of general *Byzantine* protocols by using the scheme in [9]. These pulses can be produced by setting *max-clock* to be the pulse cycle interval, and issuing a pulse each time $DigiClock = 0$.

4.7.2 Digital Clock vs. Clock Synchronization

In the described algorithm, the non-faulty nodes agree on a common integer value, which is regularly incremented by one. This integer value is considered "the synchronized (digital) clock value". Note that clock values estimating real-time or real-time rate can be achieved in two ways. The first one, is using the presented algorithm to create a new distributed pulse, with a large enough cycle, and using the algorithm presented in [1] to synchronize the clocks. The second, is to adjust the local clock of each node, according to the value of the common integer value, multiplied by the predetermined length of the beat interval.[1] This way, at each beat of the global beat system, the clocks of all the nodes are incremented at a rate estimating real-time.

[1] This value need also be defined as part of the incorruptible code of the nodes.

Chapter 5

Pulse Synchronization

This chapter details a solution for the PULSE synchronization problem. The algorithm given in this chapter supports up to $\frac{n}{3}$ *Byzantine* nodes, and all references to f assume that $f < \frac{n}{3}$.

5.1 Chapter Outline

Section 5.2 provides an overview of the proposed solution. Section 5.3 defines the requirement of the $\mathcal{BBB}$, a sub-routine used by the given solution. In Section 5.4 describes the basic algorithm used to solve the PULSE synchronization problem. Section 5.5 proves the correctness of the algorithm provided in Section 5.4. The following section, Section **??**, uses the algorithm in Section 5.4 to produce the required solution. Sections 5.7 discusses the connectivity requirements of the given algorithm. The rest of the chapter is dedicated to complexity analysis of the algorithm, and a discussion of the results.

5.2 Solution Overview

We first show how to construct a $[\Delta, \Delta + Cycle']$-PULSER $\mathcal{A}$ for any $Cycle' > \Delta$, where Δ is a bound on running a given distributed agreement protocol. We continue by showing how to construct a $[\phi + \psi]$-PULSER from any $[\phi, \psi]$-PULSER. Using this, we construct a $[2 \cdot \Delta + Cycle']$-PULSER $\mathcal{A}'$ for any $Cycle' > \Delta$. Lastly, using $\mathcal{A}'$, we construct a $[Cycle]$-PULSER for any $Cycle \geq 1$.

The construction of $\mathcal{A}$ uses a building block that is similar to the *Byzantine firing squad* problem. We denote this building block by $\mathcal{BBB}$ (*Byzantine* Black Box).

5.3 The *Byzantine* Black Box Agreement

$\mathcal{BBB}$ is defined to be a round based distributed protocol, such that each node p has a binary input value v_p and a binary output value $\mathcal{V}_p$. $\mathcal{BBB}$ has the following properties:

1. *Termination*: The algorithm terminates within Δ rounds.
2. *Agreement*: All *non-faulty*[1] nodes agree on the same output value $\mathcal{V}$. That is, for any two *non-faulty* nodes p, p' it holds that $\mathcal{V}_p = \mathcal{V}_{p'}$.
3. *Validity*: $\mathcal{V} = 1$ only when there exists at least one *non-faulty* node that has initiated the protocol with 1 as its input value. If there are $f + 1$ *non-faulty* nodes that have initiated the protocol with input value equal to 1, then $\mathcal{V} = 1$.

$\mathcal{BBB}$ is required to be *Byzantine* tolerant, but is not required to be self-stabilizing. The self-stabilization of the $[\phi + \psi]$-PULSER $\mathcal{A}$ (presented in a later section) will not be hampered by this. In addition, $\mathcal{A}$ will rely only on the properties of $\mathcal{BBB}$ (when it is executed by enough correct nodes) for its operation. In $\mathcal{A}$ all messages exchanged among the nodes will use $\mathcal{BBB}$. Since the presented $\mathcal{BBB}$ can tolerate $f < \frac{n}{3}$ *Byzantine* nodes, $\mathcal{A}$ can tolerate the same ratio of permanent *Byzantine* failures.

Remark 5.3.1. *A $\mathcal{BBB}$ protocol that satisfies the above properties can be constructed by executing concurrently a separate* Byzantine *agreement on the value of each node. Once all these agreements terminate (after some Δ rounds), check whether there are $f + 1$ "1"s in the resulting vector of output values. One can easily prove that this protocol satisfies the above properties when there are more than $2n/3$ nodes that where continuously non-faulty throughout all the Δ rounds of executing the protocol.*

the algorithm above executes $n \cdot \Delta$ Byzantine *agreement algorithms concurrently. It can be improved to execute only Δ algorithms concurrently by executing* Byzantine *consensus instead of* Byzantine *agreement.*

As commented above, since $\mathcal{BBB}$ requires the nodes to maintain a consistent state throughout the protocol, a non-faulty node that has recently recovered from a

[1] A node is considered non-faulty in $\mathcal{BBB}$ only if it is non-faulty throughout the whole execution of $\mathcal{BBB}$.

transient fault cannot be considered correct. In the context of this chapter, a non-faulty node is considered *correct* once it remains non-faulty for at least $\Delta_{node} = \Delta + 1$ and as long as it continues to be non-faulty.

5.4 A $[\Delta, \Delta + Cycle']$-PULSER for $Cycle' > \Delta$

Figure 5.1 presents an algorithm that produces a $[\Delta, \Delta + Cycle']$-PULSER, for $Cycle' > \Delta$. This algorithm executes Δ simultaneous $\mathcal{BBB}$ protocols. We denote by $\mathcal{BBB}_i$ the ith instance of $\mathcal{BBB}$. $\mathcal{V}(\mathcal{BBB}_i)$ refers to the output value $\mathcal{V}$ of $\mathcal{BBB}_i$ (when defined).

Algorithm **Large-Cycle-Pulser** */* executed at each beat */*

1. for each $i \in \{1, .., \Delta\}$ do
   execute the i^{th} round of the $\mathcal{BBB}_i$ protocol;
2. (a) if $Counter > 0$ then
   $Counter := \min\{Counter - 1, Cycle'\}$;
   $WantToPulse := 0$;
   (b) else
   $WantToPulse := 1$;
3. if $\mathcal{V}(\mathcal{BBB}_\Delta) = 1$ then
   (a) do PULSE;
   (b) $Counter := Cycle'$;
4. for each $i \in \{2, ..., \Delta\}$ do
   $\mathcal{BBB}_i := \mathcal{BBB}_{i-1}$;
5. initialize $\mathcal{BBB}_1$ by invoking $\mathcal{BBB}(WantToPulse)$.

Figure 5.1: A $[\Delta, \Delta + Cycle']$-PULSER algorithm for $Cycle' > \Delta$.

5.5 Proof of Large-Cycle-Pulser's correctness

All the lemmata, theorems, corollaries and definitions hold only as long as the system is coherent. We assume that nodes may start in an arbitrary state, and nodes may fail and recover, but from some time on, at any round there are at least $n - f$ correct nodes.

Let $\mathcal{G}$ denote a group of non-*Byzantine* nodes that behave according to the algorithm, and that are not subject to (for some pre-specified number of rounds) any new transient faults. We will prove that if $|\mathcal{G}| \geq n - f$ and all of these nodes remain

non-faulty for a long enough period of time ($\Omega(\Delta)$ global beats), then the system will converge.

For simplifying the notations, the proofs refer to some "external" beat number. The nodes do not maintain it and have no access to it, it is only used for the proofs' clarity.

Definition 5.5.1. *A group $\mathcal{G}$ is* CORRECT(α, β) *if $|\mathcal{G}| \geq n - f$, and every node $p \in \mathcal{G}$ is correct during the beat interval $[\alpha, \beta]$. Let δ mark the length of the interval, that is $\delta = \beta - \alpha + 1$.*

Note that each node $p \in \mathcal{G}$, when $\mathcal{G}$ is CORRECT(α, β), has not been subject to a transient fault in the beat interval $[\alpha - \Delta_{node}, \beta]$; and is non-faulty during that interval.

Definition 5.5.2. *We say that a system is* CORRECT(α, β) *if there exists a set $\mathcal{G}$ such that $\mathcal{G}$ is* CORRECT(α, β).

In the following lemmata, $\mathcal{G}$ refers to any set implied by CORRECT(α, β), without stating so specifically. The proofs hold for any such set $\mathcal{G}$.

Note that if the system is coherent, and there has not been a transient fault for at least $\Delta + 1$ beats, then $\mathcal{G}$ contains all nodes that were non-faulty during that period.

Lemma 5.5.3. *$\forall \beta \geq \alpha$: If the system is* CORRECT(α, β) *then at any beat $r \in [\alpha, \beta]$, either all nodes in $\mathcal{G}$* PULSE *or they all do not* PULSE.

Proof. A node PULSEs only in line 3.a, which is executed only when the value of $\mathcal{V}(\mathcal{BBB}_\Delta) = 1$. All nodes in $\mathcal{G}$ have not been subject to transient faults in the $\Delta_{node} = \Delta + 1$ beats preceding r. Therefore, $\mathcal{BBB}_\Delta$ has been initialized properly Δ beats ago, and during the Δ rounds of $\mathcal{BBB}_\Delta$'s execution, it has been executed properly by at least $n - f$ nodes. Hence, according to *Agreement* of $\mathcal{BBB}$, all nodes in $\mathcal{G}$ have the same value of $\mathcal{V}(\mathcal{BBB}_\Delta)$. Therefore, all nodes in $\mathcal{G}$ "act the same" when considering line 3.a: either all of them execute line 3.a or they all do not execute it. This holds for any beat after α (as long as $\mathcal{G}$ continues to contain $n - f$ correct nodes). Therefore, at any such beat $r \in [\alpha, \beta]$, either all nodes in $\mathcal{G}$ PULSE or they all do not PULSE. □

Lemma 5.5.4. *$\forall \beta \geq \alpha + \Delta + Cycle'$: If the system is* CORRECT(α, β)*, then at some beat $r \in [\alpha, \beta]$ all nodes in $\mathcal{G}$* PULSE.

Proof. According to the previous lemma, all nodes in $\mathcal{G}$ PULSE together during the interval $[\alpha, \beta]$. Hence, if one of them PULSEd in the interval $[\alpha, \alpha + Cycle']$, all of them PULSEd, and we are done.

Otherwise, consider the case where no node in $\mathcal{G}$ has PULSEd in the interval $[\alpha, \alpha + Cycle']$. Hence, at beat $\alpha + Cycle'$, for all the nodes in $\mathcal{G}$, the *Counter* variable has decreased to 0 or is negative. This is because *Counter* is bounded from above by $Cycle'$ (which is a fixed parameter of the protocol and is identical at all nodes); and as long as it holds a positive value, it decreases by 1 during each beat of the interval $[\alpha, \alpha + Cycle']$ (since no node PULSEs in that interval, *Counter* never increases). Since the interval is at least $Cycle'$ beats long, the value of *Counter* is less than (or equal to) 0.

Therefore, at beat $\alpha + Cycle'$ there are $|\mathcal{G}| \geq n - f > f + 1$ correct nodes with $WantToPulse = 1$. Therefore, according to *Validity* of $\mathcal{BBB}$, Δ beats afterwards $\mathcal{V}(\mathcal{BBB}_\Delta)$ will output 1, and all nodes in $\mathcal{G}$ will PULSE. Thus, in the interval $[\alpha, \alpha + \Delta + Cycle']$ all nodes in $\mathcal{G}$ PULSE. Therefore, the claim holds for any beat interval $[\alpha, \beta]$, where $\beta \geq \alpha + \Delta + Cycle']$. □

Remark 5.5.5. *The above lemma proves* progress. *That is, starting from any state, eventually there will be a* PULSE.

Consider a system that is CORRECT(α, β) (for $\beta \geq \alpha + \Delta + Cycle'$), from Lemma 5.5.3, starting from beat α all nodes in $\mathcal{G}$ PULSE together. From Lemma 5.5.4, by beat $\alpha + \Delta + Cycle'$ all nodes in $\mathcal{G}$ have PULSEd. Therefore, by that round they have all reset their *Counter* values at the same beat. Since $WantToPulse$ depends solely on the value of *Counter*, and since all nodes in $\mathcal{G}$ agree on the output value of the $\mathcal{BBB}$ protocols, all nodes in $\mathcal{G}$ perform exactly the same lines of code following each beat in the beat interval $[\alpha + \Delta + Cycle', \beta]$.

Lemma 5.5.6. $\forall \beta \geq \alpha + 3 \cdot \Delta + 2 \cdot Cycle'$*: If the system is* CORRECT(α, β)*, then the system is* $[\Delta, \Delta + Cycle']$-PULSING *in the beat interval* $[\alpha + 3 \cdot \Delta + 2 \cdot Cycle', \beta]$.

Proof. According to previous lemmas, all correct nodes PULSE at some beat γ, no later than beat $\alpha + \Delta + Cycle'$; and from then on they all PULSE together. At beat γ they all reset their counters and will have positive *Counter* values for at least $Cycle'$ rounds. Since $Cycle' > \Delta$, in the following Δ beats, the value of $WantToPulse$ will be 0, and hence $\mathcal{BBB}_1$ is initialized during these beats with the value 0. Therefore, once these values will emerge from $\mathcal{BBB}$, there will be a period of $Cycle' > \Delta$ with

no pulses. That "quite" period will start at beat $\gamma + \Delta$. This quite period might be longer than $Cycle'$ if there were other PULSEs during the beat interval $[\gamma, \gamma + \Delta]$. In any case, a quite period will commence at beat $\gamma + \Delta$ and will be at least $Cycle'$ beats long, and no more than $Cycle' + \Delta$ beats long.

Now consider what happens after this quite period. Eventually, the value of $WantToPulse$ will be set to 1 (after no more than $Cycle' + \Delta$ beats), and will stay so until the next PULSE. Mark the beat at which all nodes in $\mathcal{G}$ set $WantToPulse$ to 1 as γ'. Notice that because the quite period is greater than Δ then once its values start "coming out" of $\mathcal{BBB}_\Delta$ there will be quite for at least Δ beats. Hence, once $WantToPulse$ is set to 1, it will stay that way for Δ beats, until 1 comes out of $\mathcal{BBB}_\Delta$. This will happen at beat $\gamma' + \Delta$. Once this happens, there are Δ 1's "on the way" in the coming $\mathcal{BBB}$s. Therefore, there will be a PULSE for Δ beats. Due to the first PULSE, $WantToPulse$ will be 0 for all the Δ PULSE beats. After the last PULSE beat, $WantToPulse$ will be 0 for an additional $Cycle'$ beats. Afterwards, $WantToPulse$ will turn to 1, and will stay so for Δ beats. Thus there is a pattern of $WantToPulse$ being 0 for $\Delta + Cycle'$ beats then being 1 for Δ beats, and so on. Therefore, the PULSEing pattern will be the same, as required.

Note that the PULSEing pattern starts on beat $\gamma' + \Delta$, and the pattern continues (at least) until beat β. Hence, the system is $[\Delta, \Delta + Cycle']$-PULSING in the beat interval $[\gamma' + \Delta, \beta]$. Because $\gamma' \leq \gamma + Cycle' + \Delta$ and since $\gamma \leq \alpha + \Delta + Cycle'$, we conclude that $\gamma' + \Delta \leq \alpha + 3 \cdot \Delta + 2 \cdot Cycle'$, as required. □

Remark 5.5.7. *The above lemma shows that the convergence time of the* PULSE*ing algorithm depends on the value of Cycle. However, since for clock synchronization the value of Cycle is in the order of* Δ*, the convergence of the clock synchronization will depend on* Δ *and not on the value of* max-clock *(the wrap around value of the digital clock).*

The following theorem states that we have constructed a $[\Delta, \Delta + Cycle']$-PULSER.

Theorem 5.5.8. *The Large-Cycle-Pulser algorithm is a* $[\Delta, \Delta + Cycle']$-PULSER.

Proof. By Lemma 5.5.6, once there are enough nodes that have not been subject to transient faults for $3 \cdot \Delta + 2 \cdot Cycle'$ beats, the system becomes $[\Delta, \Delta + Cycle']$-PULSING for the beat interval $[\gamma' + \Delta, \beta]$. This is true for any $\beta \geq \alpha + 3 \cdot \Delta + 2 \cdot Cycle'$. Hence, as long as the system is coherent, once the system is $[\Delta, \Delta + Cycle']$-PULSING in the beat interval $[\gamma' + \Delta, \beta]$, it is also $[\Delta, \Delta + Cycle']$-PULSING in the beat interval $[\gamma' + \Delta, \beta + 1]$; and therefore Large-Cycle-Pulser algorithm is a $[\Delta, \Delta + Cycle']$-PULSER. □

5.6 A $[Cycle]$-PULSER for $Cycle > 0$

In the previous section a $[\Delta, \Delta + Cycle']$-PULSER was presented, for any value of $Cycle' > \Delta$. Now a general way to transform a $[\phi, \psi]$-PULSER into a $[\phi + \psi]$-PULSER is given. Combining this with the previous result will demonstrate how to construct a $[2 \cdot \Delta + Cycle']$-PULSER. Since $Cycle' > \Delta$, the technique constructs a $[Cycle]$-PULSER, for any $Cycle > 3 \cdot \Delta$. In Subsection 5.6.2 this requirement is eliminated, and the objective of building $[Cycle]$-PULSER is achieved for any $Cycle > 0$.

5.6.1 $[\phi, \psi]$-PULSER to $[\phi + \psi]$-PULSER

Given a $[\phi, \psi]$-PULSER $\mathcal{A}$, the following algorithm $\mathcal{B}$ uses $\mathcal{A}$ as a black-box:

Algorithm $[\phi + \psi]$-PULSER /* *executed at each beat* */

1. execute a single round of $\mathcal{A}$;
2. if $\mathcal{A}$ PULSEd at the current beat and $\mathcal{A}$ did not PULSE at the previous beat, then $\mathcal{B}$ PULSEs at the current beat.

Figure 5.2: An algorithm that transforms a $[\phi, \psi]$-PULSER into a $[\phi + \psi]$-PULSER.

Note that the above algorithm $\mathcal{B}$ does not rely on anything other than the output of $\mathcal{A}$ in the current and previous beats. Hence, if $\mathcal{A}$ is self-stabilizing, so is $\mathcal{B}$.

Theorem 5.6.1. *The algorithm $\mathcal{B}$ is a $[\phi + \psi]$-PULSER.*

Proof. $\mathcal{A}$ is a $[\phi, \psi]$-PULSER, hence, it PULSEs in a pattern of ϕ pulses, then ψ quite rounds. Therefore, once every $\phi + \psi$ beats, there is a transition from not PULSEing to PULSEing. Thus, the PULSEing output of $\mathcal{A}$, implies that exactly once every $\phi + \psi$ beats it holds that $\mathcal{A}$ PULSEd at the current beat, and did not PULSE at the previous beat. This is continuously true (as long as $\mathcal{A}$ continues to PULSE), which implies that the proposed algorithm $\mathcal{B}$ will PULSE exactly once every $\phi + \psi$ beats, in a pattern of a single PULSE, and then $\phi + \psi - 1$ beats of quite rounds.

Since $\mathcal{A}$ is a $[\phi, \psi]$-PULSER, starting from an arbitrary state, it eventually starts PULSEing in the required pattern, and continues so as long as the system is coherent. Hence, the above algorithm $\mathcal{B}$ will eventually start PULSEing in the expected pattern, and will continue so as long as the system is coherent. Hence it is a $[\phi+\psi]$-PULSER. □

Using the above, the previously built $[\Delta, \Delta + Cycle']$-PULSER $\mathcal{A}$ is transformed it into a $[2 \cdot \Delta + Cycle']$-PULSER $\mathcal{B}$. Since $\mathcal{A}$ required that $Cycle' > \Delta$ we actually constructed a $[Cycle]$-PULSER for any $Cycle > 3 \cdot \Delta$.

In the next subsection this limitation will be removed.

5.6.2 Eliminating the $Cycle > 3 \cdot \Delta$ Requirement

Building upon the $[2 \cdot \Delta + Cycle']$-PULSER, $\mathcal{B}$, from the previous subsection, a $[Cycle]$-PULSER, $\mathcal{C}$, for any $Cycle \leq 3 \cdot \Delta$ is created.

Algorithm $[Cycle \leq 3 \cdot \Delta]$-PULSER *executed at each beat*

set $Cycle' > \Delta$ to be such that $Cycle' + 2 \cdot \Delta$ is divisible by $Cycle$

1. execute $\mathcal{B}$;
2. if $\mathcal{B}$ PULSEd at the current beat then $Counter := Cycle' + 2 \cdot \Delta$;
3. if $Counter$ is divisible by $Cycle$ then $\mathcal{C}$ PULSEs at the current beat;
4. $Counter := Counter - 1$.

Figure 5.3: A $[Cycle]$-PULSER algorithm for $1 \leq Cycle \leq 3 \cdot \Delta$.

Theorem 5.6.2. *The algorithm $\mathcal{C}$ is a $[Cycle]$-PULSER for any $1 \leq Cycle \leq 3 \cdot \Delta$.*

Proof. Since $\mathcal{B}$ is a $[Cycle' + 2 \cdot \Delta]$-PULSER, starting from an arbitrary state, eventually it starts PULSEing in a pattern of a single PULSE, and then $Cycle' + 2 \cdot \Delta - 1$ beats of quite (and continues so as long as the system is coherent). Therefore, eventually, all correct nodes will see the same PULSEing output from $\mathcal{B}$. Hence, each time $\mathcal{B}$ pulses, all correct nodes set $Counter$ to $Cycle' + 2 \cdot \Delta$, and have the same value of $Counter$ at each beat (because they all set it together, and decrease it together). Hence, each time a correct node enters line 3, all correct nodes do the same. Therefore, all correct nodes have $\mathcal{C}$ PULSE together. Lastly, since each $Cycle$ beats $Counter$ will be divisible by $Cycle$, $\mathcal{C}$ PULSEs once every $Cycle$ beats.

Therefore, for each PULSE of $\mathcal{B}$ we have $\frac{2 \cdot \Delta + Cycle'}{Cycle}$ PULSEs of $\mathcal{C}$. Due to the choice of $Cycle'$ such that $2 \cdot \Delta + Cycle'$ is divisible by $Cycle$, the PULSEs are nicely aligned with the PULSEs of $\mathcal{B}$; and therefore, the above algorithm $\mathcal{C}$ is a $[Cycle]$-PULSER. □

Remark 5.6.3. *Setting $Cycle'$ such that $2 \cdot \Delta + Cycle'$ is divisible by $Cycle$ is crucial. Otherwise, the* PULSE*ing pattern would not be of a $[Cycle]$-*PULSER.

Theorem 5.6.4. *For any $Cycle > 0$ a $[Cycle]$-PULSER can be constructed.*

Proof. If $Cycle > 3 \cdot \Delta$, then set $Cycle' := Cycle - 2 \cdot \Delta$. By Theorem 5.5.8, construct a $[\Delta, \Delta + Cycle']$-PULSER and by Theorem 5.6.1 construct a $[2 \cdot \Delta + Cycle']$-PULSER. According to the choice of $Cycle'$ the required $[Cycle]$-PULSER is constructed.

If $Cycle \leq 3 \cdot \Delta$, calculate $Cycle'$ such that $Cycle' > \Delta$ and $\frac{2 \cdot \Delta + Cycle'}{Cycle}$ is an integer number. Now, by Theorem 5.5.8 build a $[\Delta, \Delta + Cycle']$-PULSER. From Theorem 5.6.1 construct a $[2 \cdot \Delta + Cycle']$-PULSER. Finally, from the above algorithm construct an algorithm that is a $[Cycle]$-PULSER, as required. □

5.7 Network Connectivity

The above discussion did not assume anything about the network connectivity. More precisely, the only connectivity assumption was about the behavior of the $\mathcal{BBB}$ protocol. That is, whatever connectivity $\mathcal{BBB}$ requires to operate properly, is the required connectivity in order for the $[Cycle]$-PULSER construction to work properly.

In [20] it is shown that *Byzantine* agreement is achievable if and only if:

1. f is less than one-third of the total number of nodes in the system.
2. f is less than one-half of the connectivity of the system (that is, between any two nodes there are at least $2 \cdot f + 1$ distinct paths).

Therefore, since $\mathcal{BBB}$ is implemented by executing *Byzantine* Agreement for each node's input value, $\mathcal{BBB}$ can be tolerant up to $\frac{n-1}{3}$ *Byzantine* faults. In addition $\mathcal{BBB}$ can work properly even if the connectivity graph is not fully connected, but rather there are at least $2 \cdot f + 1$ distinct paths between any two non-faulty nodes.

Remark 5.7.1. *As noted in [20], the nodes are required to know the connectivity graph while executing the algorithm. This implies, due to self-stabilization, that each node has the network connectivity as incorruptible data.*

Since the PULSEing algorithm presented in this chapter depends solely on $\mathcal{BBB}$ for communication with other nodes, it is tolerant up to $\frac{n-1}{3}$ *Byzantine* faults and can operate in a network where there are at least $2 \cdot f + 1$ distinct paths between any two nodes, and it is optimal with respect to these two parameters.

Previous algorithms do not easily extend to operated in a network that is not fully connected. This is a result of their dependency of their "current state" on

messages received in the "current round"; in a network that is not fully connected, such messages are received $\mathcal{D}$ rounds later, where $\mathcal{D}$ is the diameter of the network.

For example, in [10], the *DigiClock* value depends on the values sent in the current round. Therefore, if the network is not fully connected, node p does not receive messages from node p' that is not his neighbor, in the same round. Hence, p cannot change its current state according to the algorithm's definition. This does not mean that previous algorithms cannot be transformed to operate in such a setting, just that it is not straightforward.

5.8 Complexity Analysis

Using PULSEing for clock synchronization leads to use a *Cycle* that is in the order of Δ. Hence, the PULSEing algorithm presented in the previous sections converges in $O(\Delta)$ beats. If the system is fully connected, then $\Delta = 2 \cdot (f+1)$, because executing *Byzantine* agreement takes $2 \cdot (f+1)$ rounds (for example, see [19]). Therefore, convergence is reached in $O(f)$ beats.

If the system is not fully connected, as discussed in the previous section, and the diameter is $\mathcal{D}$, then $\Delta = \mathcal{D} \cdot 2 \cdot (f+1)$. Therefore, convergence is reached in $O(\mathcal{D} \cdot f)$ beats.

5.9 Discussion

5.9.1 *Byzantine* Firing Squad

Three synchronization problems are discussed herein. Two are self-stabilizing *and* tolerant to *Byzantine* faults: the PULSEing problem and the digital clock synchronization problem. The third one is the *Byzantine* firing squad problem, which is not self-stabilizing.

As was see in Section 3.4, the Digital Clock synchronization problem is equivalent to the PULSE synchronization problem.

In the (strict) *Byzantine* firing squad problem, at each beat the nodes might receive a "START" message from "the outside". If $f+1$ non-*Byzantine* nodes receive a "START" message then eventually all non-*Byzantine* nodes fire. If a non-*Byzantine* node fires, then at least one non-*Byzantine* node received a "START" message. Lastly,

all non-*Byzantine* nodes fire together. A detailed discussion of this problem is given in [21].

The solution in this chapter was inspired by the solution to the *Byzantine* firing squad problem given in [21]. The $\mathcal{BBB}$ protocol is essentially the same protocol used in [21] to solve the firing squad problem. Thus, the "non self-stabilizing", *Byzantine* firing squad problem can be seen as a basic synchronization element, which is a building block for self-stabilizing, *Byzantine* tolerant synchronization problems, the PULSEing problem and the digital clock synchronization problem.

The self-stabilization property is "added" to the *Byzantine* firing squad solution by repeatedly executing the protocol that solves the firing squad problem. In this way, new instances of the firing squad algorithm (the $\mathcal{BBB}$ instances), that are "clean" of memory faults, are continuously produced.

This tactic of re-executing a sub-solution to some *Byzantine* problem in order to solve a self-stabilizing *and Byzantine* tolerant problem should be explored more thoroughly, as it has already produced two solutions to previously unsolved problems: the current problem, and the problem presented in Chapter 4 (see also [10]).

5.9.2 *Byzantine* Tolerant Stabilizer

We now present briefly how a stabilizer can be built using the PULSEing algorithm provided in the above sections. The stabilizer will *stabilize* a *Byzantine* tolerant algorithm $\mathcal{A}_0$. That is, given a *Byzantine* tolerant algorithm $\mathcal{A}_0$ that is not self-stabilizing, the stabilizer will transform it into a self stabilizing version of $\mathcal{A}_0$ (preserving the *Byzantine* tolerance).

Clearly, not all algorithms can be viewed as self-stabilizing. E.g. an algorithm that is allowed to do some action ACT only once, cannot be a self-stabilizing algorithm. We do not discuss here the requirements of an algorithm $\mathcal{A}_0$ so that it can be stabilized. For a more in depth discussion of such requirements, refer to [17] and [18]. In the following, it is assumed that the *Byzantine* tolerant algorithm $\mathcal{A}_0$ has a meaning as a self-stabilizing algorithm.

Intuitively, every so often, all nodes will collect a global snapshot $\mathcal{S}$ of the local states of all nodes. Then, all nodes inspect $\mathcal{S}$ for any inconsistencies. If any are found, all nodes reset their local state to some consistent state.

Given a general *Byzantine* tolerant algorithm $\mathcal{A}_0$, we construct an algorithm Byz-State-Check. Byz-State-Check gathers a global snapshot of the local states at

each node and ensures that the local states are consistent. In addition if the states were consistent to start with, then Byz-State-Check does not alter them. That is, Byz-State-Check alters the local states to a consistent state, only if required. Figure 5.4 present the algorithm Byz-State-Check.

Algorithm **Byz-State-Check** */* executed at node p */*

1. execute a *Byzantine* agreement on local state of $\mathcal{A}_0$;
2. after all correct nodes see the same global snapshot $\mathcal{S}$:
   (a) if $\mathcal{S}$ represents a legal state
   repair local state if it is inconsistent with $\mathcal{S}$;
   (b) otherwise
   reset local state.

Figure 5.4: A *Byzantine* tolerant state validation and reset.

Given a general *Byzantine* tolerant algorithm $\mathcal{A}_0$, a $[f+1]$-PULSER $\mathcal{P}$ and a Byz-State-Check algorithm $\mathcal{C}$, the algorithm SS-Byz-Stabilizer is constructed, as in Figure 5.5.

Algorithm **SS-Byz-Stabilizer** */* executed at each beat */*

1. execute a single round of $\mathcal{A}_0$;
2. execute a single round of $\mathcal{C}$;
3. execute a single beat of $\mathcal{P}$;
4. if $\mathcal{P}$ PULSEd this beat re-initialize $\mathcal{C}$.

Figure 5.5: A Self-stabilizing *Byzantine* tolerant Stabilizer.

Theorem 5.9.1. *SS-Byz-Stabilizer transforms a* Byzantine *tolerant algorithm* $\mathcal{A}_0$ *into Self-stabilizing* Byzantine *tolerant algorithm.*

Proof. $\mathcal{P}$ is a $[f+1]$-PULSER. Hence, eventually it starts PULSEing $f+1$ beats apart. When this happens, $\mathcal{C}$ is re-executed periodically, and terminates between such 2 executions (executing *Byzantine* agreement takes $f+1$ rounds). Hence, $\mathcal{C}$ performs correctly. This means that the local states of $\mathcal{A}_0$ will be consistent. And we have that starting from any initial state of $\mathcal{A}_0$'s local states, eventually $\mathcal{A}_0$'s local states are consistent. □

Chapter 6

Conclusions and Future Work

6.1 Self-stabilization by "Rotating" Sub-problems

The described solutions in this thesis surface a more general scheme for "rotating" X instances; where in the digital clock problem, X is *Byzantine* consensus, and for the PULSE problem X is *Byzantine* firing squad. This allows all non-faulty nodes to have a valid termination of X each beat. This mechanism is self-stabilizing and tolerates the permanent presence of *Byzantine* nodes. In addition, this mechanism ensures that all non-faulty nodes decide on their next step at the next round, based on the termination of the same instance.

By re-executing the sub-solution to some *Byzantine* problem X, the property of self-stabilization is "added" to it. Each sub-problem requires its own consideration, since it is not enough to execute many sub-problems simultaneously. A careful examination of the transformation between the state at some round r to the next state at round $r + 1$ is required.

In this thesis two different such sub-problems are given, and it is shown how to use them to solve "real" problems. This produces a "general" tactic of creating self-stabilization and *Byzantine* tolerant algorithms, by executing multiple instances of some *Byzantine* tolerant instance (that is not self-stabilizing) and connecting the results from each beat in an intelligent way. The success of this tactic, as applied in the thesis, leads us to believe that additional problems may be solved with it.

6.2 Future Work

We consider four main points to be of interest for future research.

1. How would an algorithm that solves the digital clock synchronization problem directly, with $f < \frac{n}{3}$, look like?

2. What happens if the global beats are received at intervals that are less than the message delay, i.e. common clock ticks. Is there an easy solution to achieve synchronization? If yes, can it attain optimal precision like the above clock synchronization solution? If no, is the only option then to synchronize the clocks in a fashion similar to [13, 11, 1]? i.e. by executing an underlying distributed pulse primitive with pulses that are far enough apart in order to be able to terminate agreement in between. In that case, is there any advantage in having a common source of the clock ticks or is it simply a replacement for the local timers of the nodes?

3. Currently, the only solution for a self-stabilizing *Byzantine* tolerant PULSEing algorithm in a system without a global beat was given by [13]. However, that solution is very complicated. This leaves the following question open: can the above simple scheme of pulsing be transferred into a bounded-delay network?

4. When the communication network is not fully connected, it was shown above that the PULSEing algorithm converges linearly in Δ, were $\Delta = O(f \cdot \mathcal{D})$ (where $\mathcal{D}$ is the diameter of the graph). Can the convergence time be reduced to be linear in $O(f + \mathcal{D})$?

Chapter 7

Appendix: Proofs for $\mathcal{BC}$ Protocol

We now present the proof for the protocol Byz-Consensus presented in Section 4.3 along with the proofs for the correctness of the *broadcast* primitive. We start with the proof of the properties of the *broadcast* primitive; In the following subsection we prove the correctness of the Byz-Consensus algorithm.

7.1 Proof of the Properties of the *broadcast* Primitive

The proofs are very similar to the proofs given in [19]. Where the proofs are identical, we refer the reader to the original proofs in [19].

Claim 7.1.1. *The* Correctness *property holds.*

Proof. We consider *Broadcast* operations for some $p \in P$. Since $I_0 \notin P$, and I_0 does not send or receive any messages, the behavior of the *Broadcast* primitive regarding $p \in P$ is the same as in [19]. Hence, the proof of the *Correctness* property holds, as in [19]. □

Claim 7.1.2. *The* Unforgeability *property holds.*

Proof. The same considerations of the *Correctness* property hold here too and this property is proven the same as in [19]. □

Claim 7.1.3. *The* Relay *property holds.*

Proof. For $p \in P$, the proof from [19] holds. We consider $p = I_0$. Hence, we need to show that if a correct node p accepts $(I_0, m, 1)$ at round r, then every other correct node accepts $(I_0, m, 1)$ in round $r + 1$ or earlier.

If p accepted $(I_0, m, 1)$ in round 1, then p received more than $n - f$ $(echo, I_0, m, 1)$ messages. Hence, more than $n - 2 \cdot f$ were sent by correct nodes, which implies that all correct nodes received $n - 2 \cdot f$ such messages by the end of the second phase of round 1. Therefore, at the first phase of round 2, all correct nodes will send $(init', I_0, m, 1)$ to all nodes. Hence, at the second phase of round 2, all correct nodes will receive at least $n - f$ $(init', I_0, m, 1)$ messages, and they will all send $(echo', I_0, m, 1)$ message to all. Hence, at the end of the second phase of round 2 every correct node received $n - f$ $(echo', I_0, m, 1)$ messages, and therefore it accepts $(I_0, m, 1)$ at round 2.

If p accepted $(I_0, m, 1)$ in round 2, then p received $n - f$ $(echo', I_0, m, 1)$ messages. Hence, at least $n - 2 \cdot f$ such messages were sent by correct nodes in the previous phase of the that round. Therefore, at the first phase of round 3, all correct nodes will receive $n - 2 \cdot f$ $(echo', I_0, m, 1)$ messages and will all send $(echo', I_0, m, 1)$ (if they haven't send such a message yet). Therefore, at the second phase of round 3, all correct nodes will receive $n - f$ $(echo', I_0, m, 1)$ messages, and will accept $(I_0, m, 1)$.

If p accepted $(I_0, m, 1)$ in round $r \geq 3$ or above, then some correct node received $n - f$ $(echo', I_0, m, 1)$ messages. Hence, at least $n - 2 \cdot f$ such messages were sent by correct nodes in previous rounds. Therefore, at round r, all correct nodes will receive $n - 2 \cdot f$ $(echo', I_0, m, 1)$ messages and will all send $(echo', I_0, m, 1)$ at the first phase of round $r + 1$ (if they haven't send such a message yet). Therefore, at the second phase of round $r + 1$, all correct nodes will receive $n - f$ $(echo', I_0, m, 1)$ messages, and will accept $(I_0, m, 1)$.

And we have shown that for every round r, if a correct node accepts $(I_0, m, 1)$, then all correct nodes accept $(I_0, m, 1)$ at round $r + 1$ or earlier.

□

Claim 7.1.4. *The* Detection of broadcasters *property holds.*

Proof. For $p \in P$, the proof from [19] holds. We consider $p = I_0$. Hence, we need to show that if a correct node p accepts $(I_0, m, 1)$ at round 1 or later, then every correct node has $I_0 \in broadcasters$ at the end of round 2.

We consider the first correct node p to accept $(I_0, m, 1)$ at some round r. If $r = 1$ then p received $n - f$ $(echo, I_0, m, 1)$ messages at the second phase of round 1. Hence, all correct nodes received $n - 2 \cdot f$ such messages by the first phase of round 2, and

therefore all correct nodes sent $(init', I_0, m, 1)$ at that phase. Hence, at the second phase of round 2, all correct nodes add I_0 to $broadcasters$.

If $r = 2$, then p received $n - f$ $(echo', I_0, m, 1)$ messages at phase 2 of round 2. Hence, it received at least one such message from a correct node p', and therefore, p' received $n - f$ $(init', I_0, m, 1)$ messages at phase 1 of round 2. Hence $n - 2 \cdot f$ correct nodes sent $(init', I_0, m, 1)$. Therefore, all correct nodes received $n-2 \cdot f$ $(init', I_0, m, 1)$ messages at phase 1 of round 2, and all correct nodes add I_0 to $broadcasters$.

If $r \geq 3$, then p received $(echo', I_0, m, 1)$ from some correct node p' at some round. We consider the first such p'. p' must have received $n - f$ $(init', I_0, m, 1)$ messages at phase 1 of round 2 (otherwise, it received $n - 2 \cdot f$ $echo'$ messages at phase 1 of round $r' \geq 3$ which means it received an $echo'$ message from a correct node from a round before r', in contradiction to p' being the first node to send $echo'$ message). Hence, all correct nodes received $n - 2 \cdot f$ $(init', I_0, m, 1)$ messages at phase 1 of round 2, and therefore, all correct nodes have $I_0 \in broadcasters$ at the end of round 2.

And we have shown that if a correct node accepts $(I_0, m, 1)$, then all correct nodes have $I_0 \in broadcasters$ at the end of round 2. □

7.2 Byz-Consensus – Proof of Correctness

Lemma 7.2.1. *The "Agreement" condition holds for Byz-Consensus.*

Proof. First we show that messages sent as part of the *broadcast* primitive, by correct nodes, contain a single value v. That is, $echo, init'$ and $echo'$ are all sent with $m = v$.

Messages can be sent either due to a *broadcast* executed, or due to line 3.a. If a message is sent due to a *broadcast*, then it is sent since $v \neq \perp$, hence it is sent due to $v = v'$ in line 3.b or in line 4.b. In the later case, it can be sent only if $(I_0, v', 1)$ was accepted, which means that an $(echo, I_0, v', 1)$ was sent by some correct node. If it is due to $v = v'$ in line 3.a it also means that an $(echo, I_0, v', 1)$ was sent by some correct node.

We assume by contradiction that there are two different correct nodes, one sent $(echo, I_0, v_1, 1)$ and the other sent $(echo, I_0, v_2, 1)$. We note that in order for a correct node to send $(echo, I_0, v', 1)$, it must have received $n - f$ message in the first phase of round 1, with the same value v'. Hence, there must have been $n - 2 \cdot f > \frac{n}{2}$ correct nodes with the same initial value. Therefore, more than half of the nodes had v_1 as their initial value, and more than half of the nodes had v_2 as their initial value. And

we reached a contradiction.

An immediate conclusion from the above, is that if all correct nodes return a value other than $\perp$, then they all return the same value. Hence, all we need to show is, that either all correct nodes return $\perp$, or they all return some value other than $\perp$.

If all correct nodes return $\perp$, we're done. Otherwise, there is some correct node p, that returned some value $w \neq \perp$. We examine the node p that was first to return a value $\neq \perp$. In order for node p to return a value $\neq \perp$, it must have set $v = v'$ at some stage. We consider 2 options: v was set due to line 3.b, or it was set later.

If v was set by line 3.b, then p received $n - f$ $(echo, I_0, w, 1)$ messages at round 1. Therefore, according to the *broadcast* primitive, p accepts $(I_0, w, 1)$. Therefore, according to the *Relay* property of the *broadcast* primitive, we have that all correct nodes accept $(I_0, w, 1)$ in round 2. Moreover, p executed *broadcast* $(p, w, 2)$, and due to the *Correctness* property, all correct nodes accept $(p, w, 2)$ by round 2. Hence, when executing line 4.b, all correct nodes have accepted $(I_0, w, 1)$ and $(p, w, 2)$, therefore they all set $v := w$, and in the following loop step, they will all enter line 4.a, decide on w and stop.

We now consider the case where p set $v = w$ after line 3.b. Since $w \neq \perp$, v must have been updated by line 4.b (note that this update can only be done once on p, since afterwards (in the next loop step) p stops executing the protocol). We consider 2 options: v was updated on some round $r < f + 2$, or v was update on round $r = f + 2$. In the first case, v was update at node p on round $r \leq f + 1$. Therefore, p had accepted $(I_0, w, 1)$ and another $r - 1$ messages of the form (p_i, w, i) for all $i, 2 \leq i \leq r$. Therefore, due to the *Relay* property, all correct nodes will accept $(I_0, w, 1)$ and (p_i, w, i) for all $i, 2 \leq i \leq r$ by next round. Also, since next round $v \neq \perp$ at p, p will enter line 4.a, and will *broadcast* $(p, w, r + 1)$, and due to the *Correctness* property, all correct nodes accept $(p, w, r + 1)$ at round $r + 1$. Therefore, at round $r + 1$, all correct nodes have accepted $(I_0, w, 1)$ and (p_i, w, i) for all $i, 2 \leq i \leq r + 1$, and therefore, all correct nodes set $v := w$. Note that all correct nodes don't stop on line 4.c, due to the *Detection of broadcasters* property. That is, because p accepted $r - 1$ $(\_, w, i)$ for all $1 \leq i \leq r - 1$, then each correct node has at least $r' - 1$ nodes in *broadcasters* by round r' for all $1 \leq r' \leq r$. Therefore, no correct node stopped due to line 4.c.

We now consider the last option, that is, that p updated $v := w \neq \perp$ at round $f + 2$. Therefore, p accepted $(I_0, w, 1)$ and (p_i, w, i) for all $i, 2 \leq i \leq f + 2$. Because there are no more than f *Byzantine* nodes, we have that one of the $p_i = p'$ is a correct

node. Due to the *Unforgeability* property, p' broadcasted (p', w, j) at some round j. In order for p' to broadcast, it must have set $v := w$ at some round. We choose p to be the first node to set $v := w$, and hence, p' cannot set $v := w$ at round $j < f + 1$, and we reach a contradiction. Hence, either the first correct node to set its value $v := w$ does so in a round $r \leq f + 1$, or it doesn't do so at all.

And we have shown that if some node returned $w \neq \perp$ then all correct nodes return w, otherwise, all correct nodes return $\perp$. □

Lemma 7.2.2. *The "Validity" condition holds for Byz-Consensus.*

Proof. If all correct nodes have the same initial value v_0, then in phase 1, all correct nodes send (v_0). Therefore, at the end of phase 1, all correct nodes received $n - f$ distinct messages of the form (v_0). Hence, at phase 2, all correct nodes send $(echo, I_0, v_0, 1)$ to everyone. Therefore, at the end of phase 2, all correct nodes receive $n - f$ distinct messages of the form $(echo, I_0, v_0, 1)$. Hence, according to line 3.b, they all set $v = v_0$, and at the beginning of line 4.a, they stop and return the agreed value v_0.

And we have shown that if all correct nodes have the same initial value $v_0 \neq \perp$ then that is their output value. □

Lemma 7.2.3. *The "Termination" condition holds for Byz-Consensus.*

Proof. Define $\Delta = 2 \cdot (f + 2)$. Clearly, within Δ phases, all correct nodes terminate. This is because each round consists of two phases, and the main loop executes no more than $f + 1$ times, in addition to the 2 phases executed outside of the loop. □

Lemma 7.2.4. *The "Solidarity" condition holds for Byz-Consensus.*

Proof. There are two locations in which the output can be changed such that it is not $\perp$. Either in line 3.b, or in line 4.b. If at least one correct node returned a value v' due to execution of line 3.b, then it received $n - f$ $(echo, I_0, v', 1)$ messages. Hence, it received at least one such echo message from a correct node p'. Therefore, in the previous round, p' received at least $n - f$ (v') messages. Therefore, at least $n - 2 \cdot f$ correct nodes send (v') which means that at least $n - 2 \cdot f$ had v' as their initial value. And "Solidarity" holds.

If all correct nodes returned v' due to executing line 4.b, then it means that a correct node accepted $(I_0, v', 1)$. Therefore, it either received $n - f$ $(echo', I_0, v', 1)$ messages, or $n - f$ $(echo, I_0, v', 1)$ messages. In the first case, at least one correct

node had to send $(echo', I_0, v', 1)$ message, which means that it received at least one $(init', I_0, v', 1)$ message from a correct node, which means that this node received at least one $(echo, I_0, v', 1)$ message from a correct node. And we have that in order for a correct node to accept $(I_0, v', 1)$, some correct node must have sent $(echo, I_0, v', 1)$ at some time. This message can be sent (by a correct node) only at line 3.a. Therefore, the correct node that sent it, received at least $n - f$ messages of the form (v'). This means that at least $n-2\cdot f$ correct nodes had v' as their initial value. And "Solidarity" holds. □

Bibliography

[1] 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.

[2] 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.

[3] B. Liskov. Practical use of synchronized clocks in distributed systems. In *Proceedings of 10th ACM Symposium on the Principles of Distributed Computing*, 1991.

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

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

[6] S. Dolev and J. L. Welch. Wait-free clock synchronization. *Algorithmica*, 18(4):486–511, 1997.

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

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

[9] 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.

[10] 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.

[11] A. Daliot, D. Dolev, and H. Parnas. Self-stabilizing pulse synchronization inspired by biological pacemaker networks. In *In Proceedings of the Sixth Symposium on Self-Stabilizing Systems (DSN SSS '03)*, LNCS 2704, San Francisco, Jun 2003.

[12] A. Daliot and D. Dolev. Self-stabilizing byzantine token circulation. Technical Report TR2005-77, School of Engineering and Computer Science, The Hebrew University of Jerusalem, Jun 2005. Url: http://leibniz.cs.huji.ac.il/tr/834.pdf.

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

[14] 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.

[15] M. R. Malekpour and R. Siminiceanu. Comments on the byzantine self-stabilizing pulse synchronization protocol: Counterexamples. Technical Memorandum NASA-TM213951, NASA, Feb 2006. http://hdl.handle.net/2002/16159.

[16] Y. Afek and S. Dolev. Local stabilizer. In *Proc. of the 5th Israeli Symposium on Theory of Computing Systems (ISTCS97)*, Bar-Ilan, Israel, Jun 1997.

[17] A. S. Gopal and K. J. Perry. Unifying self-stabilization and fault-tolerance. In *IEEE Proceedings of the 12th annual ACM symposium on Principles of distributed computing*, Ithaca, New York, 1993.

[18] S. Katz and K. J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, 1993.

[19] S. Toueg, K. J. Perry, and T. K. Srikanth. Fast distributed agreement. *SIAM Journal on Computing*, 16(3):445–457, Jun 1987.

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

[21] J. E. Burns and N. A. Lynch. The byzantine firing squad problem. *Advances in Computing Research*, 4:147–161, 1987.