A Layered Synchronized Simulation

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

by

Noam Teomim

Supervised by

Prof. Danny Dolev

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

December 2015

סימולציית סינכרון רב שכבתית

תזה זו מוגשת כחלק מתנאי הסף
הנחוצים לקבל תואר
Master of Science

הוגש ע"י
נועם תאומים

בהנחיית
פרופסור דני דולב

ביה"ס להנדסה ולמדעי המחשב
האוניברסיטה העברית בירושלים
ישראל

טבת תשע"ה

Acknowledgments

First, I would like to express my deepest gratitude to Prof. Danny Dolev. Danny was the first to introduce me to this fascinating world of distributed algorithms. Since then, he also guided me through my lab project. Under his guidance, I finally decided that this is the field I want to focus on for my academic research. In the last three years, Danny supported me in every stage of my research. In his endless patience, support and guidance during this period, Danny thought me the gentles insights and the fine details that make distributed systems so complicated and challenging. I was truly inspired by his deep understanding of the field, his reach experience, and by his passion for distributed protocols.

Second, I would like to gratefully thank all of the Computer Science dept. stuff in the Hebrew University for all of their support. Especially, I would like to thanks Hagit Yaar-On, for her kindness, and for her advices during the completion of this thesis.

Third, I would like to thank my parents, Avi and Sara, for both the moral and financial support during my studies. The trust you gave me, and endless encouragement, gave me the strength to keep going even in difficult times.

I would also like to thank my children: Yinon, Shahar and Hadar, for their smiles and hugs which always reminded me of the bright side of life.

Last but not least, to my wonderful wife Renana, for her endless support and love. For pushing me forward and helping me out in every aspect of life during a very intensive and challenging period.

Abstract

This thesis presents a modular protocol which is aimed to solve the clock synchronization problem over a semi-synchronized system (also known as a bounded-delay system). The protocol is a self-stabilizing protocol that can cope with up to $\frac{N}{2}$ faulty processors subject to omission faults (where N is the number of devices in the system). The protocol converges within a finite time, bounded by the number of faulty processors in the system. We show that for such synchronization models this bound is a lower bound for deterministic protocols. We used a modular methodology in order to divide the task into three simpler sub-tasks. The modular method simplifies the constructed solution and also gives it the ability to easily adjust to other systems using the same core protocol. This algorithm can be simply applied as a wrapper around existing distributed protocols that assume a synchronized faultless system, making it highly usable for porting legacy protocols to modern systems.

Hebrew Abstract

תזה זו מציגה פרוטוקול מודולרי שפותר את בעיית סנכרון השעונים במערכת סמי-סינכורונית (הידועה גם בשם "bounded-delay system"). הפרוטוקול הוא בעל יכולת התייצבות עצמית (self-stabilizing) שמסוגל לתפקד כראוי, כל עוד מספר המעבדים התקולים במערכת הוא מתחת ל- $\frac{N}{2}$ (כאשר N הוא מספר המעבדים במערכת). הפרוטוקול מתכנס בזמן סופי שחסום ע"י מספר המעבדים התקולים במערכת. אנו מראים כי עבור מודל הסנכרון שאותו הפרוטוקול מנסה לדמות, חסם זה הוא אופטימלי עבור אלגוריתם דטרמיניסטי. הפרוטוקול משתמש בשיטה מודולרית על מנת לחלק את משימת הסנכרון לתתי-משימות פשוטות יותר. חלוקה זו מפשטת את הפרוטוקול המוצע, ואף נותנת לו את הגמישות להתאמה עם מודלי שגיאות אחרים (כדגומת המודל הביזנטי) ללא שינוי משמעותי בליבו של הפרוטוקול. ניתן להשתמש באלגוריתם זה כמעטפת לאלגוריתמים מבוזרים קיימים שמניחים מערכת סינכורנית ללא שגיאות. ע"י מעטפת זו ניתן להשתמש בפרוטקולים הללו בקלות יחסית וללא שינויים רבים, גם במערכות רבות שבהן הנחות אלו אינן מתקיימות.

Contents

Acknowledgments **3**

Abstract **4**

Hebrew Abstract **5**

1 Introduction **8**

2 Related Work **12**
- 2.1 Clock Synchronization Algorithms 12
- 2.2 Self-Stabilizing Clock Synchronization 13
- 2.3 Simulations and Layers 15
- 2.4 Omission Error Model 16

3 Model of the System and Problem Definition **17**
- 3.1 Model of the System 17
  - 3.1.1 Synchronous System: 17
  - 3.1.2 Semi-Synchronous System: 18
- 3.2 Problem Statement 19
- 3.3 Drift handling in the System 21

4 Simulating a Fail-Stop Fault Model over an Omission Fault Model **23**
- 4.1 The FS Simulation Problem 23
- 4.2 The FS Simulation Protocol (FSSP) 25
  - 4.2.1 Processors' Variables 26
  - 4.2.2 Protocol's Messages 26
  - 4.2.3 FSSP Rules 27
- 4.3 FSSP as an Emulation layer 33

5 Adjusting a Semi-Synchronized System to a Pair-Wise Synchronized System in the Emulated Fail-Stop Model 34
5.1 The Pair-Wise Clocking Protocol (PWCP) 36
5.2 Processors' Variables 37
5.3 Protocol Messages and Definitions 38
5.4 PWCP Protocol 39
5.5 Claims and Proofs 40
5.6 Ticking Mechanism 49
5.7 Simulate Pair-Wise Synchronized System Using PWCP 52

6 Pulse Synchronization in Pair-Wise Synchronized Systems 54
6.1 Pulse Synchronization Definitions 54
6.2 Pulse Synchronization Lower Bound 55
6.3 Gain the Globally Synchronized property using the Common Virtual Pulse 60
6.4 The Global Sync Protocol 62
6.5 The Protocol 63

7 Conclusions and Future Work 66
Bibliography 67

Chapter 1

Introduction

Traditionally, distributed computing challenges such as clock synchronization and fault tolerance are relevant in practice to areas such as internet protocols or sensor networks. These areas are characterized by a relatively high robustness to faults in the system, and by a reliable timing mechanism (such as an accurate internal clock and steady communication overhead). Due to the reliability of the system's components, such systems often assume that no faults exist and manage to work comparatively smoothly with that assumption. And, due to the reliable timing, such systems can also be assumed to be fully synchronized (as if there exists an external clock that can be sampled by every component in the system).

Nowadays, we are seeing a drastic increase in individual computing components in a very wide spectrum of devices. Systems may now comprise many devices, from relatively robust mobile phones to significantly less robust wearable and embedded devices, which also have significantly weaker time evaluation capacities and unreliable *fault-tolerance* mechanisms. Any protocol assigned to these systems needs to be tolerant of as wide a spectrum of faults as possible.

In addition, collaboration in such an enormous system with different components makes the *synchronization* between these components essential and difficult. Synchronization is crucial because subsets (or even subsets of subsets) from the computing units have to cooperate with one another in order to achieve their desired goal. On the other hand, it is far from trivial to achieve the synchronization property when the private clock that each device holds is highly unreliable.

When handling large-scale systems, we cannot assume that all (or even

most) of the components in the system have a common starting point. Devices might plug in or out and start their execution with unknown starting values and with no set schedule. Thus, *self-stabilization* is also crucial for the system's correct functioning. Furthermore, modern systems are expected to recover from every random state within a reasonable finite time, which is another important reason why self-stabilization is needed.

This thesis focuses on creating synchronization over a *semi-synchronized system* (also known as the *bounded-delay* model). In distributed systems many tasks rely on time being common to all components of the system. *Synchronizing* a distributed system means establishing a counter that shows the same value upon access by any component of the system. Algorithms that achieve this are called *synchronizers* [1] and have been studied and understood since the early days of distributed computing [2]. Unfortunately, in most systems, the time it takes to transmit a message and process it cannot be precisely predicted or measured and usually varies among the devices in the system. This makes the synchronized model impractical for real world systems: apart from inconsistencies in end-to-end message delivery, it also leads to inconsistent timing of each processor's clock value readings. Fortunately, however, a wide range of scenarios fit the semi-synchronized system model, in which reasonable bounds for the transmission and processing timings exist and are known by all the devices in the system.

Another significant obstacle that comes into play in real world distributed systems is *clock drift*. Signal propagation speed on computer chips depends on many uncontrollable factors, such as temperature, variable component quality, and voltage fluctuations. This means that, although every component of the system is equipped with an almost identical clock, different clocks will exhibit different drifts. We address the issue of clock drift in order to simulate a synchronized system.

The constructed protocol is a *self-stabilizing protocol*. Conceptually, a self-stabilizing model is one in which the system may be unstable and undergo fairly arbitrary changes for an unknown (arbitrarily long) period. These changes are thought of as transient errors. From some point, however, transient errors end and all processes start behaving according to their protocols. A self-stabilizing algorithm is one in which the system's behavior is guaranteed to conform to its intended specification within finite time following the last transient error. Formally, self-stabilizing algorithms are modelled as starting out at an arbitrary initial state (corresponding to arbitrary transient failures) and following the prescribed protocol, with the goal of converging

into correct behavior in finite time. Self-stabilization is a broad subfield of fault-tolerance in distributed systems.

The error model that we address is the *omission* error model. By 'omission' we mean that every faulty processor in the system might drop messages addressed to it, or might drop messages that it is supposed to send. This model fits a very wide spectrum of errors common in distributed systems, for example, loss of packets due to anomalies in the network's traffic, unexpected failures of links in the system, routing failures, etc. Omission faults are very common in practice and pose a significant challenge to many protocols. We present a protocol that can cope with up to $\frac{N}{2}$ faulty processors (where N is the number of processors in the system). Clearly, any deterministic protocol that relies on the data from the other processors cannot cope with more than $\frac{N}{2}$ faulty processors, and so our protocol is optimal in that sense.

The protocol uses a modular strategy to simplify the solution by dividing it into several sub-tasks, each sub-task is assigned a protocol to handle it, and every protocol acts as a building block - such that the final outcome is built by layering these building blocks one on top of the other. These protocol layers include simulations of properties that the system does not otherwise have. Simulation is a well known technique, defined formally in [3] and in [4]; we use the simulation definition as described in [5].

The simulation technique uses layers, each layer simulating a system with some extra characteristics beyond those of the layer below it. The lowest layer is the actual communication system, and the highest layer has the properties that we wish to achieve (fully synchronized, for example). Using a layered model not only simplifies the task, but also allows the protocol to be extended to other natural directions by replacing only one of the layers. For example, adjusting the protocol to cope with Byzantine faults instead of omission faults requires changing only one layer, leaving the other layers untouched (as mentioned in Chapter 7). The same argument holds for modifying the algorithm from a deterministic solution to a randomized protocol that can converge in expected constant time. Thus, the layering methodology gives the protocol robustness and agility in addition to simplifying the solution.

Traditionally, systems used protocols that assumed the system was synchronized and had no faults. Currently, such protocols try to make some adjustments (as few as possible) in order to cope with faults and to support bounded delay systems. Such changes are very difficult and entail a high risk of mishandling scenarios or damaging the properties of the original protocol. This thesis takes such an existing protocol, and wraps it in a new layer that

simulates the assumed fault tolerance and synchronization. The advantage of using a protocol that wraps the original protocol is clear: every system can keep working with its original protocol as an upper layer above our protocol and due to this it is promised to have the assumed requirements. This allows practical integration to existing systems with imperceptible effects.

To summarize, this thesis presents a modular protocol that aims to synchronize a semi-synchronized system (also known as a bounded-delay system). The resulting protocol is a self-stabilizing protocol that can cope with up to $\frac{N}{2}$ faulty processors (where N is the number of processors in the system), where the faulty processors are subject to omission faults. The protocol converges within finite time bounded by the number of faulty processors in the system. It will be shown that for such a deterministic synchronization model this bound is a lower bound. This algorithm can be simply wrapped around existing distributed protocols that assume synchronized faultless systems, making it highly usable for porting legacy protocols to modern systems.

The rest of the thesis is organized as follows. Chapter 2 presents the main literature pertaining to this research. Chapter 3 formally defines the network and the error model; it also defines a simulation and describes how to simulate one system over another. Each subsequent section formally defines the sub-task of simulating a system over the previous layer, and presents a protocol that can complete the respective sub-task:

- Chapter 4 presents the sub-task of simulating an *emulated fail stop* system over a semi-synchronized system with omission faults.
- Chapter 5 presents the sub-task of simulating a *pair-wise synchronized* system over an *emulated fail stop* system.
- Chapter 6 presents the sub-task of simulating a *simulated synchronous* system over a *pair-wise synchronized* system.

Finally, Chapter 7 presents the main conclusions of this thesis and proposes some natural directions for future research.

Chapter 2

Related Work

2.1 Clock Synchronization Algorithms

Clock synchronization algorithms are algorithms that ensure that physically dispersed clocks associated with different processors in the system all have a common knowledge of time. In other words, every processor in the system has approximately the same view of time as all the others. The classic approach for clock synchronization algorithms assumes that all clocks are initially synchronized and focuses on keeping them synchronized in the presence of clock drift and faulty processors.

The first to present the problem was [2], later developed by [6], who showed that in the presence of F Byzantine (i.e. maliciously failing) processors, $3F+1$ processors are sufficient in order to keep the system synchronized. They assumed that all clocks are initially "close enough" to each other, and that the overall system is synchronized, and presented two algorithms that keep the maximum difference between two non-faulty processors within a certain bound, δ. These algorithms are based on statistical evaluations of the clocks' values. Later, [7] and [8] proved the necessity of $3F+1$ processors to tolerate Byzantine faults, thus proving the optimality of the faulty vs. non-faulty processor ratio presented in [6].

Paper [6] also presented a third algorithm that can handle up to $\frac{N}{2}$ faulty processors but requires digital signatures, a model referred to as the *authenticated Byzantine model*. This was further developed in [9] to use a clock synchronization algorithm that can cope with any link or processor fault, and not to restrict the ratio of faulty to non-faulty processors, so long as all

non-faulty processors remain connected by reliable links.

Since the initial study in [6], the clock synchronization problem has been extensively studied over the last three decades. Thus, several surveys can be found, such as [10], [11], [12] and [13]. The surveys differ from one another by the way they categorize the different clock synchronization algorithms: in [10] and [11] the algorithms are classified according to their accuracy and precision; in [12] the surveyed algorithms are listed according to the system synchrony (the upper bound on the communication latency) and the error models it can tolerate, and in [13] a more ambitious classification is made by identifying repeated internal structures and basic building blocks that are common for several sets of clock synchronization algorithms. Another natural classification distinguishes between deterministic, probabilistic and statistical algorithms.

2.2 Self-Stabilizing Clock Synchronization

In this thesis, the focus was to solve the *self-stabilizing clock synchronization problem*, which is defined as the clock synchronization problem without the assumption of initial synchronization among the system components. Initial solutions, such as [14], used external protocols to establish the initial synchronization.

As research in the self-stabilized clock synchronization problem has evolved, initial suspicions of unsolvability have been dismissed. The first positive results were probabilistic algorithms [15] [16], which stabilize within exponential time for both synchronized and semi-synchronized models. The main concept of this solution is to maintain a steady state in which all non-faulty processors are synchronized, and to transfer to a predefined Reset state if a processor identifies that the system state is not steady. The possibility of moving from the Reset state to the steady system state is decided probabilistically (a coin flip in the synchronized model and a random value in the semi-synchronized model). [17] took a similar approach but differs in his treatment of the unsteady state: after identifying an imbalance, a component then transfers to a Restore state and follows a new protocol until the stable state is reestablished.

Like [16], this thesis also predefines a Reset state to which processors move upon noticing instability. However, our approach differs in that we use deterministic criteria for moving out of the Reset state, where these criteria

ensure that the time spent in the Reset state is bound by a constant. This allows the entire protocol convergence time to be linearly upper bounded by the number of the faulty processors (which, as mentioned, is optimal).

The next research milestone involved using a deterministic algorithm known as the *pulse synchronization problem* to reduce the clock synchronization time-bound from exponential to polynomial. [18] defined the pulse synchronization problem based on the corresponding biological phenomenon, and showed the direct connection between pulse synchronization and clock synchronization. This was further developed in [19] and in [20], which combine existing Byzantine consensus protocols in order to achieve pulse synchronization, and then use a pulse synchronization protocol to solve the clock synchronization problem on a semi-synchronized system with Byzantine faults. This approach achieves a state-of-the-art clock synchronization run-time bound of O(n), where n is the number of nodes in the system.

The current work also uses pulse synchronization to achieve the clock synchronization, and also uses a similar approach of binding several algorithms in order to solve the problem. However, while [19] and [20] incorporate the external secondary algorithm as part of the primary algorithm's execution, we use the more modular approach of dividing the problem into layers, with each layer dependent only on the layers below it. This way, even if one layer has not yet reached its stable state, the layers below it can nevertheless complete their stabilization unhindered.

A randomized pulse synchronization algorithm was recently proposed [21] that broadcasts a constant number of bits in constant time, and which can achieve O(n) stabilization time with probability 1-$2^{-\Omega(n)}$. This results in O(n^2) bits being sent by each non-faulty node during stabilization. Other probabilistic solutions that reach expected constant convergence time use common-coin protocols, as in [22]. This thesis focuses on the deterministic model, although a very natural probabilistic approach can extend from this research (as mentioned in Chapter 7).

This thesis uses a similar approach to [20] to handle *clocks' drift* in the system, encapsulating the drift rate bound as part of the bound on the processing and transmission time. This method is natural for bounded-delay systems since the ability to reach full similarity between different clocks is restricted, as proven in [23], [24].

The linear bound on the number of faulty processors for the classic clock synchronization problem was shown in [25]. The synchronized model presented here maintains this same lower bound; this is shown in Chapter 6,

using a similar technique to the one used in [26], in which convergence to a synchronized system is shown to encapsulate the classic synchronous consensus problem (presented in [4]). Since this thesis' protocol converges within the linear bound of the classic problem, it can also be considered optimal.

2.3 Simulations and Layers

This thesis relies heavily on a modular layer-based architecture, in which each layer refines the previous layer by adding a few extra properties. The first study of a reliable layer on top of an existing system was in [27], defining the *reliable message transmission problem (RMTP)*. This approach was later expanded in [28] and [29], using a simulation layer to adapt the classic *Byzantine consensus problem* (which does not support broadcast) to a *Byzantine broadcast* protocol, thus optimizing the message complexity of the Byzantine consensus problem. [30] solved the clock synchronization problem using a simulation layer referred to as publication, which is similar to the broadcast method mentioned above.

Constructing two layers of simulation was first proposed in [31]. The first layer is constructed similarly to [28] and [29], and fulfils the similar goal of enabling broadcast in the Byzantine consensus problem. However, [31] takes the layering a step further and builds a second layer, which simulates authenticated signatures over the broadcast mechanism. This allows the execution of authenticated Byzantine protocols, such as in [9] and [6].

This thesis takes it one natural step further, and uses three layers of abstraction to simulate the synchronized system. However, this thesis also abstracts the modularity of the layers: each layer simulates a different system (e.g. fail-Stop, synchronized, etc.) as opposed to an isolated action (i.e. broadcast, safe transmission, etc.). This allows each layer to be independent of the layers above it, and to continue functioning for as long as the layers below it allow.

This approach significantly improves the system's convergence time, since the lower layers can keep working towards their steady state even if the top layers are not stable. This means that the convergence time of the entire protocol becomes the *sum* - rather than the *product* - of the convergence times of all layers. However, it also requires a more complex architecture, including a dedicated interface between the layers that is able to handle the various stages of instability. This introduces the disadvantage of greater

fragility and complexity of interaction.

The system simulation methodology naturally leads to the use of the simulation formalization presented by [5], which focuses on the interface (API) between the user and the simulation layer, as well as on the interface between the simulation layer and the underlying system. This formalization simplifies the building of simulation layers so long as a suitable API is maintained for every pair of layers. It is therefore more suitable for our model than other approaches, such as the one presented in [3], where the main focus is on the system's output and behavior.

2.4 Omission Error Model

This thesis uses the omission error model, as opposed to the Byzantine error model used in most of the papers mentioned above. The omission error model was first discussed in [32] in the context of communication networks, and was referred to as *changing topology networks* since it is equivalent to the problem of failing communication links that frequently change the underlying network topology. It was part of the motivation for establishing the concept of *distributed network protocols (DNP)* in [33] and was solved by [34], who took advantage of the system's connection redundancy to ensure the quality of service.

The model made its debut in the distributed systems literature, as the omission error model, in [35]. Here, a broadcast primitive is used to maintain a joint view of the system and thus identify and overcome any omissions. A similar approach was taken in [36], showing that the use of broadcast can improve the solution of the consensus problem even in the presence of omission faults.

This thesis chooses to use the omission fault model as opposed to the Byzantine fault model and uses a similar approach to [35] and [36] in order to deal with these faults. The main reason for this is to simplify the protocols used. However, a very natural extension would be to cope with Byzantine faults, as mentioned in Chapter 7. This would require a change to the lowest layer of the protocol, which currently shifts the error model from omission to fail-stop. The *grade-cast* algorithm [37] simulates fail-stop over a Byzantine error model in synchronized systems, and has been used in layer-based protocols such as [38]. The adjustment of grade-cast protocol to the semi-synchronous model is not trivial but seems to be achievable.

Chapter 3

Model of the System and Problem Definition

3.1 Model of the System

In this thesis, we assume a semi-synchronous distributed system, and present a protocol that simulates a synchronous system over it. These terminologies will be defined formally in this section.

The system is modeled as a set of N processors $P_1, P_2 ... P_N$, that communicate by message transmission. Each processor in the system has a set (not necessarily finite) of actions that it can perform. We assume a fully connected network in which any pair of processors can reliably exchange messages. We also assume that the network is reliable in the sense that every message that is accepted by a processor in the system was indeed sent by a sending processor.

3.1.1 Synchronous System:

In the synchronous system model all processors are connected to a global pulse system, which provides a common pulse at a regular interval and each pulse reaches all nodes simultaneously. In addition, all processors can send a set of messages, receive all other messages, and complete all of their internal processing between one pulse and another. The time interval between one pulse and another is called a phase. A message sent from P_i to P_j at phase r is guaranteed to reach P_j, and to be internally processed by P_j before phase $r+1$.

3.1.2 Semi-Synchronous System:

Note that in the semi-synchronous model we refer to a global reference clock and to local clocks. The global clock is a theoretical clock that we can use in our proofs; every time interval mentioned in this thesis refers to this clock. In this model, the processors do not have access to the global reference time. However, processors have local clocks, which are not necessarily synchronized; they may run arbitrarily fast or slow as long as they are 'close enough' to each other in comparison. The concepts relating to local clock times are clarified in the formal definitions in this section.

By a semi-synchronous system, we mean that there exists a constant time τ_{trans}, such that after the end of the transient faults, every message in the system (including in transit messages) either reaches its destination or gets lost within τ_{trans} clock units. In addition, we assume the existence of a constant bound, d_{trans}, on the time it takes to send/receive and process messages. For every pair of non-faulty processors (P_i, P_j), $\tau_{trans} + \tau_{process} < d_{trans}$, where $\tau_{process}$ is the time P_j requires in order to process P_i's message.

Although we assume an upper bound on the processing and message transmission time, the only assumption we make on the lower bound is that it is positive. Formally, there exists a constant $\varepsilon_{min} \in (0, d_{trans})$, such that a message may pass from one processor to another, only ε_{min} clock units after it was sent (including the processing time).

Definition 1. Each processor P_i is equipped with a *hardware clock* H_i that is a continuous, strictly-increasing function H_i: $R_0^+ \to R_0^+$, mapping the global clock into P_i's local time.

Definition 2. We mark the *maximum drift rate* between different clocks in the system as $\alpha > 1$. α is defined such that for any two times (in the reference clock), t, t', $t < t'$ we mark $T := t' - t$, and it holds that: $T \leq H_i(t') - H_i(t) \leq \alpha T$.

Definition 3. A processor P_i is considered to be *non-faulty* if it complies with the following conditions:

1. P_i operates according to the instructed protocol.
2. For every message that P_i sends to any other non-faulty processor P_j at t_{send}, and is accepted and processed by P_j until t_{acc}, it holds that $\varepsilon_{min} < t_{acc} - t_{send} < d_{trans}$.

A processor that is not non-faulty is said to be *faulty*.

We mark the upper bound on the number of faulty processors as F. Our model assumes that $\frac{N}{2} > F \geq 0$.

In this thesis we consider two fault models for the faulty processors in the system:

1. Omission faults - Any faulty processor might fail to send or receive messages. Formally, if a certain processor P_i is faulty, then any message that P_i sends to any other processor P_j might not be received by P_j, and any message sent to P_i, may not be accepted by P_i.

2. Fail-stop faults - Any faulty processor might stop functioning at any given point, including after sending messages to some but not all of the other processors in the system. Once a faulty processor has stopped, no non-faulty processor receives any more messages from it.

3.2 Problem Statement

Definition 4. For every processor P_i we define *P_i's phases* according to the global reference clock as a set of time intervals $D_i = \{\Delta_{i_1}, \Delta_{i_2}...\}$, where $\Delta_{i_k} = [t_k, t_{k'}]$, $t_{k'} > t_k$, which satisfies the following conditions:

1. For every $\Delta_{i_k} \in D_i$, where $\Delta_{i_k} = [t_k, t_{k'}]$, $t_{k'} - t_k \geq d_{trans}$.

2. Every message that was sent/accepted by P_i is associated with one of Pi's phases. Formally, for every two processors P_i and Pj, and message m_{ij} sent at time t_k there exists a time interval $\Delta_{i_{m_{ij}}} \in D_i$ such that $t_k \in \Delta_{i_{m_{ij}}}$.

3. If processor P_i sends a message to processor P_j then P_i will not send another message to P_j in the same phase.

4. There is no overlap between any two phases. Formally, for every two phases, $\Delta_{i_k}, \Delta_{i_\ell} \in D_i$, where $\Delta_{i_k} = [t_k, t_{k'}]$, and $\Delta_{i_\ell} = [t_\ell, t_{\ell'}], \Delta_{i_k} \cap \Delta_{i_\ell} = \emptyset$.

P_i's *local phase* is defined as the mapping of its phases into its local clock. So, for every phase $\Delta_{i_k} \in D_i$, where $\Delta_{i_k} = [t_k, t_{k'}]$, its corresponding local phase is $\delta_{ik} = [H_i(t_k), H_i(t_{k'})]$.

Note that the labelling of local phases is not necessarily the internal representation of the local phase at the processor. The next definition captures that.

Definition 5. Each processor P_i holds a variable called $\theta_i[\ell] \in \mathbb{N}_0^+$, which contains the *phase labelling* of its current local phase δ_{i_ℓ}. P_i changes its label to the next label $\theta_i[\ell+1]$ when it moves to the next local phase. Note that every protocol can use a different labelling function, $L : \mathbb{N}_0^+ \rightarrow \mathbb{N}_0^+$, to map the local phase index (which is of cardinality $\aleph_0$) to the value of $\theta_i[\ell]$, which has a finite range of values.

Definition 6. We consider a distributed system to be *pair-wise synchronized* if it complies with the following conditions:

1. All processors in the system have the predefined constant τ, such that for each processor in the system P_i, and every $d_{i_k} \in D_i$, $t_{k'} - t_k \geq \tau$.

2. For every pair of non-faulty processors (P_j, P_i) $(j \neq i)$, if P_j sends a message in its local phase δ_{j_k} and P_i receives it in its local phase δ_{i_ℓ}, P_j will not send any messages in any ensuing local phases $\delta_{j_{k'}}, k' > k$ until P_i stops receiving messages in its local phase δ_{i_ℓ}.

3. For every pair of non-faulty processors (P_j, P_i) $(j \neq i)$, if P_j sends a message in its local phase δ_{j_k} and P_i received it in its local phase δ_{i_ℓ}, then if P_i sends a message in its local phase δ_{i_ℓ}, P_j will receive it in its local phase δ_{j_k}.

Observe that since this definition applies for every pair of processors, it follows that if a system is pair-wise synchronized, then after a processor sends a message, no other processor can send in any of their subsequent local phases, until all of the processors finish receiving messages in their current local phase.

Definition 7. We consider a pair-wise synchronized system to be a *globally synchronized* system if for every pair of non-faulty processors (P_j, P_i), it holds that $\theta_i[k] = \theta_j[\ell]$, whenever a message sent from P_i at P_i's local phase δ_{i_k} was received by P_j at P_j's local phase δ_{j_ℓ}.

Definition 8. We consider a distributed system to be *simulated self-Stabilizing synchronous* if it complies with the following conditions:

1. Convergence - Starting from any arbitrary system state, the system will be able to **deterministically** converge to a globally synchronized system within a bounded time.

2. Closure - If at time t_0 the system is globally synchronized, then the system is globally synchronized for every time $t \geq t_0$.

Problem Definition: These definitions lead us to the definition of the central problem addressed in this thesis, specifically: Given a distributed system S, a communication interface layer must be constructed that turns S into a simulated self-stabilizing synchronous system.

3.3 Drift handling in the System

As stated, there exists an upper bound d_{trans} on message transmission duration. We allow every processor in the system to use this bound as part of executing the protocol's instructions. However, since every hardware clock has its own particular internal drift, measuring d_{trans} with the hardware clock is not suitable. However, this internal drift can be bound from above by an constant, d. This constant is called the drifted transmission bound.

Definition 9. We consider $d \in R_0^+$ to be a *drifted transmission bound* in a semi-synchronous system if for every processor P_i with the corresponding hardware clock H_i, it holds that if P_i sends a message at time $H_i(t)$ to another processor P_j then one of the following applies:

1. P_j will accept the message strictly before $H_i(t) + d$.

2. P_j will not receive the message from P_i (this option exists only if at least one of them is faulty).

Remark 10. Note that this definition does not prevent an adversary from instructing faulty processors to process the accepted message at any given time.

Lemma 11. *For any semi-synchronous system with maximum drift rate α, $d = \alpha d_{trans}$ can be used as a drifted transmission bound for that system.*

Proof. For every processor P_i, if P_i sends a message at $H_i(t_{send})$, then it will either reach its destination or get lost before $t_{send} + d_{trans}$. Since d is defined

by multiplying the upper bound on transmission time by the maximum drift rate, we are guaranteed that $H_i(t_{send}) + d > H_i(t_{send} + d_{trans})$. Thus, either the message is indeed accepted by the recipient processor, or it gets lost within d global clock units. $\square$

Since the drifted transmission bound constant is measured locally by every processor, we want to use a constant global bound on these local measurements, in order to evaluate the global system's properties (e.g. convergence time). We will again use the maximum drift rate to define this bound, as follows: $d_{analysis} = \alpha d = \alpha^2 d_{trans}$.

Chapter 4

Simulating a Fail-Stop Fault Model over an Omission Fault Model

In this section, we will show how to simulate a system with fail-stop faults over a system with omission faults in a semi-synchronous system. The consequence of such a simulation is that for any given protocol φ that achieves a certain property in a system with fail-stop faults, we can achieve the same property in a system with omission faults by running φ over the emulated fail-stop system.

4.1 The FS Simulation Problem

In order to formally define a simulation we must first define the interface through which the system interacts with the external world, i.e. with the users of the system, or with upper layers.

Definition 12. For a distributed system S, we mark the set of possible inputs for this system as $in(S)$ and the set of possible outputs as $out(S)$. Each input or output element has a type, a recipient and maybe also some associated data. We define the union of $in(S)$ and $out(S)$ as the *system interface* of S and we refer to it as API(S).

Definition 13. A sequence σ of elements from the interface (API(S)) of a given distributed system S is considered an *allowable sequence* if it is allowed

by S, i.e. by the properties of S's communication network, fault model, *etc.* We mark the set of all allowable sequences over S as $seq(S)$.

Definition 14. We say that a protocol φ is a *simulation protocol* that simulates a distributed system S_1 over another distributed system S_2, if it satisfies the following:

1. φ uses S_2.
2. The interface of φ is $API(S_1)$.
3. For any sequence σ_{in} of inputs from $in(S_1)$, there exists an allowable sequence $\sigma \in seq(S_1)$, such that σ_{in} is a subsequence of σ, and the sequence of elements that φ produces from $API(S_1)$ is σ.

In other words, running with the simulation protocol over S_2 has the same sequence of events in the external world as running on top of S_1.

A simulation protocol φ runs on every processor in the system as a layer between the processor P and the system S_2. The input elements from API(S_1) are sent from P to its instance of φ as part of the upper layer protocol, and the outputs of φ from API(S_1) are sent back to P. Throughout this thesis, when we refer to the processor that runs the instance of φ we will call it the *host* processor.

In order to define the fail-stop simulation we need the next definitions.

Definition 15. A processor is *active* in the fail-stop simulation if it creates output elements for its host, and sends the host's input elements to the other processors.

A processor is *inactive* if it does not create output elements for its host, and does not send the host's input elements to the other processors.

Definition 16. An interface is an *emulation fail-stop interface* if it contains a predefined message that is marked as $\langle BTA \rangle$ (back to active). $\langle BTA \rangle$ is a message sent from the simulation layer to the host processor, and indicates to the host processor that its simulation layer has returned to being active after a period of time in which it was inactive.

Definition 17. Given a protocol φ with a bounded runtime τ_{off} that runs on a semi-synchronous system S, we say that φ's run over S is an *emulated fail-stop run* if the following condition holds:

For every faulty processor P_{fault}, if P_{fault} does not act according to φ's rules at its mth local phase, it will be inactive for every local phase that is larger or equal to $m+1$, until the end of φ's run. Every non-faulty processor has no constraints on sending or receiving messages. Accepting a $\langle BTA \rangle$ message during the run does not prevent it from being considered as an emulated fail-stop run.

More formally, a run of φ is an emulated fail-stop run, if for every sequence, $\sigma \in seq(S)$ $\sigma = (e_1, e_2, ..., e_i, e_{i+1}, ...)$ that was produced by φ, for each $e_i \in \sigma$ that does not comply with φ's rules, and was created by P_{fault} in its mth local phase. Then for any $e_j \in \sigma$ $(j \geq i+1)$, e_j was either created by a processor other than P_{fault}, or was created by P_{fault} in its mth Local Phase. Every non-faulty processor has no constraints on the elements that this processor creates.

Note that the definition above does not preclude a faulty processor from sending and receiving messages that are not part of running φ.

We now want to expand the emulated fail-stop run over a semi-synchronous system to be a property of the system for a set of protocols.

Definition 18. Given the constant τ_{off}, a semi-synchronous system S is considered an *emulated fail-stop system*, if for every protocol φ with runtime bounded by τ_{off}, φ's run over S is an emulated fail-stop run.

The Fail-stop Simulation Problem: We are given a distributed semi-synchronous system S subjected to omission faults and with a drifted transmission bound d. Given the constant τ_{off}, one must construct a simulation protocol that, after a bounded time from the initial transient faults, simulates an emulated fail-stop system with τ_{off}. The simulated system is required to keep the semi-synchronous property with another drifted transmission bound $(\bar{d}_{trans})$.

In the following section we present a protocol that solves the fail-stop simulation problem.

4.2 The FS Simulation Protocol (FSSP)

We will refer to the protocol that uses the simulation protocol (the host) as φ. φ represents any protocol that has a runtime bounded by τ_{off} and is assumed to run over an emulated fail-stop system. We mark as e every API element that is being used by φ in order to interact with the simulation layer.

Next, we will present the FSSP by describing the assumption on each processor, the messages that are being passed during its execution, and the set of rules that instructs each processor.

4.2.1 Processors' Variables

Every processor in FSSP has the following variables:

- Set of timers:
  1. T_{2d}: Timeout timer, that elapses after $2d$.
  2. T_{3d}: Timeout timer, that elapses after $3d + \varepsilon_{min}$ (see Definition 3.1.2).
  3. T_{off}: Timeout timer, that elapses after τ_{off}.
- An elements queue (FIFO), where the input elements from φ are stored until they are handled.
- A parity bit that is used to distinguish between identical input elements that were sent one after the other. Each time the processor handles the next input element it flips the parity bit.
- A local array of size N, which is called LatestElements and contains the latest input elements (including the corresponding parity bit) that were received from the various processors and a timestamp indicating when each one was received.

4.2.2 Protocol's Messages

In order to ensure that faulty processors will disable themselves in case they do not receive part of the messages in the system, every processor needs to send two messages for each input element it receives from φ, one in order to update all other processors and the other in order to verify that the message was indeed accepted. This poses the intuition for the different types of messages in the protocol.

The protocol uses the following types of messages:

1. Element message $\langle e, b \rangle$ - A message that contains an input element e, with its corresponding parity bit b. We distinguish between two types of element messages:

(a) Update message $\langle e, b \rangle$ - If it is the first time that this input element is being sent in FSSP.

(b) Verification message $\langle e, b \rangle$ - If this element was previously sent as an update message.

2. Empty message $\langle 0 \rangle$ - Used when the elements queue is empty. This message is sent in order to verify that every processor will have to receive eco messages even if its queue is empty.

3. Eco message $\langle E, i \rangle$ - When a processor receives an element message $\langle e, b \rangle$ or an empty message $\langle 0 \rangle$ from another processor i, it sends the eco message $\langle \text{LatestElements}, i \rangle$ to all other processors.

For describing the protocol we make use of the following definitions:

Definition 19. An element in LatestElements is considered to be an *UpToDateElement* if it was updated in the last $3d$.

Definition 20. A received message is considered a *RenewalMessage* if it contains at least one pair of values $\langle e, b \rangle$ associated with a certain processor i, where the receiver's i^{th} entry in LatestElements (the corresponding pair) is not an UpToDateElement and is different from $\langle e, b \rangle$.

Specifically:

- An element message $\langle e, b \rangle$ from processor i is considered to be a RenewalMessages, if the ith entry in the receiver's LatestElements is not an UpToDateElement and contains an element that differs from $\langle e, b \rangle$.

- An eco message $\langle E, i \rangle$ is considered to be a RenewalMessages if E has at least one element $\langle e, b \rangle$ at some entry i, where the ith entry in the receiver's LatestElements isn't an UpToDateElement and it differs from $\langle e, b \rangle$.

4.2.3 FSSP Rules

When a processor becomes inactive, it resets T_{off} and stops sending and receiving elements from φ. In addition, it clears all existing messages in its elements queue. Although the processor is inactive, it keeps accepting the FSSP messages from other processors, and sends eco messages according to FSSP.

After T_{off} elapses the processor switches back to active and returns to send and receive elements from φ. Once a processor switches to active mode it sends a $\langle BTA \rangle$ message to its host (φ).

For a given element e, we call the action of sending an element message $\langle e, b \rangle$ to all other processors in the system a *sending round*. A sending round ends when a new sending round starts. A sending round that starts with an update message is called an update sending round, and a sending round that starts with a verification message is called a verification sending round.

FSSP Protocol:

Every processor acts according to the following rules:

- For every received RenewalMessage:
  - Update LatestElements for elements that are not UpToDateElements.
  - If the updated element was addressed to your host and you are active, forward it as an output element to your host.
- If an element message $\langle e, b \rangle$ from processor i was received, send an eco message $\langle \text{LatestElements}, i \rangle$ to all other processors.
- When T_{3d} elapsed:
  1. Reset T_{3d} & T_{2d}.
  2. If the previous sending round was an update sending round with $\langle e, b \rangle$, start a verification sending round with the same pair ($\langle e, b \rangle$).

     If the previous sending round was a verification sending round with $\langle e, b \rangle$, flip the parity bit, and start an update sending round with the next input element $\bar{e}$ from the elements queue $\langle \bar{e}, \neg b \rangle$.

     If the queue is empty, send an empty element message $\langle 0 \rangle$.
- When T_{2d} elapsed: If you received less than $(N - F)$ eco messages that contain your latest input value with your ID, change your status to be inactive.

Note that every modified element in LatestElements can only be forwarded once as an output element to the host for every pair of update and verification sending rounds, since the modification can only happen once every update sending round as will be shown in Lemma 23.

Remark 21. In the current protocol, the runtime of φ (τ_{off}, resp.), needs to take into consideration that every sending of an element in the system will take at least $6d + 2\varepsilon_{min}$ ($2T_{3d}$). It is possible to overcome this limitation in several ways, such as:

1. Use several instances of this protocol in a pipeline.
2. Enlarge LatestElements to be at the size of N^2.
3. Cluster all elements sent by the processor in each round to a single element.

We focus on the simpler version in the cost of the $2T_{3d}$ latency for every sending round.

Next, we will prove that the above protocol poses a solution to the fail-stop simulation problem.

Lemma 22. *After $7d + \varepsilon_{min}$ from the end of the transient faults, all API elements that are sent or received by FSSP are not effected by the transient faults.*

Proof. After $3d+\varepsilon_{min}$ from the end of the transient faults, T_{3d} had elapsed for every processor in the system. This means that every active processor sent an element message that was either received from its host's elements queue or an empty element (if there were no elements in the queue). Therefore, all input elements are not affected by the transient faults.

On the other hand, within $3d$ from the end of the transient faults, all the elements that were in LatestElements when transient faults ended are no longer UpToDateElements. After additional $3d + \varepsilon_{min}$, T_{3d} will elapse in at least one of the processors causing it to send a valid element message. When a valid RenewalMessage is received (within d), the modified element is forwarded as an output element, where this output element is not affected by the transient faults. □

All claims from now on refer to the system states later than $7d + \varepsilon_{min}$ from the end of the transient faults.

Lemma 23. *For any processor P_i that started a sending round at t with $\langle e, b\rangle$, during $[t,\ t + 3d + \varepsilon_{min}]$ every processor can only update P_i's value to be $\langle e, b\rangle$.*

Proof. When P_i sends the element message $\langle e, b \rangle$, it means that for this sending round that will last $3d + \varepsilon_{min}$ (T_{3d}), it will not send any element message other than $\langle e, b \rangle$. All of the other processors will receive its element message within d. Every processor that received the element message will send an eco message that will arrive to the receivers within $2d$ (it might be much less but at most $2d$). So, we can conclude that as long as any other processor can still receive an eco message that contains $P_i's$ new value, P_i will not send another element, because this sending round has not ended yet. Only after all eco messages are accepted ($t+3d+\varepsilon_{min}$) may P_i send the next element message.

It is possible that eco messages that contain the previous value of P_i as part of their LatestElements exist during $[t,\ t + 3d + \varepsilon_{min}]$ (because they did not receive $P_i's$ update message yet). These messages will not update the receiver's LatestElements because either the receiver processor did not get the update message $\langle e, b \rangle$ and hence will not consider the eco message as a RenewalMessages. On the other hand, if the receiver processor got the update message and updated the element, then this element is considered an UpToDateElement and will not be changed by this eco message. □

Lemma 24. *If at a certain point in time t a processor P_i sends an update message $\langle e, b \rangle$, every processor that receives this message will include $P_i's$ updated value in every eco message it sends during $[t + d,\ t + 6d + 2\varepsilon_{min}]$.*

Proof. Every processor that received the update message $\langle e, b \rangle$ accepted it before $t + d$. From this point on, $P_i's$ new value is updated in each of the receivers' LatestElements, and therefore is part of every eco message they will send. According to Lemma 23 P_i will not send another element message until $t+3d+\varepsilon_{min}$. Then according to the protocol's rules it will send the same value in a verification sending round. Then again, according to Lemma 23 we know that this value will not be changed until $t + 6d + 2\varepsilon_{min}$. Therefore, we are assured that all processors that received this message will keep this value as part of their eco messages, as claimed. □

Lemma 25. *If at a certain point in time t a processor P_i sends an update element $\langle e, b \rangle$ to $N - F$ processors, every processor will either update $\langle e, b \rangle$ in its* LatestElements *or will change its state to inactive before $t + 6d + 2\varepsilon_{min}$.*

Proof. According to Lemma 24 if P_i sends an update element $\langle e, b \rangle$ to $N - F$ processors, then during $[t + d,\ t + 6d + 2\varepsilon_{min}]$, every one of them will contain

$P_i's$ updated value as part of its eco messages. Thus, every set of $N - F$ Eco messages must contain at least $N - 2F$ eco messages that contain $P_i's$ updated value.

According to the protocol's rules, all processors must start a sending round every $3d+\varepsilon_{min}$ (when T_{3d} elapses), and every processor needs to accept $N-F$ eco messages within $2d$ of the beginning of the sending round in order to stay active. We can conclude that in order to stay active, every processor needs to get $N - F$ eco messages once every $5d + \varepsilon_{min} < 5d + 2\varepsilon_{min}$. Since the time window $[t+d,\, t+6d+2\varepsilon_{min}]$ is longer than $5d+\varepsilon_{min}$, we know that every processor that did not get $N - F$ eco messages during that period, must have changed its status to inactive. Therefore, every processor that stayed active must have got at least $N - 2F > 1$ eco messages that contain $P_i's$ updated value, and thus updated its LatestElements accordingly. □

Lemma 26. *For every processor P_i that sends an element message $\langle e, b\rangle$, one of the following must hold:*

- *Every active processor will updated $\langle e, b\rangle$ for P_i in its* LatestElements*.*
- *P_i will become inactive within* $2d$ *(*i.e.,T_{2d}*).*

Proof. If P_i sends its new element to at least $N - F$ processors, according to Lemma 25 all active processors will also update this element in their LatestElements. But, if P_i did not send its message to $N - F$ processors, it will not get an eco message from any of them and will become inactive when T_{2d} elapses. □

Lemma 27. *After* $13d + 3\varepsilon_{min}$ *from the end of the initial transient faults, FSSP simulates a system with emulated fail-stop faults over a system with omission faults.*

Proof. According to Lemma 22 after $7d+\varepsilon$ all elements in the system are not affected by the transient faults. According to Lemma 26 a processor whose element was not received by all other active processors will become inactive within $2d$. In addition, according to Lemma 25, for every processor P_i that sends a message and is still active, every other processor that fails to receive P_i's messages will become inactive within $6d + 2\varepsilon$.

We must note that a processor that becomes inactive, stays inactive until T_{off} elapses, which means at least as long as the predefined constant τ_{off}. That means that an element that does not follow the protocol's rules will cause

the element creator not to produce any other elements for τ_{off}, which causes the system's interface to comply with the emulated fail-stop fault model. □

Lemma 28. *$2\tau_{\mathit{off}} + 13d + 3\varepsilon_{min}$ after the end of the transient faults, every run of φ on the system is an emulated fail-stop run.*

Proof. Even if some non-faulty processors were inactive as part of the transient faults, eventually after T_{off} elapses they will return to be active. Non-faulty processors will never become inactive, since inactive processors keep sending eco messages. Therefore, we are guaranteed that all non-faulty processors will always get enough eco messages. Thus, according to Lemma 22 $\tau_{\mathit{off}} + 7d + \varepsilon_{min}$ after the end of transient faults all non-faulty processors return to being active. Since the emulated fail-stop protocol φ runtime is bounded by τ_{off}, it is possible that when all non-faulty processors got back to be active, φ is in the middle of its current run. But, after $2\tau_{\mathit{off}}$ every instance of φ will have an emulated fail-stop run on the system, which is $2\tau_{\mathit{off}} + 6d + 2\varepsilon_{min}$ after the end of the transient faults. □

Lemma 29. *FSSP keeps the semi-synchronous property with the drifted transmission bound $\bar{d}_{trans} = 6d + 2\varepsilon_{min}$.*

Proof. According to the protocol's rules, a processor will not send an update element message until the end of the second (verification) sending round. This means that a processor sends a message only once every $6d + 2\varepsilon_{min}$. Therefore, $6d + 2\varepsilon_{min}$ can be used as the upper bound for processing and sending duration between every pair of processors in the simulated protocol. □

So, as claimed, this protocol simulates a semi-synchronous system with emulated fail-stop faults over a system with omission faults.

Lemma 30. *For every non-faulty processor, once it is in active mode, it will not become inactive.*

Proof. Note that all processors (even those that are inactive) keep sending eco messages. Thus, for every non-faulty processor P_{good} that sends an element message $\langle e, b \rangle$, P_{good} is assured that all non-faulty processors will receive this message within d. Therefore, all of them (even those potentially inactive) will send to P_{good} an eco Message $\langle E, i \rangle$ with the updated entry and with P_{good}'s ID. By accepting these messages, P_{good} will not change its status to inactive, as claimed. □

4.3 FSSP as an Emulation layer

We presented a protocol that simulates an emulated fail-stop system. Next, we'll want to use it as a layer where we can run protocols with a bounded runtime. Therefore we call this layer, the *emulation* layer. Few remarks regarding this layer:

- Given an emulation layer that has several layers above it, where each layer has separated attributes in the API elements, it is possible for the emulation layer to cope with different values of τ_{off} for each of the layers. If a processor violates the rules of one of the layers, it will be inactive for all the layers above the violated layer for the τ_{off} that corresponds to the violated layer. However, the layers below the violated layer will continue to act as usual. In this way we can promise a faster convergence in the lower layers even if the upper layer requires a higher convergence time.

- We refer to protocols that are time bounded and therefore used τ_{off} as a bound of the protocols runtime. We can have a discrete version of the simulation protocol if we refer to protocols that are bounded by the amount of local phases. In this case, τ_{off} will be a bound on the number of the local phases in the protocol.

Chapter 5

Adjusting a Semi-Synchronized System to a Pair-Wise Synchronized System in the Emulated Fail-Stop Model

In this section, we present a protocol that given a semi-synchronous distributed system subjected to emulated fail-stop faults, can achieve the pair-wise synchronized property.

In order to simulate a pair-wise synchronized system, we first need to define allowable sequences for such systems (Definition 31). For every API element we add a logical attribute in order to get a clearer definition of the allowable sequences. We mark the API with the additional attributes as a pair-wise synchronized interface.

Pair-Wise Synchronized Interface: An interface, where the following logical attributes are added to the API elements:

- For an input element, add: (sender's current local phase (SCLP)).
- For an output element, add: (receiver's current local phase (RCLP)).

Note that the SCLP and RCLP are only logical fields used for the definition and need not be sent as part of the element, since the sender knows its SCLP and the receiver knows its RLCP without retrieving them from the API element.

Using these attributes, we can define the pair-wise synchronized allowable sequences. Informally, for every pair of processors P_{send} and P_{rec}, with Local phases k, ℓ, respectively (i.e. P_{send} sent an input element in its local phase k, and it was received by P_{rec} in its local phase ℓ), we require that P_{send} will not create an input element of a local phase that is greater than k as long as P_{rec} keeps getting output elements for its local phase ℓ.

Definition 31. Let S be a semi-synchronous distributed system. A sequence $\sigma \in seq(S)$ is a *pair-wise synchronized allowable sequence*, iff for every quadruplet of elements in $\sigma = (e_i, e_j, e_\ell, e_k)$, where $i > j > \ell > k$, and for which the following holds:

1. e_i and e_ℓ are input elements that were produced by the same processor P_{send};
2. e_j and e_k are output elements assigned to some other processor P_{rec};
3. e_i was received by P_{rec} as the output element e_j;
4. $\mathrm{e}_j.\mathrm{RCLP} = \mathrm{e}_k.\mathrm{RCLP}$ (another output element that was received by P_{rec} in the same local phase);

then $\mathrm{e}_\ell.\mathrm{SCLP} = \mathrm{e}_i.\mathrm{SCLP}$.

Pair-Wise Synchronized Simulation Problem: Let S be a semi-synchronous distributed system with a drifted transmission bound d subjected to emulated fail-stop faults. We assume there exists a predefined constant τ for the local phase duration (see Definition 6). One must construct a protocol, ψ, that after a bounded time from the stabilizing of the fail-stop emulation layer, ψ will simulate a pair-wise synchronized system with local phases of length τ over S.

In other words, for any set of inputs of API(S), ψ will generate a set of output elements of API(S), such that the received sequence of elements σ, is a pair-wise synchronized allowable sequence.

In order to accomplish this goal, we make use of another clocking protocol PWCP (Pair-Wise Clocking Protocol), which is presented in the next subsection. PWCP receives d and τ as inputs, and it is assumed to run over a fail-stop system and has a bounded runtime. We define the local phases of PWCP to be a sequence of windows of size d. Since the system is an emulated fail-stop system, for every instance of PWCP, within d after a processor violates the PWCP rules it ceases to participate in the current PWCP

instance. Thus, every instance of PWCP can be considered as executing on a fail-stop system.

We instruct the protocol ψ to run a sequence of instances of PWCP, such that the starting system state of the PWCP instance is the ending system state of the previous PWCP instance. In the next subsections we will present PWCP and prove its properties. We will also show how it can be used by ψ in order to solve the pair-wise synchronized simulation problem.

5.1 The Pair-Wise Clocking Protocol (PWCP)

We will now present PWCP. Given a drifted transmission bound d and the predefined constant τ, we construct a self-stabilized protocol with a bounded runtime, that can produce ticks beneficial to indicate the appropriate time to send and stop receiving messages in every local phase. Formally, the protocol requirements are:

- Self-stabilization, as expressed by the following properties:

  PWCP has an internal set of system states (see Definition 35) C, such that the following properties hold:

  1. Steadiness: If the initial system state of PWCP is $c \in C$, then the system state when PWCP finishes its run is $\bar{c} \in C$.
  2. Convergence: If the initial system state of PWCP is $c \notin C$, then after PWCP completes its run, the system state of PWCP is $\bar{c} \in C$.

- Local phase ticks: the protocol needs to provide to each of the processors two types of signals:

  1. Send tick: When the host can start its sending session.
  2. Stop receiving tick: When the host should stop receiving messages of the current sending session.

The motivation for these properties is that after the emulation layer stabilizes, by PWCP's convergence property, any of PWCP's instances will end in a safe system state. Therefore, the system state in the beginning of the next PWCP will also be in C. According to PWCP's steadiness property, each of the following instances of PWCP will also start and finish in a system states that belongs to C. Thus, running sequences of PWCP will result in

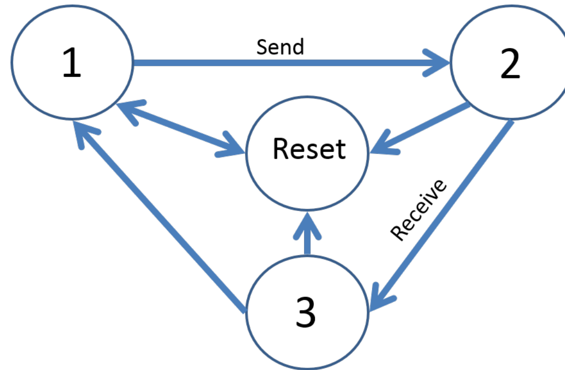

Figure 5.1: The PWCP Abstract State Machine

a traditional self-stabilized protocol. Using the convergence property, also implies a bound on the protocol runtime.

After the system stabilizes, it will produce pairs of clock ticks that allow the processors to send and receive their messages, while ensuring that the pair-wise synchronized property (see Definition 6) holds.

Notations Throughout this section we make use of the following notations:

1. $D = \frac{\tau}{3} - \varepsilon_{min}$ (assuming that $2d < \frac{\tau}{3} - \varepsilon_{min}$).
2. $X + 1 = \mod(X, 3) + 1$.
3. $X + 2 = \mod(X + 1, 3) + 1$.

5.2 Processors' Variables

- Every processor remembers the last message it received from every other processor, and when it was received.
- Every processor has a personal instance of the abstract state machine appearing in Figure 5.1.
- Every processor has the following set of timers:
    1. ϕ: The time elapsed since the processor arrived to the current state.

2. T_d: Timeout timer, that for some $\varepsilon_1 < \varepsilon_{min}$, elapses after $d + \varepsilon_1$.
3. T_D: Timeout timer, that elapses after $D + \varepsilon_{min}$.

5.3 Protocol Messages and Definitions

The protocol uses two types of messages:

1. State message <at X>: Sent to all others to notify them of your current state.

2. Move message <moved to X>: Notifies of a transition from the former state to a new state X.

Definition 32. For a certain processor P in state X, we say P is *Ready to move to* $X + 1$, if all the accepted messages in the memory of P are either in X or $X + 1$ and at least one of them (including a message from itself) is from state X.

Definition 33. For a certain processor P in state X, we define a *progress move* as a move to $X+1$ message, which was generated when all the accepted messages in the memory of P were from its current state (X).

Definition 34. For a certain processor P in state X, we define a *union move* as a move to $X+1$ message, which was generated when some of the accepted messages in the memory of P were from the following state ($X + 1$).

Definition 35. For a certain point in time $\tilde{t}$, we say that the system is in a *steady state* if it holds that in the time window $[\tilde{t}, \tilde{t} + D + \varepsilon_{min}]$ the following conditions hold:

1. For every non-faulty processor P that was in state X at $\tilde{t}$, P moved to state $X + 1$ until $\tilde{t} + D + \varepsilon_{min}$.

2. All non-faulty processors were in the same state for at least $\bar{\varepsilon}$.

3. All move messages that are in transit are for at most two consecutive states ($X + 1 \& X + 2$).

4. All state messages that are in transit are for at most two consecutive states ($X \& X + 1$).

We define **the set of safe system states C** to be all system states that apply the system's steady state.

Definition 36. For a certain message M, we define this message as *clean* if M was generated after the system is assured to remove all transient messages, and was created according to the protocol's rules.

A *dirty* message is a message that was created as part of the initial random noise, or a message that was generated according to the protocol's rules but before the time where there can't be any transient message.

Notation. We mark the following set of actions for a processor in state X as a *move action*:

1. Move to state $X + 1$.
2. Send a move message $<$moved to $X + 1>$.
3. Reset all timers.

5.4 PWCP Protocol

We now present the PWCP protocol. Throughout this protocol we mark the processor's current state as X.

PWCP(d,τ): Every processor acts according to the following rules:

- If a $\langle BTA \rangle$ message (see Definition 16) was received from the emulation layer, go to the Reset state.
- Always remove messages older than $2d + \varepsilon_1$ from the memory.
- If $X \neq Reset$, act according to the following order:
  1. If there exists a set of messages in the memory from states 1,2 and 3, move to the Reset state.
  2. If a move message $<$moved to $Y>$ is received and $Y = X + 1$:
     (a) Move to state Y.
     (b) Send a state message $<$at Y$>$.
     (c) Reset all timers.

3. If T_D elapsed and "Ready to move to $X+1$" applies, then make a move action.
4. If all of the following conditions hold make a move action:
   (a) A state message was received or removed.
   (b) $\phi > D$.
   (c) "Ready to move to $X+1$" applies.
5. If T_d elapsed:
   (a) Send a state message <at X>.
   (b) Reset T_d.

- If $X = Reset$, and one of the following holds:
  1. If a message is removed from memory ($2d$ after it was received), such that no messages are left in the memory:
     Perform the following actions:
     (a) Move to state 3.
     (b) Send a state message <at 3>.
     (c) Reset all timers.
  2. If "Ready to move to 1" applies:
     Perform the following actions:
     (a) Move to state 1.
     (b) Send a state message <at 1>.
     (c) Reset all timers.
- When moving from state 1 to state 2, send a send tick to the host.
- When moving from state 3 to state 1, send a stop receiving tick to the host.

5.5 Claims and Proofs

In this subsection, we will show that from any arbitrary initial state, the system converges into one of the safe system states within a constant time. In order to prove this property, we use the following set of lemmas.

Lemma 37. *$3d + \varepsilon_1$ after the stabilizing of the fail-stop emulation layer, all accepted messages in the system are clean messages.*

Proof. From any arbitrary state, after $2d + \varepsilon_1$ the memory of all processors no longer contains noisy data from the transient faults. This means that only messages that were sent according to the protocol rules (even if they are dirty) exist in the processors' memories. Therefore, all generated messages from this point on are clean messages.

Dirty messages that were generated before $2d + \varepsilon_1$ had passed, will reach their destination within d. From that point on (after $3d+\varepsilon_1$), every accepted message is a clean message. □

We mark the point in time where all the sent messages are clean as t_{clean}.

Lemma 38. *For every processor P that is not in the Reset state at time $t_0 > t_{clean}$, P must change its state before $t_0 + 4d + D + \varepsilon_{min}$.*

Proof. We mark P's current state as X. Let us assume in contradiction that P stays at X until $t_0 + 4d + D + \varepsilon_{min}$. At $t_0 + d$ all other processors have received $P's$ message (we assume it is active until $t_0 + 4d + D + \varepsilon_{min}$). Thus, "Ready to move to $X + 2$" is not enabled from $t_0 + d$ and on. This implies that from $t_0 + 2d$ and on, no <moved to $X + 2$> message can be accepted or generated, which means that no processor from state $X + 1$ can move to state $X + 2$.

Regarding processors in the Reset state, there exist two options to move out of Reset:

- Moving to state $X + 2 = 1$ if "Ready to move to 1" applies - Since "Ready to move to $X + 2 = 1$" is not enabled, no processor from the Reset state can move to state 1.
- Moving to state $X + 2 = 3$ if no message remains in the processor's memory - Since P is active for more than d in state $X = 1$, every processor in Reset must have received a message from it. Therefore, this option is not possible.

Before $t_0 + D + \varepsilon_{min}$ T_D elapses for P. Since P did not perform a move action (according to our assumption) until $t_0+D+\varepsilon_{min}$, it means that "Ready to move to $X + 1$" is not enabled for P. This implies that another processor was at state $X + 2$ for the last d. We mark the processor at state $X + 2$ as Q. We showed that Q could only move to $X + 2$ until $t_0 + 2d$.

Before $t_0 + 2d + D + \varepsilon_{min}$ if Q is still at $X + 2$, T_D will elapse for Q. Therefore, we consider the following scenarios:

- Q moved to state X - in this case, within d P will get $Q's$ updated message and will move to $X + 1$.
- Q is no longer active or moved to Reset state - in these cases, within $2d$ $Q's$ messages will be removed from $P's$ memory and P will move to $X + 1$.
- "Ready to move to X" does not apply for Q - this means that another processor exists in state $X + 1$. But, since Q must be at $X + 2$ before $t_0 + 2d$, it must have a message of state X sent from P. Therefore, Q will move to the Reset state (which was handled in the previous item).

Thus, until $t_0 + 4d + D + \varepsilon_{min}$, P will have to change its state (move forward or switch to Reset) as claimed. □

Notations:

- We mark $\vartheta = 4d + D + \varepsilon_1$ and call this time duration the *Static Bound*.
- Timing Notations:
  1. We mark some point in time as $t_1 > t_{clean}$.
  2. $t_2 = t_1 + d$.
  3. $t_3 = t_2 + \vartheta$.
  4. $t_4 = t_3 + 3d + D + \varepsilon_{min} + \varepsilon_1$.

We will now prove that the creation of a <moved to 2> message is obligated in the system.

Lemma 39. *A clean move message will be generated before t_4.*

Proof. As mentioned in Lemma 37, all messages that were accepted after t_1, are clean messages. Let us assume by contradiction that no move message was generated in the system before t_5. According to our assumption, no clean move message was generated during $[t_1, t_5]$, and therefore, during $[t_2, t_5]$ no move message was accepted. According to Lemma 38, since $t_3 - t_2 = \vartheta$ during $[t_2, t_3]$ every processor that is not in Reset must change its state. Since there exists an active processor that is not in Reset state a move from Reset to

state 3 is not enabled. By the protocol's rules a processor that moved to state 2 or 3, must have sent or received a move message. Thus, if no move message was generated it means that all processors are in state 1 or in Reset state.

If all processors are in Reset state, within $2d + \varepsilon_1$ at least one of them will remove all the messages in its memory and will move to state 3. For every processor in Reset that received the state message of state 3, "Ready to move to 1" applies for it, and it will move to state 1. For every processor in Reset that did not receive the state message of state 3, it will remove all messages in its memory and will move to state 3. Either way, after another $D + \varepsilon_{min}$, T_D will elapse for all processors. If there exist processors in states 1 and 3, then those in state 3 will create a clean move message. If there is no active processor in state 3, it means that all processors that are in state 1 they will create a clean move message.

We can conclude that before t_4, a clean move message will be generated as opposed to our assumption. □

Lemma 40. *For a certain point in time $t_5 > t_{clean} + 3\vartheta + 4d + 2D + \varepsilon_{min} + \varepsilon_1$, a clean <moved to 2> message will be generated before t_5.*

Proof. We mark two additional points in time:

1. $t_6 = t_5 - (\vartheta + 4d + 2D + \varepsilon_{min} + 2\varepsilon_1)$.

2. $t_7 = t_6 + D + \varepsilon_1$

Since $t_6 > t_{clean}$ and according to Lemma 39 a clean move message will be generated before t_7.

We will cover the three possibilities for this move message:

1. <moved to 2> - In this case the lemma holds.

2. <moved to 3> - In this case, there must be a processor in state 2 that generated this message. We will mark this processor as P_2. According to Lemma 38, at least once every ϑ a processor that is not in Reset must change its state. That means that P_2 has moved to state 2 from state 1 during $[t_6 - \vartheta, t_7]$. In order for a processor to move to state 2, it must either make a move action from state 1, or receive a <moved to 2> message. Either way, a <moved to 2> message is generated in the system prior to t_7.

3. <moved to 1> - In this case, there must be a processor in state 3 that generates this message. We will mark this processor as P_3.

   - If P_3 moved from state 2: In a similar way to what we have shown in the previous item, P_3 had moved to state 3 from state 2 during $[t_6 - \vartheta, t_7]$, and from state 1 to state 2 during $[t_6 - 2\vartheta, t_7]$. The move from state 1 to state 2, was either caused by a clean <moved to 2> message or caused the generation of such a message.
   - If P_3 moved from Reset state: We mark the time that P_3 moved to state 3 as t_{back}. At t_{back} there were no other processors in any other state (otherwise P_3 would have accepted a message from it). On the other hand, from $t_{back} + d$ and on, processors in Reset had accepted the <at 3> message from P_3 and will move to state 1 since "Ready to move to 1" applies for them. Therefore, when P_3 generated the <moved to 1> message, "Ready to move to 1" applied to any other processor in state 3. Thus, within $t_{back} + d + D + \varepsilon_1$, all processors that are not in Reset will be at state 1. Within another $D + \varepsilon_1$ (t_5), "Ready to move to 2" will apply to some of them and they will generate a clean <moved to 2> message.

To conclude, in all three scenarios a clean <moved to 2> message will be created prior to t_5. □

After proving that a clean <moved to 2> message generation is obligated, we will now prove that it enables the protocol's convergence.

Lemma 41. *If at time $\bar{t}$ a processor (possibly faulty) P generated a clean <moved to 2> message, then no non-faulty processor was in state 3 at that time.*

Proof. In order for P to generate a move message, the following conditions must apply to it:

1. P must be at least $D + \varepsilon_{min}$ in state 1.

2. "Ready to move to 2" applies for P. Thus, all accepted messages in P's memory were from states 1 and 2.

Since all accepted messages in P's memory were from states 1 and 2, we can conclude that no active processor can be in state 3, while staying there for more than d. Since if a processor Q was in state 3 for more than d a message from Q to P of state 3 must have arrived and "Ready to move to $X+1$" would not have applied for P. In order to show that at $\bar{t}$ no non-faulty processor was in state $X+2$, we need to prove that no non-faulty processor moved to $X+2$ during $[\bar{t}-d, \bar{t}]$.

First we will show that processors that were in the Reset state will not move to state 3 during $[\bar{t}-d, \bar{t}]$. Since P generated a move message, it means that it was in state 1 for more than d. Therefore, all other processors have a state message from it before $\bar{t}-d$, including those in Reset. Thus, a processor in Reset cannot move to state 3 because there are no messages in its memory. Even if "Ready to move to 1" applies to a processor in Reset, it can only move to state 1 and not to state 3.

Next we will show that processors that were not in Reset cannot move to state 3 during $[\bar{t}-d, \bar{t}]$. Since P must be at least $D+\varepsilon_{min}$ in state 1, it means all active processors have received a message from P of state 1 before $\bar{t}-(D+\varepsilon_{min})+d < \bar{t}-d-\varepsilon_{min}$. Therefore, for every processor Q that was in state $X+1$ during $[\bar{t}-d-\varepsilon_{min}, \bar{t}]$:

1. "Ready to move to $X+2$" does not apply to Q because of $P's$ message.

2. If Q receives a <moved to $X+2$> message it will switch to Reset, since it has messages from state X received from P, its own message from state $X+1$, and the move message from state $X+2$.

Thus, every processor Q that was in state $X+1$ during $[\bar{t}-d, \bar{t}]$ cannot move to state $X+2$ during these periods of time.

□

In the upcoming lemmas we will show that all processors will share the same state for at least ε_{min} (Definition 3.1.2).

Lemma 42. *If at time $\bar{t}$ a processor (possibly faulty) P generates a clean <moved to 2> message, then within $D+d$, for at least ε_{min}, all non-faulty processors will be either in Reset state or in state 2.*

Proof. According to Lemma 41, there is no processor in state 3 when the move message is generated (at $\bar{t}$). That means all processors that are not in Reset are spread over states 1 and 2. We will show that all processors

in state 1 will move to state 2 before any processor in state 2 can move to state 3. In order to do so, we split the proof into two parts according to the processors in state 1: processors in state 1 that were in Reset during $[\bar{t}-d, \bar{t}]$, and processors in state 1 that were not in Reset during $[\bar{t}-d, \bar{t}]$.

- For every processor Q in state 1 that was not in Reset during $[\bar{t}-d, \bar{t}]$. Since Q was not in Reset for the last d, every processor in the system has a message from Q in its memory. Processors that received $Q's$ message from state 1 have the updated state 1 message as $Q's$ latest message, and processors that have not received $Q's$ state 1 message have a state 3 message as $Q's$ latest message. For each processor R in state 2, R has P's message from state 1 (which was received before P made the move action). If the last message that R received from Q was of state 3 it would have moved to Reset, since it has messages from states 1,2 and 3 received from processors P, R and Q, respectively. If the message that R received from Q was of state 1, "Ready to move to 3" will not apply for R until $Q's$ latest message will either be removed or change to a state 2 message. Therefore, no processor in state 2 can produce a <moved to 3> message as long as Q is still active in state 1.

  On the other hand, since all processors that are not in Reset are in states 1 and 2, "Ready to move to 2" applies for every processor in state 1, which eventually (after T_D elapses) will cause any one of them to move to state 2. We can conclude that all processors that were out of Reset state during $[\bar{t}-d, \bar{t}]$ will in state 2.

- For every Processor S that was in Reset in the last d (during $[\bar{t}-d, \bar{t}]$) we showed that S cannot move from Reset to state 3, which means that S might only move to state 1 if "Ready to move to 1" applies to it. If "Ready to move to 1" applies to it, that means that S has at least one message of state 3 and no messages of state 2. That means that processors that have already moved to state 2 have not been there for more than d before S moved to state 1.

  On the other hand, S can only move to state 1 before $\bar{t}+d$, otherwise it will receive P's message and "Ready to move to 1" will not apply to it. Therefore, S will be in state 1 before $\bar{t}+d$, and the state message <at 1> from S will reach any processor in state 2 before $\bar{t}+2d < \bar{t}+D+\varepsilon_{min}$. Thus, the state message <at 1> from S will reach any processor before

it can send a <moved to 3> message. And after the T_D that belongs to S will elapse, it will join state 2 as well.

Only after ε_{min} from the time when all processors that were out of Reset state are in state 2, may the updated messages of processors that moved from state 1 to state 2 be accepted by a certain processor in state 2, potentially allowing it to move to state 3.

To conclude, within $D + d$, all processors will be in state 2 for at least ε_{min} before the next Progress move is enabled. □

Lemma 43. *Local Steadiness - For a certain point in time $\bar{t}$, if all processors out of Reset state were at the same state for more than ε_{min}, then all active processors will be in state 1 within $3(D + \varepsilon_{min})$.*

Proof. If some processors are not in Reset (we mark their state as X), "Ready to move to $X + 1$" will apply for them within d. This holds because even if some will move to the next state after accepting a move message, "Ready to move to $X + 1$" still applies for all processors in state X. Therefore, within $D + \varepsilon_{min}$, all of the processors will either get a <moved to $X + 1$> message and will move to $X + 1$, or T_D will elapse and they will make a move action to $X + 1$. In both cases, all processors will be in the same state $(X + 1)$ within $D + \varepsilon_{min}$. As long as a processor is in Reset state it will not send any message, so processors in Reset state will not affect the progress of processors that are not in Reset state. Since the state machine is composed out of three states, a round-trip takes $3(D + \varepsilon_{min})$.

Processors in Reset state can only move to state 1 if "Ready to move to 1" applies for them. Let $\bar{t}$ be the time when all processors out of Reset are located in state 3. During $[\bar{t}, \bar{t} + D + \varepsilon]$, processors that were in state 3 might move to state 1, but they cannot move to state 2 (since T_D has not elapsed). On the other hand, within $2d$ all messages that were dent by faulty processors and reached processors in Reset state have been removed from their memories. Thus, at $\bar{t} + 2d$ all processors that were in Reset at $\bar{t}$ will have "Ready to move to 1" apply for them and they will move to state 1, before any processor can move to state 2. We can conclude that all processors will be in state 1 for more than ε_{min}. □

Lemma 44. *Steadiness - Once all the processors are in the same state, the system reaches its steady state.*

Proof. Once all processors are in the same state X, within d from that state "Ready to move to $X + 1$" will apply for them, because even if some will move one state ahead, "Ready to move to $X+1$" still applies for all the rest that did not move. Therefore, within $D + \varepsilon_{min}$, either all of the processors will receive a <moved to $X+1$> message and will move to $X+1$, or T_D will elapse and they will move to $X + 1$. In both cases, we have all processors in the same state for at least ε_{min} within $D + \varepsilon_{min}$.

On the other hand, once the first processor moves to state $X+1$, at least $D+\varepsilon_{min}$ need to pass in order for any processor to proceed from state $X+1$ to state $X + 2$. That is because, in order for the first processor to move to the next state, its T_D must elapse, which can only happen $D + \varepsilon_{min}$ after moving to state $X + 1$.

It also applies that no processor can move to Reset since the processors are spread over at most two states, so the condition to move to Reset state cannot apply for any of them.

Note that all four conditions required for the system steady state (Definition 35) hold by using $\bar{\varepsilon} = \varepsilon_{min}$:

1. Once every $D + \varepsilon_{min}$, every processor moves one state ahead.

2. In every time window of $D + \varepsilon_{min}$, all non-faulty processors are in the same state for $\bar{\varepsilon}$.

3. All move messages that are in transit are from either the state that was common to all or from the following state.

4. All state messages that are in transit are from either the state that was common to all or from the following state.

Thus, the PWCP reaches a state that complies with the steady state definition. □

Lemma 45. *Convergence - All non-faulty processors will get to a steady state within $6(D + \varepsilon_{min}) + 8d$ from the stabilizing time of the FS Emulation layer.*

Proof. According to Lemma 37 and Lemma 40 from the stabilizing time of the FS Emulation layer a clean <moved to 2> message will be generated within $3(D + varepsilon_4) + 8d$. According to Lemma 42 within $D + d$ all processors will be in state 2 for ε_{min}, and due to lem:Local-Steadiness this move causes all processors to be in the same state for some $\bar{\varepsilon}$ within $3(D + \varepsilon_{min})$. According to Lemma 44 this brings us to a steady state. □

Lemma 46. *Processors that became active in the Fail-Stop Emulation layer after the system reached a steady state, will not damage the protocol's steadiness*

Proof. A Processor that became active in the Fail-Stop Emulation layer will receive a $\langle BTA \rangle$ message and will move to Reset state. Since we assume the system to be in a steady state, according to Lemma 43 within $3(D+\varepsilon_{min})$ the joined processor will also be in the same state like all others, which means that the Steadiness property holds. Meanwhile, all the rest of the processors maintain the steadiness property, since the processor in Reset state does not send any message. □

Lemma 47. *Processors that became active in the fail-stop emulation layer before the system reached a steady state will not damage the protocol's convergence*

Proof. Throughout the proof of Lemma 45 processors in Reset state do not harm the convergence, and if the steadiness property holds for the rest of the processors out of Reset state, we showed in Lemma 46 that the system also reaches a steady state. □

So far, we proved the Self-Stabilization property of PWCP. Next, we will present the properties of the ticks it generates.

Remark 48. Throughout this section we referred to drifted transmission bound (d) which is measured according to the local hardware clock. But, since the system convergence is a global property, we will bound the convergence time by using $d_{analysis}$ (see Claim 11).

5.6 Ticking Mechanism

Definition 49. We define a *processor synchronized round* for each processor to begin when entering state 1 in the state machine until the next time it reaches state 1, without going through Reset.

According to Lemma 44, after the system reaches its steady state, every non-faulty processor will complete a full synchronized round every $3(D+\varepsilon_{min})$. Since $D+\varepsilon_{min} = \frac{\tau}{3}$, $3(D+\varepsilon_{min}) = \tau$. Thus, the duration of a synchronized round corresponds to the time of the processor's local phase.

Definition 50. For a certain point in time $\tilde{t}$ after the system reaches a steady state, we define for each processor P_i its *processor's simulation phase* ϕ_i to be a counter of processor synchronized rounds that have passed since $\tilde{t}$.

Remark 51. ϕ_i is not known to the processors in the system, and is only used for marking and proving purpose.

In order to present the ticks properties, we will show that processors cannot be too far from one another. In order to formally express this demand, we define a distance function. The basic intuition is that a distance between a pair of states is the minimal number of hops that are needed in order to get from one of the states to the other. In order to attend the wrap around in the state machine (from state 3 to state 1), we always look at two rounds of the state machine (6 states overall). We distinguish between the rounds using the parity of the simulation phase, which brings us to the following definition:

Definition 52. We define the *simulated distance* between two processors x and y with the corresponding state and simulation phase (S_x, ϕ_x) and (S_y, ϕ_y) to be:

$$SDist(x, y) = \begin{cases} \infty & \text{if } x \text{ or } y \in Reset \\ \operatorname{mod}\Big(\{(mod(\phi_x, 2) + S_x) - (mod(\phi_y, 2) + S_y)\}, 6\Big) & \text{otherwise.} \end{cases}$$

Lemma 53. *For any given point in time $\tilde{t}$ after the system reaches its steady state, the simulated distance (with respect to $\tilde{t}$) of every pair of processors is at most 1.*

Proof. By definition, a safe system state implies that all processors will be in the same state for at least ε_{min}, in every time window of size $D + \varepsilon_{min}$. It might be that some moved one state ahead before the others but they will stay there for at least $D + \varepsilon_{min}$ (until T_D elapses) since the first processor moved, and within that time all of the rest will catch up.

Therefore, for every pair of processors {x,y}, one of the following must hold:

1. $\phi_x = \phi_y$ and $\mid S_x - S_y \mid \leq 1$.

2. $\mid \phi_x - \phi_y \mid \leq 1$ and WLOG $(S_x = 1 \wedge S_y = 3)$.

In both cases, SDist is not greater than 1. □

Lemma 54. *For every pair of non-faulty processors {x,y} if the simulation distance between them is at most 1, then processor x will not send a send tick in its $\phi_x + 1$ simulation phase to its host before y sends a stop receiving tick in its ϕ_y simulation phase to its host.*

Proof. When processor x sends a send tick in its simulation phase $\phi_x + 1$, it means that x was in state 1 (of phase $\phi_x + 1$) and moved to state 2. According to Lemma 53 we know that when x moved to state 2, y (and all other non-faulty processors) must have been in state 1 of its simulation phase ϕ_y. Therefore, y had already sent a stop receiving tick in simulation phase ϕ_y, when it moved from state 3 to state 1 (of phase ϕ_y) as claimed. □

Lemma 55. *For every pair of non-faulty processors {x,y}, if the simulation distance between them is at most 1, then processor x will not send a stop receiving tick in its ϕ_x simulation phase to its host before D has passed from the time that y sent a send tick in its ϕ_y simulation phase to its host.*

Proof. When processor x sends a stop receiving tick in its simulation phase ϕ_x, it means that x was in state 3 (of phase ϕ_x) and moved to state 1. According to Lemma 53 we know that when x moved to state 1, y (and all other non-faulty processors) must have been in state 3 of its simulation phase ϕ_y. Therefore, all of them had already sent a send tick in simulation phase ϕ_y, when moved from state 1 to state 2 (of phase ϕ_y), and an additional $D + \varepsilon_{min}$ passed before moving to state 3. Therefore, at least D has passed, since the send tick was sent as claimed. □

Lemma 56. *For a given $\epsilon > 0$, for every non-faulty processor, the duration of time that passes from one send tick to the next send tick is lower than τ.*

Proof. According to Lemma 44, once the system is in its steady state every processor will move one state ahead within $D + \varepsilon_{min}$. Therefore, every processor that generated a send tick at time t_{st1} by moving from state 1 to state 2 must have made the same move within $3(D + \varepsilon_{min})$, which means it generated another send tick within that time. Since $D + \varepsilon_{min} = \frac{\tau}{3}$, then $3(D + \varepsilon_{min}) = \tau$ and the lemma holds. □

5.7 Simulate Pair-Wise Synchronized System Using PWCP

In this subsection we will show how to use the ticks provided by PWCP in order to simulate a pair-wise synchronized system with local phase of length τ, and by that solve the pair-wise synchronized simulation problem. We show how to construct a self-stabilized simulation protocol ψ, that can use PWCP in order to simulate a pair-wise synchronized system (Definition 6) with local phases of length τ.

Thus, ψ needs to align with the following requirements:

1. We mark the time that passes between two consecutive send ticks sent by ψ as T_{st}. For a given $\epsilon > 0$, it holds that $T_{st} \leq \tau + \epsilon$.

   This implies that every local phase can be associated with a corresponding pair of send ticks, without exceeding the bound of a local phase duration. Therefore, ψ is aligned with the first pair-wise synchronization requirement.

2. For every pair of non-faulty processors (P_i, P_j), if P_i sends a message associated with its local phase ℓ, that is received by P_j in its local phase k. Then, P_i will not accept a send tick from PWCP in local phase $\ell+1$ before P_j accepts a stop receiving tick from PWCP in local phase k.

   If no message of the next local phase is sent prior to accepting the send tick, and no message associated with the current local phase is accepted after accepting the stop receiving tick, the above requirement implies the second pair-wise synchronization requirement.

3. For every pair of non-faulty processors (P_i, P_j), if P_i sent a message associated with its local phase ℓ, that was received by P_j at its local phase k. Then, if P_j sent a message associated with local phase k P_i will accept it in local phase ℓ.

   That means that the duration between the send tick and the stop receiving tick is long enough in order to receive all messages which are related to the current local phase.

We assume every processor can send all of the API elements of a local phase at the beginning of the phase. We also assume that a processor can hold the API elements until it instructed to send them by PWCP.

Protocol's description:
ψ uses PWCP in the following way (as shown in Figure 5.1):

- When a send tick is received from PWCP, send all input elements associated with the current local phase. After sending these phase messages do not send any more messages of the current local phase.

- When a stop receiving tick is received from PWCP, stop receiving messages associated with the current local phase (ignore future output elements with the same local phase).

In the next set of lemmas, we will show that the protocol handles the requirements stated above.

Lemma 57. *For a given $\epsilon > 0$ and for every processor in the system P, the time duration between two contiguous send tick sent by P is less than $\tau + \epsilon$.*

Proof. According to Lemma 56, after the system reaches its steady state the duration between two contiguous send tick is less than τ. □

Lemma 58. *For every pair of non-faulty processors (P_i, P_j), if P_i sends a message associated with its local phase ℓ that was received by P_j in its local phase k. Then, P_i will not accept a send tick from PWCP in local phase $\ell+1$ before P_j accepts a stop receiving tick from PWCP in local phase k.*

Proof. According to Lemma 54, in PWCP steady state, for every pair of non-faulty processors (P_i, P_j), P_i invokes a send tick for the next local phase only after P_j invokes its stop receiving tick of the current local phase. □

Lemma 59. *For every pair of non-faulty processors (P_i, P_j), if P_i sends a message associated with its local phase ℓ that was received by P_j at its local phase k. Then, if P_j sends a message associated with its local phase k, P_i will accept it in its local phase ℓ.*

Proof. According to Lemma 55 in PWCP steady state, for every pair of non-faulty processors (P_i, P_j), P_i invokes a stop receiving tick for the its current local phase ℓ at least D after P_j invokes its send tick of the local phase k. Since P_j sends all messages that are associated with local phase k when it receives the send tick, after another $d < D$ all of these messages have reached to P_i before it generates the stop receiving tick. □

Chapter 6

Pulse Synchronization in Pair-Wise Synchronized Systems

In this section, a protocol to assign synchronization to pair-wise synchronized systems is presented. Given a pair-wise synchronized system S subjected to emulated fail-stop faults, the protocol creates a common event for all non-faulty processors in the system, which we call a common virtual pulse, in a fixed predefined delay from one pulse to another. It will be shown that creating this common virtual pulse deduces the global synchronization property of the system. In addition, a lower bound of reaching this goal is proven, and since the constructed protocol reaches this bound, it is considered optimal.

6.1 Pulse Synchronization Definitions

The pair-wise synchronized property implies a general property on the system, which is captured by the following definition.

Definition 60. In a pair-wise synchronized distributed system, where every processor sends messages to all other processors during every local phase. For a given time t_0, let $k_1, k_2, ..., k_n$ be the indices of the processors' local phases at time t_0. For each $m \in \mathbb{N}$, let P_j be the first processor to send a message of local phase $k_j + m$ at $t_m > t_0$.

We define the system's m^{th} *global phase* (with respect to t_0) to be the period of time that passed from t_m to t_{m+1}.

The following example demonstrates the definition in order to clarify it. Let P_j be the first processor to send a message of its $k_j + 1$ local phase at $t_1 > t_0$, and let P_i be the first processor to send a message of its $k_j + 2$ local phase at $t_2 > t_1$. We define the system's first global phase with respect to t_0 to be the period that passes from t_1 to t_2.

Remark 61. The start and end points of the global phase are not known to the processors in the system. This is due to the fact that the definition of the starting points of the global phases depends on the time that a processor sends a message of its next local phase, and no other processor can know the sending time of another processor.

Definition 62. A *common virtual pulse* is a recurring distributed event for all non-faulty processors in the system, as simultaneous as possible and with a frequency that matches a predetermined regularity.

In our context, we define the predetermined regularity with respect to the number of global phases.

Using these definitions, the formal problem can be presented as follows:

Pulse Synchronization Problem: Given a pair-wise synchronized distributed system S with the predefined length τ for the local phase (Definition 4), and the a drifted transmission bound d (Definition 9), subjected to emulated fail-stop faults, and a constant $T \in \mathbb{N}$. Let t_0 be the point in time where the pair-wise synchronized system stabilizes, meaning every sequence of messages from t_0 is a pair-wise synchronized allowable sequence (Definition 31). One must construct a protocol ψ that from some point in time $\tilde{t} \geq t_0$, all non-faulty processors will generate a common virtual pulse according to ψ in the same global phase such that from this pulse on, the protocol generates a common virtual pulse every T global phases.

6.2 Pulse Synchronization Lower Bound

Next the lower bound on the number of global phases required for solving the pulse synchronization problem is presented. The number of faulty processors is notated as F, where the only assumption regarding F is that $F < \frac{N}{2}$.

Theorem 63. *For a given Constant $T \geq 2$, let $\mathcal{A}$ be any protocol that solves the pulse synchronization problem in a pair-wise synchronized system subjected to Emulated Fail-Stop faults. Let t_0 be the point in time at which the*

pair-wise synchronized system stabilizes (every sequence of messages from t_0 is a pair-wise synchronized allowable sequence (Definition 31)). There exists a run over $\mathcal{A}$ that takes more than F global phases with respect to t_0 until the system generates a common virtual pulse, such that from this pulse and on, the protocol generates a common virtual pulse every T global phases (i.e. solves the pulse synchronization problem).

In order to prove Theorem 63, a reduction from the pulse synchronization problem to the consensus problem [4] will be presented. The lower bound of $F + 1$ phases for the consensus convergence is well known in distributed protocols [25]. A mapping between the consensus problem and the pulse synchronization problem will be presented. Using this mapping, the lower bound on the convergence is deduced.

The Synchronous model referred to is the one presented in [25], where in each phase the following steps are taken in the described order:

1. Each non-faulty processor sends its messages.
2. Each processor reads all messages addressed to it, processes them and decides on its next actions.

Since the consensus problem is defined over a synchronous system and the pulse synchronization problem is defined over a pair-wise synchronized system, we need to justify the connection between the two. The following lemma captures this.

Lemma 64. *Let S be a Synchronous distributed system, which started at t_0. If each of the processors in S defines its Local phases in the following way:*

- *A new local phase starts at t_0.*
- *Whenever a new synchronous phase starts, proceed to the next local phase.*

Then, every sequence of input and output elements $\sigma \in seq(S)$ from t_0 and on is a pair-wise allowable sequence.

Proof. In order to prove the lemma two issues need to be handled:

1. Show that the local phase construction described in the lemma is aligned with the local phase definition (Definition 4).

2. Prove that the sequences produced by the synchronous system are pair-wise synchronized allowable sequences (Definition 31).

According to the synchronous model [25], the duration of each phase is sufficient in order to receive and process all messages sent by any of the processors. In addition, every message that was sent or received by each of the processors is associated with a single synchronous phase, and is hence associated with a particular local phase. Finally, for every pair of processors (P_i, P_j) every message sent from P_i to P_j must be associated with a different synchronous phase, implying that only one message from P_i to P_j can be sent in every local phase. Since a new local phase starts at t_0, the behavior of the Local Phases cannot be affected by transient activities prior to t_0.

After showing that local phases are well defined, the pattern of the constructed system's elements sequences can be examined. In the synchronous model, every input element e_{in} that was sent by processor P_{send} is promised to reach, as an output element e_{out}, the destination processor P_{rec} before the end of the current synchronous phase. In addition, P_{rec} is assured to stop receiving and processing all of its output elements prior to the end of the current synchronous phase, which is also the end of its current Local phase. P_{send} will not send any message until the synchronous phase starts, and since each of the processors defines its Local phase according to the synchronous phases, it is assured that P_{send} will not create any input element of its next Local phase until P_{rec} stops receiving output elements of its current Local phase. To conclude, it was shown that every sequence of input and output elements $\sigma \in seq(S)$ is a pair-wise synchronized allowable sequence. □

Lemma 64 shows that a synchronous system can be referred to as a pair-wise synchronized system since the sequences of the API elements produced by the system are pair-wise allowable sequences. Therefore, it allows us to use a protocol assumed to run on a pair-wise synchronized system in order to solve the consensus problem on a synchronous system.

In order to map the consensus problem to the pulse synchronization problem, the following observation is used. Let $\mathcal{A}$ be a protocol that solves the pulse synchronization problem within $\zeta \geq T$ global phases. Since $\mathcal{A}$ is a self-stabilized protocol, it has a system state for the situation where all processors create the common virtual pulse. We refer to this state as the *pulse system state*. Furthermore, it has an additional system state for the global phase that comes after the common virtual pulse. We refer to this state as the *post-pulse system state*.

The mapping between the consensus problem over a synchronous system and the pulse synchronization problem over a pair-wise synchronized system is captured in the following definition:

Definition 65. Let $\mathcal{A}$ be any self-stabilized protocol that, given $T \in [2, F]$ for the virtual pulse reoccurring rate, solves the pulse synchronization problem over a pair-wise synchronized system, within $\zeta \geq T$ global phases. The *pair-wise consensus mapping (PWCM)* maps $\mathcal{A}$ to a protocol that solves the consensus problem over a synchronous system in the following way:

According to Lemma 64, $\mathcal{A}$ may be executed over a synchronized system. For each of the processors, PWCM sets a counter C for the number of global phases that pass since the last common virtual pulse (increased by 1 for every new global phase). Given an instance of a binary consensus problem over a synchronized fail stop system, every processor that has the initial value of 0 in the consensus problem, starts protocol $\mathcal{A}$ in the pulse system state and is assigned with $C = 0$. Every processor with an initial value of 1 in the consensus problem starts $\mathcal{A}$ in the post-pulse system state and is assigned with $C = 1$.

In order to complete the mapping, we need to map the counter value (C) to either 0 or 1 for the consensus problem. For that, we use the following function:

$$\text{Consensus Return Value} = \begin{cases} 1 & \text{if} \mod (C - \mod (\zeta, T)) = 1 \\ 0 & \text{otherwise.} \end{cases}$$

Remark 66. In the presented mapping function, three assumptions were made regarding the value of T:

1. $T \geq 2$.
2. $\zeta \geq T$
3. $T \leq F$.

Regarding the first assumption, note that if $T = 1$ it means that the pulse is expected to be produced in every global phase. This implies the trivial protocol that instructs the processor to always pulse. Also, note that the PWCM mapping function will be mapped to a trivial solution of the consensus problem of always returning 0.

Regarding the second assumption, since the pulse synchronization problem is defined by the pulse that from it and on, the reoccurring event holds,

it implies that every protocol might need T global phases in order to reach a pulse. Even a stable system that starts from the state when the pulse was just created will need to wait T global phases in order to reach to the stable pulse, which justifies the assumption that $\zeta \geq T$.

Following the previous assumption, considering $T \geq F+1$ will force any protocol to have a run longer than $F+1$ global phases. Hence, the lower bound of $F+1$ holds. Therefore, the reduction focuses on the scenarios where $T \leq F$.

Next, the correctness of the protocol constructed by PWCM function for the consensus problem is proven, using the following set of lemmas:

Lemma 67. *If all processors were in the "post-pulse system state" with counter value 1 at the beginning of $\mathcal{A}'s$ run, all counters values will be* $\mod(\zeta+1, T)$.

Proof. Since $\zeta > (T-1)$, then after $T-1$ global phases all processors will generate the common virtual pulse (according to the definition of this system state in $\mathcal{A}$), and the counter will be set to 0 for all processors. We stop the run after $\zeta-(T-1)$ global phases after the first common virtual pulse. Therefore the value of the counter at the end of the run will be $\mod(\zeta-T+1, T) = \mod(\zeta+1, T)$, as claimed. □

Since $\mod(\mod(\zeta+1,T) - \mod(\zeta,T)) = 1$, it holds that the return value for the consensus problem with the suggested protocol in the scenario that all processors have the initial value of 1 is 1 as expected.

Lemma 68. *If all processors were in the "pulse system state" at the beginning of $\mathcal{A}'s$ run, all counters values will be* $\mod(\zeta, T)$.

Proof. Since $\zeta \geq T$, then after T global phases all processors will generate the common virtual pulse, and the counter will be set to 0 for all processors. We stop the run after $\zeta - T$ global phases after the first common virtual pulse, and therefore the value of the counter at the end of the run will be $\mod(\zeta-T, T) = \mod(\zeta, T)$, as claimed. □

Since $\mod(\mod(\zeta+, T) - \mod(\zeta, T)) = 0$, it holds that the return value for the consensus problem with the suggested protocol in the scenario that all processors have the initial value of 0 is 0 as expected.

Lemma 69. *All processors will have the same counter value at the end of $\mathcal{A}'s$ run.*

Proof. For any system state, we are promised by the properties of $\mathcal{A}$ that the system reaches a common virtual pulse within ζ. After the common virtual pulse is created all processors will set their counters to 0. Therefore we are assured that all processors will have the same counter value from this pulse and on. □

Thus, it holds that the return value for the consensus problem, with the suggested protocol in the scenario where processors have different initial values, will be the same value for all processors as expected.

To conclude, the suggested protocol solves the consensus problem in exactly ζ global phases. If $\zeta \leq F$, we can conclude that such a protocol does not exist. Therefore, every protocol that solves the pulse synchronization problem might need at least $F+1$ global phases in a deterministic model in order to ensure a valid pulse in the system, as claimed in Theorem 63.

6.3 Gain the Globally Synchronized property using the Common Virtual Pulse

In this subsection, we show that solving the pulse synchronization problem deduces the global synchronization property to the system. First, a mapping between each processor's local phase and the system's global phase is presented. Afterward, the usage of the common virtual pulse in order to label each processor's local phase is shown. Finally, we show that if every processor uses this labelling function for its local phase label, the result is an agreed label for all processors, and by that achieves the globally synchronized property.

First, the mapping between each processor's local phase and the system's global phase is presented using the following lemmas.

Lemma 70. *For each of the processors in the system P_i, which in t_0 was in its kth local phase, P_i sends all messages associated with its $k+m$ local phase during the mth global phase.*

Proof. We mark the starting point of the mth global phase as t_m (see Definition 60). Let ${t_i}^m$ be the time when P_i sent its first message of its $k+m$ local phase. It must hold that ${t_i}^m \geq t_m$, otherwise the mth global phase would have started at ${t_i}^m$. Thus, the first message that P_i sent on its $k+m$ Local Phase was during the mth global phase. If the system moved to global phase

(m+1), it means that some processor P_j that was in its ℓth local phase at t_0 sent a message of its $\ell + m + 1$ local phase. According to the pair-wise synchronized property, when P_j sent the first message of phase $\ell + m$, P_i had finished its $k + m$ local phase, including stopping sending messages of that phase. Therefore, all messages that P_i sent in its $k + m$ local phase were during the mth global phase. □

Lemma 71. *For each of the processors in the system P_i, which in t_0 was in its kth local phase, all messages received by P_i associated with its $k + m$ local phase were sent during the mth global phase.*

Proof. Let $k_1, k_2, ..., k_n$ be the processors' local phase indices at t_0. Let P_j be the first processor to send a message at its $k_j + m$ local phase, and by that starts the mth global phase. Let P_ℓ be the first processor to send a message at its $k_\ell + m + 1$ local phase, and by that starts the $(\mathrm{m+1})^{th}$ global phase. Since every processor sends messages to all other processors on every local phase (see Definition 60), it means that P_j sends a message to all other processors in its $k_j + m - 1$ Local Phase. According to the pair-wise synchronized property, when P_j sent the first message of phase $k_j + m$, all processors had stopped receiving messages, including P_i, which stopped receiving messages of the $(k_i + m - 1)$ local phase.

Using the same argument for P_ℓ shows that P_i had stopped receiving messages of phase $k_i + m$ before P_ℓ sent its first message of local phase $k_\ell + m + 1$. Therefore, all messages accepted by P_i in its local phase $k + m$ were sent during the mth global phase. □

Next, the *Count From Pulse (CFP)* labelling function for the processors' local phase is presented. CFP assigns the number of local phases passed since the last common virtual pulse as the current local phase label. Where the label of the Local phase that follows the common virtual pulse is 0. If CFP is used by all processors then the system is globally synchronized (Lemma 73).

Definition 72. Each processor is assigned with a counter $C \in [0, T]$ that counts the number of Local phases that passed since the last common virtual pulse. The *Count From Pulse (CFP)* labelling function $CFP : \mathbb{N}_0^+ \to [0, T - 1]$ labels for each processor P_j, its local phase label $\theta_j[\ell] \in \mathbb{N}_0^+$ (Definition 5) in the following way:

$$\theta_j[\ell] = \mod (C, T) .$$

Lemma 73. *If all processors in the system are using the CFP labelling function, the globally synchronized property holds in the system.*

Proof. Since the pulse synchronization problem is assumed to be solved, from some point in time $\tilde{t}$, for every $k \in \mathbb{N}$, all non-faulty processors generate a common virtual pulse every kT global phases. Therefore, every processor labels the current local phase as the number of local phases that have passed since the last common virtual pulse according to the CFP labelling function.

The global phases in this proof are considered with respect to $\tilde{t}$, and the local phase index of every processor in $\tilde{t}$ is marked as $k_1, k_2, ..., k_n$. For every pair of processors (P_i, P_j) in the system, when P_i sends a message to P_j in Local phase $k_i + m$, according to Lemma 70, this message was sent during the mth global phase. According to Lemma 71, P_j must receive this message during the mth global phase, in $P_j^{\prime}s$ $k_j{+}m$ Local phase. Let $k_i{+}\ell$, $k_j{+}\ell$ be the indices of P_i and P_j respectively, when the last virtual pulse was generated. Hence, the counter value is $C = m - \ell$.

If $m{-}\ell < T$ the label CFP assigned to $P_i^{\prime}s$ $k_i{+}m$ local phase is $m{-}\ell$, and the same holds for P_j. If $m - \ell = T$ it is possible that one of the processors generates the common virtual pulse before the other and changes its counter value to $C = 0$. But, since CFP assigns the same value when $C = 0$ and when $C = T$, both processors will have label 0 as the local phase label. Since both share the same local phase label it means that the globally synchronized property holds in the system. □

Note that it is possible that during the run, processors will have different values for their current local phase. However, we are promised according to Lemma 73 that when a certain processor P_i receives a message from another processor P_j, the Local phase Label that P_i and P_j assign to this message is identical.

6.4 The Global Sync Protocol

In order to accomplish the common virtual pulse, we make use of another clocking protocol GSP (Global Sync Protocol). GSP is assumed to run over a Fail-Stop system, and runs for $F + 1$ local phases. We instruct a protocol ψ that wants to use GSP in order to achieve the common virtual pulse, to run a sequence of instances of GSP such that the starting system state of the next GSP is the same as the ending system state of the last GSP.

ψ's requirements are:

- Self-Stabilization, as expressed by the following properties:
  GSP has an internal set of safe system states C such that following properties hold:

  1. Steadiness: If the initial system state of GSP is $c \in C$, then the system state when PWCP finishes its run is $\bar{c} \in C$.
  2. Convergence: If the initial system state of GSP is $c \notin C$, then after GSP completes its run, the system state of GSP is $\bar{c} \in C$.

- Common virtual pulse, the protocol needs to provide to all of the processors a common virtual pulse once every T global phases.

The motivation for these properties is that after the fail-stop emulation layer stabilizes, by the convergence property of GSP, its instance will end its run in a safe system state. Since each instance has a runtime bounded by $F + 1$, a run of GSP over the stabilized fail-stop emulation layer simulates a fail-stop run. Once the ending point of the former instance is in C, the starting system state of the next GSP will also be in C. According to the Steadiness property of GSP, each of its following instances will start and finish in system states that belong to C. Thus, running sequences of GSP will result in a traditional self-stabilized protocol.

After the system stabilizes, it will produce a common virtual pulse every T local phases, as needed.

Protocol's Variables:
Each processor in the protocol has an instance of the abstract state machine appearing in Figure 6.1

6.5 The Protocol

For every processor in each local phase:

- Process all the messages received during the previous local phase in the following way:
  - If $N - F$ messages are from the same state X move to state $X + 1$ (according to the state machine).

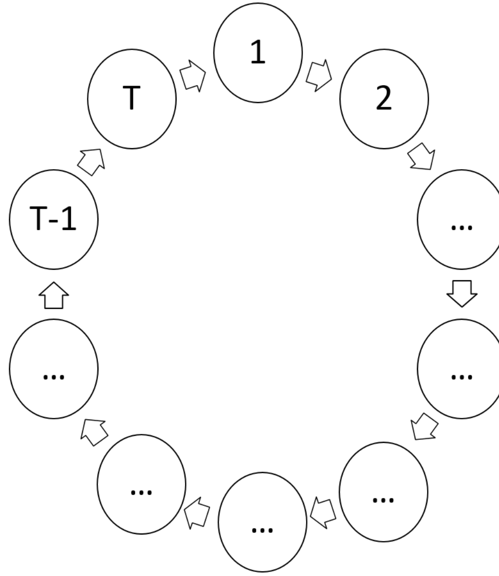

Figure 6.1: The GSP Abstract State Machine

  – Otherwise, move to state 1 (including if you did not received any message).

- Send the new state to all other processors.

- When state T is reached, create the common virtual pulse.

Definition 74. We define the *safe system states set* C as all the system states where the following hold:

- The abstract state associated with the local phase is common for all non-faulty processors.

- All the messages in transit are from the same state.

Claim 75. GSP is steady.

Proof. If all non-faulty processors are in the same state X and every one of them has sent this state to all others, all processors will accept at least $N-F$ messages of that state. Thus, all processors will move to state $X+1$ in the next global phase and will send it as their next message. □

Definition 76. A *clean phase* is a phase where no processor failed (acted against the protocol's rules and became inactive).

Lemma 77. *After a clean phase the system converges.*

Proof. If all non-faulty processors are in the same state according to Lemma 75 we are finished. Otherwise, given a clean phase, we are promised that all non-faulty processors will receive the same set of messages. Thus, if $N - F$ processors are in the same state X, then all processors will move to state $X + 1$. If there is no state with more than $N - F - 1$ processors, all processors will move to state 1. Either way, we reach a point where all non-faulty processors are in the same state. In the next global phase, every processor will sent its current state, which brings us to a safe state. □

Lemma 78. *Let t_0 be the time where the pair-wise synchronized layer had stabilized. The GS Protocol converges within $F + 1$ global phases past t_0.*

Proof. During a full run of the protocol there must be at least one steady state, because we run it for $F + 1$ local phases which means that even if we have one failing processor for each phase we still have a clean phase during this instance of the protocol. According to Lemma 77 after a clean phase the system reaches one of its safe system states.

After t_0, the pair-wise synchronized layer is stable and every local phase can be mapped to the corresponding global phase (Lemma 70 and Lemma 71). Hence, $F + 1$ local phases that started after t_0 represent $F + 1$ global phases in the system. □

Lemma 79. *The GS Protocol produces a common virtual pulsee every T global phases.*

Proof. Once the system is in one of the safe system states all processors will move one state ahead in every global phase and therefore will reach state T every T global phases and will produce the common virtual pulse. □

Therefore, ψ solves the pulse synchronization problem using GSP. According to Lemma 73, by using the CFP labelling function (Definition 72) the system can gain the globally synchronized property using the common virtual pulse.

Chapter 7

Conclusions and Future Work

In this thesis, a modular solution for a self-stabilized protocol that solves the clock synchronization problem was presented. One important conclusion from this research is the strength and advantages of dividing a complicated task into sub-tasks. This method simplifies the task, allowing one to reach lower bounds with a set of rather simple algorithms. In addition, this approach allows one to switch between different variants of the same concept without the need to construct the whole algorithm from scratch. Therefore, there are several natural directions that can evolve from this research by expanding the basic concept of the protocol to other system models.

The main innovation in the research is the sub-task of creating the pairwise layer. Informally, this layer changes the model from a bounded-delay model to a synchronous model with a constant (rather low) convergence time. The idea of being only partly synchronized significantly simplifies the task of reaching the full clocks' synchronization. In addition, this property can be good enough for other protocols that are assumed to have an external pulse to start the next phase in the system without any further addition (such as [22]).

One natural way is to expand the protocol to deal with Byzantine faults. In order to do that, there is a need to replace the bottom layer that simulates the emulated fail-stop model on top of omission faults, in a layer that simulates emulated fail-stop model on top of Byzantine faults. Such a simulation already exists in synchronous systems and is known as *grade-cast* [37]. The adjustments to the semi-synchronous are not trivial but seem possible.

Another possible direction is to switch to a probabilistic model instead of the deterministic model. The main gain such a change can contribute is

to move from linear-bounded convergence time to expected constant time. There exists a probabilistic protocol [22] that assumes the system module is *Coherent*, which is very a similar module to the pair-wise synchronized. This protocol can achieve an expected constant convergence time using *self-stabilized coin flipping algorithm* (see more details in [22]).

There are other assumptions made in this protocol in order to simplify it that may be dropped in future research, such as the full connectivity assumption, which raises the question of what the bounds are on a network with a limited connectivity.

Another simplifying assumption was made in the bottom layer, where a kind of broadcast is being used in order to make sure that every message can be covered by $N - F$ of the processors. What should it take in order to create private channels in the system? One possible solution is to sign every element with the recipient public key so that only the recipient can open them, yet they can still be propagated through the other processors. Other solutions can be developed in future research.

A lower bound was presented for the deterministic model on the synchronization problem that was raised. It is possible to come up with another definition of synchronization simulations and try to understand the upper and lower bounds with the other models.

Yet, getting to an optimal convergence time $O(N)$ with three protocols that are quite intuitive, and can be implemented in existing systems, is an impressive achievement that shows the strength of the modular system.

Bibliography

[1] Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, October 1985.

[2] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[3] Shlomi Dolev. *Self-stabilization*. MIT press, 2000.

[4] Nancy A. Lynch. *Distributed Algorithms (The Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, 1st edition, March 1997.

[5] Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

[6] Leslie Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *J. ACM*, 32(1):52–78, January 1985.

[7] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *J. ACM*, 33(3):499–516, May 1986.

[8] Danny Dolev, Joe Halpern, and H. Raymond Strong. On the possibility and impossibility of achieving clock synchronization. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, pages 504–511, New York, NY, USA, 1984. ACM.

[9] Joseph Y. Halpern, Barbara Simons, Ray Strong, and Danny Dolev. Fault-tolerant clock synchronization. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, pages 89–102, New York, NY, USA, 1984. ACM.

[10] P. Ramanathan, K.G. Shin, and R.W. Butler. Fault-tolerant clock synchronization in distributed systems. *Computer*, 23(10):33–42, Oct 1990.

[11] Fred B. Schneider. A paradigm for reliable clock synchronization. Technical report, In Proceedings Advanced Seminar of Local Area Networks, 1986.

[12] Barbara Simons. An overview of clock synchronization. In Barbara Simons and Alfred Spector, editors, *Fault-Tolerant Distributed Computing*, volume 448 of *Lecture Notes in Computer Science*, pages 84–96. Springer New York, 1990.

[13] Emmanuelle Anceaume and Isabelle Puaut. A taxonomy of clock synchronization algorithms, 1997.

[14] Jennifer Lundelius Welch and Nancy Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1 – 36, 1988.

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

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

[17] MahyarR. Malekpour. A byzantine-fault tolerant self-stabilizing protocol for distributed clock synchronization systems. In AjoyK. Datta and Maria Gradinariu, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 4280 of *Lecture Notes in Computer Science*, pages 411–427. Springer Berlin Heidelberg, 2006.

[18] Ariel Daliot, Danny Dolev, and Hanna Parnas. Self-stabilizing pulse synchronization inspired by biological pacemaker networks. In Shing-Tsaan Huang and Ted Herman, editors, *Self-Stabilizing Systems*, volume 2704 of *Lecture Notes in Computer Science*, pages 32–48. Springer Berlin Heidelberg, 2003.

[19] Ariel Daliot, Danny Dolev, and Hanna Parnas. Linear time byzantine self-stabilizing clock synchronization. In Marina Papatriantafilou and

Philippe Hunel, editors, *Principles of Distributed Systems*, volume 3144 of *Lecture Notes in Computer Science*, pages 7–19. Springer Berlin Heidelberg, 2004.

[20] Danny Dolev and EzraN. Hoch. Byzantine self-stabilizing pulse in a bounded-delay model. In Toshimitsu Masuzawa and Sébastien Tixeuil, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 4838 of *Lecture Notes in Computer Science*, pages 234–252. Springer Berlin Heidelberg, 2007.

[21] Danny Dolev, Matthias Függer, Christoph Lenzen, and Ulrich Schmid. Fault-tolerant algorithms for tick-generation in asynchronous logic: Robust pulse generation. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 163–177. Springer Berlin Heidelberg, 2011.

[22] Michael Ben-Or, Danny Dolev, and Ezra N. Hoch. Fast self-stabilizing byzantine tolerant digital clock synchronization. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 385–394, New York, NY, USA, 2008. ACM.

[23] MichaelJ. Fischer, NancyA. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.

[24] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3):549–587, July 1990.

[25] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183 – 186, 1982.

[26] Danny Dolev, Janne H. Korhonen, Christoph Lenzen, Joel Rybicki, and Jukka Suomela. Synchronous counting and computational algorithm design. *CoRR*, abs/1304.5719, 2013.

[27] Yehuda Afek, Hagit Attiya, Alan Fekete, Michael Fischer, Nancy Lynch, Yishay Mansour, Dai-Wei Wang, and Lenore Zuck. Reliable communication over unreliable channels. *J. ACM*, 41(6):1267–1297, November 1994.

[28] Brian A. Coan and Jennifer L. Welch. Modular construction of a byzantine agreement protocol with optimal message bit complexity. *Information and Computation*, 97(1):61 – 85, 1992.

[29] B. A. Coan and J. L. Welch. Modular construction of nearly optimal byzantine agreement protocols. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, PODC '89, pages 295–305, New York, NY, USA, 1989. ACM.

[30] Gil Neiger and Sam Toueg. Simulating synchronized clocks and common knowledge in distributed systems. *J. ACM*, 40(2):334–367, April 1993.

[31] T.K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

[32] Philip M Merlin, Robert G Gallager, and Adrian Segall. A recoverable protocol for loop-free distributed routing. *LIDS-P*, September 1978.

[33] Adrian Segall. Distributed network protocols. Technical report, DTIC Document, 1980.

[34] E.M. Gafni and D.P. Bertsekas. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *Communications, IEEE Transactions on*, 29(1):11–18, Jan 1981.

[35] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, January 1987.

[36] P.M. Melliar-Smith, L.E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 1(1):17–25, Jan 1990.

[37] Paul Feldman and Silvio Micali. Optimal algorithms for byzantine agreement. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 148–161, New York, NY, USA, 1988. ACM.

[38] Michael Ben-Or, Danny Dolev, and Ezra N. Hoch. Simple gradecast based algorithms. *CoRR*, abs/1007.1049, 2010.