A PRESCRIPTION FOR PARTIAL SYNCHRONY

A Dissertation

by

SRIKANTH SASTRY

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2011

Major Subject: Computer Engineering

A PRESCRIPTION FOR PARTIAL SYNCHRONY

A Dissertation

by

SRIKANTH SASTRY

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Co-Chairs of Committee,	Jennifer L. Welch
	Scott M. Pike
Committee Members,	Dmitri Loguinov
	Paolo Gardoni
Head of Department,	Valerie E. Taylor

May 2011

Major Subject: Computer Engineering

ABSTRACT

A Prescription for Partial Synchrony. (May 2011)

Srikanth Sastry, B. Tech., Calicut University, India

Co–Chairs of Advisory Committee: Dr. Jennifer L. Welch
Dr. Scott M. Pike

Algorithms in message-passing distributed systems often require *partial synchrony* to tolerate crash failures. Informally, partial synchrony refers to systems where timing bounds on communication and computation may exist, but the knowledge of such bounds is limited. Traditionally, the foundation for the theory of partial synchrony has been *real time*: a time base measured by counting events external to the system, like the vibrations of Cesium atoms or piezoelectric crystals.

Unfortunately, algorithms that are correct relative to many real-time based models of partial synchrony may not behave correctly in empirical distributed systems. For example, a set of popular theoretical models, which we call $\mathcal{M}_*$, assume (eventual) upper bounds on message delay and relative process speeds, regardless of message size and absolute process speeds. Empirical systems with bounded channel capacity and bandwidth cannot realize such assumptions either natively, or through algorithmic constructions. Consequently, empirical deployment of the many $\mathcal{M}_*$-based algorithms risks anomalous behavior.

As a result, we argue that real time is the wrong basis for such a theory. Instead, the appropriate foundation for partial synchrony is *fairness*: a time base measured by counting events internal to the system, like the steps executed by the processes. By way of example, we redefine $\mathcal{M}_*$ models with fairness-based bounds and provide algorithmic techniques to implement fairness-based $\mathcal{M}_*$ models on a significant subset

of the empirical systems. The proposed techniques use *failure detectors* — system services that provide hints about process crashes — as intermediaries that preserve the fairness constraints native to empirical systems. In effect, algorithms that are correct in $\mathcal{M}_*$ models are now proved correct in such empirical systems as well.

Demonstrating our results requires solving three open problems. (1) We propose the first unified mathematical framework based on Timed I/O Automata to specify empirical systems, partially synchronous systems, and algorithms that execute within the aforementioned systems. (2) We show that crash tolerance capabilities of popular distributed systems can be denominated exclusively through fairness constraints. (3) We specify exemplar system models that identify the set of *weakest system models* to implement popular failure detectors.

To my family, for the life that made this happen.

ACKNOWLEDGMENTS

This dissertation edifies an unremitting debt I owe my advisers Scott Pike and Jennifer Welch. I am incredibly grateful for their mentorship, training, support, and (most importantly) friendship. Professionally, their influences have been serendipitously complementary. From Scott, I learned that the hallmark of a good researcher is their ability to identify and favor the *interesting* problems over others. Scott's pursuit of excellence has had a lasting impact on my writing, both as a product and as a process. From Jennifer, I learned the value and impact of being precise while remaining informal in my research. She has been instrumental in my understanding of *excellence* versus *perfection*; the former signals sufficient maturity in the work for its publication, whereas the latter is an ideal that, if relentlessly pursued, risks becoming a mirage. My thanks to Scott and Jennifer, however, go beyond their role as my academic advisers. Their friendship, especially during my trying times, has been a valuable source of resilience and inspiration.

I also thank Nancy Lynch for her suggestions that added significant value to the content and presentation of this dissertation. I remain amazed by the impact she has had on my research given the relative sparsity in our interactions. I look forward to our future collaborations.

My thanks to Paolo Gardoni and Dmitri Loguinov for their time, effort, and commitment as members of my doctoral committee. I cannot forget the members of Scott's research group: Kaustav Ghosal, Saurin Shah, and Yantao Song; the members of Jennifer's research group: Hyun-Chul Chung, Gautam Roy, Erica Wang, Josef Widder, and Saira Viqar; and summer visitors to Jennifer's research group: Whitney Maguffee and Tsvetomira Radeva for their constant feedback, lively conversations, engaging debates, and good times in both work and play.

Hearkening back to my undergraduate years, I am especially indebted to K. Muralikrishnan for his inspiring lectures and conversations, and for motivating me to pursue research in theoretical computer science.

Finally, no amount of words can even begin to describe my gratitude for the unending love and support that I have received from my family: my brother, Shankar, for being my close friend and confidant, and my parents for supporting me in all my enterprises including the one culminating in this dissertation.

TABLE OF CONTENTS

CHAPTER Page

I INTRODUCTION 1

I.1. Motivation 3
I.2. Scope and research goal 5
I.3. Methodology 8
I.4. Organization 9

II BACKGROUND AND RELATED WORK 10

II.1. Tolerating process crashes 10
II.2. Comparing failure detectors 20
II.3. Comparing failure detectors with problems 21
II.4. Failure detector implementations 23
II.5. Open questions 28

III DISTRIBUTED SYSTEM MODELS 30

III.1. Distributed system model classes 30
III.2. The $\mathcal{M}_*$ models: a critique 44

IV METHODOLOGY 56

IV.1. Correct-reliable communication channels. 57
IV.2. Scheduler 57
IV.3. Failure detectors 59

V FORMAL FRAMEWORK AND MODELING 60

V.1. Motivation 60
V.2. Timed I/O Automata 66
V.3. Constituents of a distributed system 72
V.4. Executions of a distributed system 89
V.5. Distributed system models 91
V.6. Application protocols as TIOA 102
V.7. Algorithms and solvability 104

VI FAILURE DETECTORS 106

VI.1. Formal specification 107

CHAPTER Page

VI.2. Failure detectors within the formal TIOA framework . . 112
VI.3. Solvability 115
VI.4. Reducibility 117

VII CONSTRUCTING $\mathcal{M}_*$ 120

VII.1. Methodology 121
VII.2. $\Diamond\mathcal{P}$: a candidate failure detector 123
VII.3. Implementing Chandra-Toueg failure detectors 123
VII.4. Extracting fairness from Chandra-Toueg failure detectors 128

VIII IMPLEMENTING $\Diamond\mathcal{P}$ ON EMPIRICAL SYSTEMS 152

VIII.1. Tolerating infinite message loss 152
VIII.2. Process celeration 154
VIII.3. Bichronal clocks 160
VIII.4. Bichronal clocks as a part of the control protocol 160
VIII.5. Implementing $\Diamond\mathcal{P}$ 164

IX CONCLUSION 180

IX.1. Summary of the results 180
IX.2. Significance 184
IX.3. Open problems 186

REFERENCES 190

APPENDIX A 200

VITA 204

LIST OF FIGURES

FIGURE Page

1 The combination of constructions to implement $\mathcal{M}_*$ on top of the empirical system model. 58

2 Schematic of the Timed I/O Automata framework for a message-passing distributed system. 74

3 Schematic of the TIOA for a process consisting of multiple protocols (ignoring process crashes). 78

4 Interaction between the control protocol automaton and an application protocol automaton at a process i. See Sections V.3.2.1, V.3.2.2, V.3.2.3,and V.3.3 for detailed descriptions. 87

5 TIOA schematic for a message-passing distributed system subject to process crashes. 88

6 TIOA construction to show equivalence between failure detector $\mathcal{D}$ and system model $\mathcal{M}$. If the failure detector implemented within the control protocol of system model $\mathcal{M}$ satisfies properties of $\mathcal{D}$, then we say that $\mathcal{D}$ is solvable in $\mathcal{M}$. If the application protocol in the figure implements a scheduler using the failure detector $\mathcal{D}$ (from the control protocol), and the scheduler satisfies the properties of $\mathcal{M}$, then we say that $\mathcal{D}$ is reducible to $\mathcal{M}$. 119

7 TIOA schematic for building fairness-based partially-synchronous system on top of the empirical system using a failure detector. 122

LIST OF ALGORITHMS

ALGORITHM		Page
1	Signature for minimal states and actions of a control protocol automaton. For descriptions of the automaton, see Sections V.3.2.1 and V.3.2.3.	80
2	Minimal state transitions for the control protocol automaton. For descriptions of the automaton, see Sections V.3.2.1 and V.3.2.3.	81
3	Application protocol automaton framework. For description of the automaton, see Sections V.3.2.2 and V.3.2.3.	83
4	Fault pattern automaton.	86
5	TIOA for the asynchronous system model (signature and states).	95
6	TIOA for the asynchronous system model (state transitions).	96
7	TIOA for the fairness-based $\mathcal{M}_*$ system models (signature and states).	99
8	TIOA for the fairness-based $\mathcal{M}_*$ system models (state transitions).	100
9	TIOA for the empirical system TIOA (signature and states).	101
10	TIOA for the empirical system TIOA (state transitions).	103
11	Failure Detector TIOA.	114
12	TIOA for a Failure-Detector-Querying Application Protocol.	116
13	Implementing Chandra-Toueg failure detectors in system models where (some) processes are k-proc-fair and d-com-fair.	124
14	Signature and states of the TIOA for the partially-synchronous system model using failure detector $\mathcal{D}$. The state transitions are in Algorithm 15.	131

ALGORITHM Page

15 State transitions for the TIOA for the partially-synchronous system model using failure detector $\mathcal{D}$ shown in Algorithm 14. 132

16 Program action for the scheduler in the partially-synchronous system model using failure detector at process i. 133

17 Signature and states for the TIOA specification for the control protocol in empirical systems with a bichronal timer. The state transitions for the TIOA are specified in Algorithm 18. 162

18 Transitions for the TIOA Specification of the control protocol in empirical systems with a bichronal timer. The signature and states of the TIOA is specified in Algorithm 17. 163

19 Program action of $\Diamond\mathcal{P}$ implementation at process i. 164

CHAPTER I

INTRODUCTION

They didn't care if it worked in fact, because they were already sure it wouldn't work in theory.

Here Comes Everybody, 2008
–Clay Shirkey

Tolerating process crashes in distributed systems often requires some temporal constraints on computation and communication [41]. Consequently, efforts to design crash-fault tolerant algorithms entails assuming certain bounds on process speeds and message delays in distributed systems. These system-model assumptions may be simplifying to aid algorithm design, or they may be empirically verifiable to aid algorithm deployment (in empirical systems). Ideally, we would like these assumptions to be both, so that the algorithms may be easily designed and deployed. Unfortunately, in most of the research in distributed computing to date, the system-model assumptions either favor simplicity or verifiability, but not both. Often, simplistic idealized models are favored by theoreticians whose primary goal is to determine the relative solvability of problems and complexity analysis of solutions to problems, whereas verifiable models are favored by practitioners whose primary goal is to build and deploy distributed systems that solve the problems at hand. Not surprisingly, this phenomenon has cleaved the results in the theory of distributed computing from the results in practical distributed systems. This dissertation addresses the aforementioned apparent disconnect between theory and practice in distributed computing.

Historically, such disconnects have been observed, documented, and opined upon

---

The journal model is Distributed Computing.

for over 25 years. In the book *Distributed Systems* [64], first published in 1993, Schenider noted an apparent tension within the distributed systems community between the advocates for 'modeling and analysis' and the advocates for 'experimental observations'. He attributed this tension to the nascency of distributed systems as a discipline. He argued that this tension is inevitable in young disciplines and often "masquerades as a dichotomy between 'theory' and 'practice'." Schenider stated [64, p. 18]: "Practitioners complain that they learn little from theory. Theoreticians complain that practitioners are not addressing the right problem."

A decade later, in 2003, Fischer and Merritt, in their survey paper "Appraising Two Decades of Distributed Computing Theory Research" [44] confirmed the persistence of this disconnect between theory and practice. They contended that it was primarily due to "a lack of sufficient generality and realism in models, methods, and results" [44]. They explained that the general (and often idealized) models fail to account for significant aspects of empirical systems, whereas more specific models are difficult to apply (with respect to developing algorithms and protocols).

It has been close to two decades since Schenider's observations in [64] and over eight years since Fischer and Merritt's exposition in [44] and (except for a few isolated cases like Paxos algorithms [56] and their implementations) the disconnect, or dichotomy, between theory and practice continues to persist to this day. So the natural follow-up questions are: "Is this disconnect inherent to the discipline of distributed computing? If not, then how can the theory and practice of distributed systems be reconciled?"

This dissertation reviews this disconnect within the purview of crash faults in message-passing systems and, within the same purview, bridges a gap between the theoretical results and the practical challenges of fault-tolerant distributed computing. Specifically, we develop methodological and algorithmic techniques that demonstrate

the applicability of classic (theoretically significant) temporally-constrained models of distributed computing, and the algorithms designed for these models, on empirical distributed systems.

The remainder of this chapter provides an overview of the motivation, goals, scope, and methodology of this dissertation.

I.1. Motivation

Classic temporally-constrained distributed system models, also called *partially synchronous* models [35], make certain simplifying assumptions about computation and communication that are, at first glance, antithetical to the observed behavior of empirical systems. For instance, the *partially synchronous* models described in [20,33,35] assume that communication is (eventually) reliable. It is a remarkably simplifying assumption that provides theoreticians with a significant advantage. It allows them to focus on the intellectual challenges inherent to the problem being solved and develop algorithmic techniques specific to the problem itself, rather than conflate issues of messages loss with the challenges of specific problems. In fact, such simplifying assumptions, which have become a commonplace in the theory of distributed computing, are motivated by the advantages they confer through separations of concerns similar to the one described above.

This approach has helped theoreticians identify the boundaries of solvability of problems such as consensus [65], atomic commit [57], mutual exclusion [30], and clock synchronization [55][1] in the presence of uncertainty precipitated by faults such as process crashes [77], arbitrary (Byzantine) process behavior [65], and recoverable state corruption [34]. Despite the apparent infidelity of such system models to em-

[1] All of problems referenced in this dissertation are described in Appendix A.

pirical systems, these results have had a lasting impact on how distributed systems are designed and implemented. For instance, triple modular redundancy is no longer assumed to be sufficient in distributed systems for single-fault masking thanks to the result in [65], and tolerance to node crashes in empirical distributed systems necessarily entails guarantees on communication delay and processor speeds as established by [42]. The Paxos algorithm [56] to solve the consensus problem deserves a special mention here. Paxos has emerged as a popular choice for solving consensus and other related problems in massively distributed server farms [18, 52, 80, 81].

Despite the advantages of making such simplifying assumptions and the significant impact of the lower bounds and impossibility results from this approach, the applicability of the algorithms thus developed remains limited in empirical systems (except for the Paxos algorithm that was mentioned earlier). There are several reasons for such limited applicability. Next, we discuss three important reasons that this dissertation addresses.

The first reason, noted in [44], is insufficient realism in system models adopted in developing the theory of distributed systems. On the one hand, the idealized system models used in theory often establish overly pessimistic bounds on solvability of problems and do not reflect the capabilities of empirical systems; consequently, the impossibility results from theory do not provide the necessary insight for solving these problems in empirical systems. On the other hand, these idealized assumptions on system models (for instance, the assumption about eventually reliable channels discussed earlier) are invalidated in empirical systems thereby making the theoretical results that depend on these assumptions irrelevant.

The second reason, explained in [18], is the non-static nature of systems. Often, theoretical system models assume a static systems that are 'well behaved' insofar as there exist hard real-time bounds on message delays and process speeds that hold

eternally. In reality, empirical systems do not satisfy such static properties. Even 'relatively well-behaved' empirical systems experience periods of unbounded communication delays and low throughout especially during during periods of unexpectedly high workloads or external denial-of-service attacks.

The third reason, discussed in [18, 44], is that of fault ranges. The algorithms proposed and discussed in the theory of distributed computing tolerate different kinds of faults. But empirical systems are required to tolerate a wider range of faults. Algorithms that can tolerate a wide variety of faults often provide overly pessimistic bounds, as discussed earlier, and algorithms with more realistic bounds simply lack the fault tolerance capability over the range of faults experienced by empirical systems.

I.2. Scope and research goal

The task of developing methodologies and techniques for designing algorithms that address the issue of theoretical correctness and practical deployment for *all* families of problems subject to *all* varieties of common fault types in *all* empirical systems is well beyond the scope of a doctoral dissertation; therefore, we limit the scope of our investigation to the following.

I.2.1. Scope

This dissertation focuses on one computation-fault type, one communication-fault type, and a specific set of empirical systems. The computation-fault type is *crash faults*, and the communication-fault type is *arbitrary message delay and loss*. A process in a distributed system is said to have crashed if it ceases execution without warning and never recovers. A message said to be late if it is delivered after the expected time of arrival; a message (after being sent) is lost if it is never delivered.

Empirical and anecdotal evidence suggests that in 'real-world' distributed systems, a significant subset of messages may be arbitrarily late (or lost).

Crash faults and message delay/loss are natural fault classes. They are realistic and benign. Admittedly, empirical systems are subject to many more common fault types; however, to tolerate all the common fault classes in empirical systems, it is necessary to tolerate at least crashes, message delay and loss. Additionally, there are existing techniques that can translate algorithms tolerant of simple crash failures into ones tolerant of more severe failures (e.g. [13]). Therefore, tolerating these faults provides a reasonable starting point.

The list of empirical distributed systems is too diverse for any single family of system models to describe all of them adequately. We restrict our investigation to a set of distributed system models that share sufficiently common properties and can be abstracted by a single family of system models. Briefly, we consider systems that satisfy the following properties:

1. The absolute process speeds may be unbounded.
2. The exists an upper bound on relative process speeds that is potentially unknown.
3. Processes may crash at any time.
4. The communication links may lose an infinite subset of messages sent.
5. An infinite subset of messages sent on the communication links may be delayed for arbitrary durations
6. An infinite number of messages sent on the communication links are delivered within some (potentially unknown) bound on message delay, and such messages are not too sparsely distributed over time,

I.2.2. Research goal

In the context of crash faults and message delay and loss, the goal of the dissertation is to develop methodologies and techniques for designing crash-tolerant algorithms that can be shown to be correct in theory and deployable in the empirical systems described previously. The methodology adopted in this dissertation is motivated by two highly desirable properties for the results in this dissertation.

The first property is backward compatibility. In additional to being a realistic fault type, crash faults are also one of the most popular fault types in distributed computing literature. There is a rich body of results spanning over 25 years that focuses on tolerating crash faults in distributed system models. There exist several solutions to problems like consensus, atomic commit, mutual exclusion,[2] and such that tolerate crash faults in idealized system models. The methodology proposed in this dissertation ensures that many of these existing solutions which have been proven correct in their respective (idealized) system models continue to behave correctly in the empirical systems as well.

The second property is preserving the separation of concerns discussed in Section I.1. If the results in this dissertation were to sacrifice the simplifying assumptions in the system model specifications, then these results would not gather adopters in the theory of distributed computing. Hence, we would like to preserve the advantages offered by simplified assumptions in idealized system models while dispensing with the disadvantages associated with practical deployment of the algorithms developed for these system models.

The aforedescribed properties inform and motivate the methodology adopted by this dissertation. We discuss this methodology next.

---

[2]Each of these problems is described in Appendix A.

I.3. Methodology

The methodology followed in this research is to implement the idealized system models favored by theoreticians as an overlay system on top of the empirical systems. The primary tasks in this methodology is to eliminate the computation and communication uncertainty present in empirical systems but absent in the idealized system models while preserving the temporal bounds on computation and communication that are arguably present in the empirical systems and guaranteed by the idealized system model. By implementing these idealized system models in existing empirical systems, we address the issues of both backward compatibility and separation of concerns described in Section I.2.2.

None of the known techniques to extract idealized communication properties such as eventual reliable communication focus on preserving the temporal guarantees in communication and communication. Therefore, in order to implement such idealized system models, we need a mechanism to encapsulate the temporal guarantees satisfied by empirical systems so that they may be reintroduced into the idealized system model that is implemented. One such mechanism is provided by the *unreliable failure detector* oracles [20]. These failure detector oracles are believed to encapsulate the 'timeliness' properties of the underlying (empirical) systems.

In summary, we start with real-time based system models that (arguably) describe empirical systems adequately enough to construct appropriate *failure detector* oracles. Then, all the necessary idealized properties of the overlay system, except for the temporal guarantees, are implemented on top of the empirical system. Subsequently, the failure detector oracles are used to 'inject' the temporal guarantees that the idealized system models are required to satisfy.

I.4. Organization

Next, in Chapter II we provide the background for all the technical concepts introduced in this chapter and provide an overview of the existing literature on these concepts. In Chapter III, we explore various distributed system models, examine the characteristics of idealized partially-synchronous models that are favored by algorithm designers and yet render these models incongruent with empirical systems, and provide a precise specification for a non-trivial subset of empirical distributed systems. In Chapter IV, we describe the methodology used to implement the idealized partially synchronous models on top of the empirical distributed system model described in Chapter III. Chapter V provides the formal framework for specifying distributed system models and algorithms. Failure detectors are specified within this formal framework in Chapter VI. In Chapter VII idealized models of partial synchrony are constructed using failure detectors, and the construction is shown to be optimal with respect to relative solvability; we show that to implement the popular $\mathcal{M}_*$ system models, the eventually perfect failure detector $\Diamond\mathcal{P}$ is necessary and sufficient. In Chapter VIII, we implement $\Diamond\mathcal{P}$ on empirical systems, thus showing that the $\mathcal{M}_*$ models can be implemented on top of empirical systems through $\Diamond\mathcal{P}$. Finally, we present the conclusion of this dissertation, discuss the significance of the results, and outline future work in Chapter IX.

CHAPTER II

BACKGROUND AND RELATED WORK

Dante can be understood only within the context of Italian thought, and Faust would be unthinkable if divorced from its German background; but both are part of our common cultural heritage.

Nobel Lecture, 29^{th} June 1927
–Gustave Stresemann

This chapter provides an overview of the theoretical background associated with tolerating crash faults in distributed systems. Section II.1 introduces the two complementary approaches to crash-fault tolerance: *partial synchrony* and *failure detector*. Since we use failure detectors extensively in this dissertation, we focus our discussion on failure detectors. We discuss the notion of 'stronger' and 'weaker' failure detectors in Section II.2. We extend this notion of 'stronger' and 'weaker' to enable comparing failure detectors with classic problems in distributed computing in Section II.3. Then, we review implementations of popular failure detectors in various system models in Section II.4, and finally, we discuss some open questions that this dissertation answers in Section II.5.

II.1. Tolerating process crashes

A seminal breakthrough on crash fault tolerance was presented in [42] which showed that in an asynchronous system where process speeds and message delays are unconstrained, consensus[1] cannot be solved in the presence of crash faults. In simplistic

[1]Informally, in consensus, all processes start with an (independent) input value and are required to agree on a common output value. For a detailed description of the problem, see Appendix A.

terms, if processes can be arbitrarily slow and messages can take arbitrary amounts of time, then it is impossible to reliably distinguish a crashed process from a slow process, and consequently algorithms executing in such a systems cannot tolerate crash faults while solving consensus. Later, this impossibility was extended from consensus to many non-trivial and 'interesting' problems in distributed computing (for example, mutual exclusion, the dining philosophers problem, and process renaming) [41]. It was clear that in order to tolerate crash faults, algorithms would have to assume, either explicitly or implicitly, some non-trivial temporal bounds on computation and communication. Naturally, these bounds have to be guaranteed by the underlying distributed system. This insight was explored by two orthogonal, but complementary, approaches: *partial synchrony* [35] and *failure detector* [20]. While partial synchrony explored the possibility of associating explicit bounds on process speeds and message delay, failure detectors explored the possibility of encoding such temporal bounds implicitly in the form of external oracles augmenting an asynchronous system. The remainder of this section describes both approaches and related work in detail.

II.1.1. Partial synchrony

Briefly put, partial synchrony refers to distributed systems where timing bounds on communication and computation may exist, but knowledge of such bounds is limited [35]. These system models appeal to the intuition of distributed systems being 'somewhat timely'. This 'timeliness' property helps algorithms distinguish between a crashed process from a slow process, thus circumventing impossibility results for fault-tolerant distributed consensus [42] and other problems [41] in pure asynchrony.

The paper [35] which introduced partial synchrony introduced the so-called $\mathcal{M}$-models of partial synchrony which are still widely used and studied today. The $\mathcal{M}$-models of partial synchrony assert the existence of two constants which can vary

from run to run: an upper bound Δ on the maximum delay of any message, and an upper bound Φ on relative process speeds. In the $\mathcal{M}_1$ version of this model, Δ and Φ always hold, but are unknown. In the $\mathcal{M}_2$ version, Δ and Φ are known, but only hold after some unknown global stabilization time (GST).[2] Subsequently, [20] defined a composite model called $\mathcal{M}_3$ for which Δ and Φ exist, but both are unknown and only hold after some unknown GST. Like $\mathcal{M}_2$, some versions of $\mathcal{M}_3$ permit finite message loss prior to GST. During the remainder of this dissertation, we will refer to the foregoing three $\mathcal{M}$-models collectively as $\mathcal{M}_*$, and furthermore, since the three models in $\mathcal{M}_*$ are shown to be equivalent [16], we demonstrate our results relative to $\mathcal{M}_2$. The popularity of the $\mathcal{M}_*$ models among theoreticians is unsurprising. They are idealized distributed system models that captures the empirical observation and the intuitive notion that in practically deployed systems, messages are somewhat timely, and process speeds do not change arbitrarily, and yet are simple enough to admit solutions to many non-trivial problems in crash-prone distributed systems.

Since the introduction of $\mathcal{M}_*$ in [35], several models of partial synchrony have been proposed. Many such models can classified based on whether they were proposed to solve a problem or to describe an existing empirical systems.

II.1.1.1. Partial synchrony for solving problems

The $\mathcal{M}_*$ models were proposed to solve fault-tolerant agreement problems in instances where at least a majority of processes were guaranteed to be correct (not crashed) [33, 35]. To date, several variants of the $\mathcal{M}_*$ models have been proposed to solve these agreement problems. Note that the $\mathcal{M}_*$ models did not impose a lower bound on process speeds and message delays. In contrast [9, 70] proposed partially-

[2]Some versions of $\mathcal{M}_2$ permit messages to be lost prior to GST; other versions assume (like $\mathcal{M}_1$) that links are always reliable.

synchronous system models that, in addition to the assumptions made in the $\mathcal{M}_*$ models, imposed both upper and lower bounds on process speeds for solving consensus. The temporal bounds on the above system models are 'uniform' in the sense that the bounds on message delay apply to all communication links and bounds on process speeds apply to all processes. However, to achieve consensus, such uniform temporal bounds are not necessary. One of the first partially-synchronous system models to consider non-uniform temporal bounds was proposed in [7] in which all processes execute in lock-step synchrony and there exists some correct process whose outgoing links are eventually timely; that is, eventually there is an upper bound on the message delay on these links.

Subsequent investigations into sufficient partial synchrony to solve consensus and other agreement problems focused on restricted fault environments where no more than $f < n/2$ processes crash (where n is the total number of processes). Two seemingly incomparable system models were proposed in [61] and [6] to solve consensus in such restricted fault environments. The system model proposed in [61] guarantees that eventually some correct process has f bidirectional timely links at all times. Note that the set of f timely links need not be fixed and may vary throughout the execution. Independently, it was shown in [6] that consensus can be solved in systems where some f outgoing links at some correct process are eventually timely. These two results were superseded by [51] which showed that consensus can be solved in an even weaker system where eventually some correct process has f timely outgoing links and the set of f timely links can vary throughout the execution.

All of the above system models focused on weakening the real-time constraints on communication and computation in the pursuit of the 'weakest' model to solve agreement problems. The motivation for this pursuit is straightforward. 'Weaker' system models have less stringent constraints on communication and computation.

Therefore, 'weaker' models are more general and algorithms that behave correctly in such weak models behave correctly in a larger set of empirical models, and thus the pursuit of the 'weakest' system models to solve consensus provides us with an algorithm that will solve consensus in a maximal set of empirical systems.

An independent course of investigation on the 'weakest' system model to solve agreement problems pursued a time-free path in which system-model properties were divorced from real-time based bounds on communication and computation. Instead, these models focused on the relative ordering of messages and computational steps. The first such system model was proposed in [63] for systems consisting of n processes with at most f crash faults. Executions in the system model in [63] progress in "rounds" (the notion of a round is local to each process, not global), and processes send messages to all other processes in each round. A round terminates at a process when the process has received messages from $n - f$ processes for that round. The model stipulates that there exists some correct process i such that eventually some fixed subset consisting of f processes receive a message from i in each of their rounds. Subsequently, a weaker system model (and weakest-to-date) was proposed in [8] which permits this subset consisting of f processes to vary over time, as long as (eventually) at all times such a subset exists.

Note that the above system models were proposed to solve a specific problem: consensus. The particulars of these system models do not take the behavior of empirical systems into account. In contrast, there is a significant body of work which focuses on modeling various empirical systems. We review such system models next.

II.1.1.2. Partial synchrony for empirical systems

One of the first models of partial synchrony that focused on fidelity to empirical systems is the *timed asynchronous distributed system model* (or, for short, the *timed*

model) [27]. The timed model assumes that all the problem specifications prescribe a real-time deadline for progress, communication among processes is unreliable, processes may crash, and all processes have access to clocks with bounded drift rate. Additionally, the timed model assumes that the system may behave like an asynchronous system with unbounded message delay, message loss, and process speeds, infinitely often; however, there are 'good' periods of sufficient duration in which computation and communication is 'timely' so that the various system services can make sufficient progress to deliver the necessary outputs to the user. The results in [27] claim that this model describes a distributed system built with a network of workstations and provide actual measurements of message delay, message loss, process scheduling delays, and hardware clock drifts as evidence for the fidelity of the timed model to an empirical system built from networked workstations. Subsequently, it was shown that such systems could build 'almost' synchronous systems [38]. The timed model, although very useful in analyzing a locally connected network of workstation, fails to describe larger, more complex and heterogeneous distributed systems that our research is interested in understanding.

When subject to denial-of-service attacks, the distributed system of networked workstations are no longer described by the timed model, instead [40] claims that they are described by the so called *finite average response time* model (or, for short, FAR). The FAR model assumes that there exists an upper bound on absolute process speeds and, while delay experienced by an individual message may be unbounded, the average delay experienced by all messages is bounded. Unfortunately, the communication reliability is assumed to be *stubborn* [48] (which guarantees delivery of a message if it is the only message on transit), and, to our knowledge, this communication reliability property is unrealistic for many empirical systems (with, perhaps, the exception of collision-prone broadcast networks like wireless networks and Ethernet).

Not unlike the efforts to develop system models to solve consensus, there has been an independent investigation into time-free system models to describe empirical systems. Two such prominent time-free system models are the Θ-model [50] and the asynchronous bounded cycle (ABC) model [73]. Both models were developed to describe tightly coupled distributed systems consisting of multiple processors connected by a common bus.

The Θ-model was one of the first time-free system models used to describe an empirical system. The models bounds the ratio Θ of the end-to-end communication delay of messages that are simultaneously in transit. The variant of the Θ-model where Θ is known was shown to be sufficient to achieve lockstep execution[3] and solve clock synchronization[4] with bounded precision and accuracy [83]. Subsequently [50] showed that the solutions in this model can be transferred to empirical systems and the solution, despite being time-free, exhibits behavior consistent with real-time based algorithms. However, if Θ is unknown, then [84] showed that the Θ-model can implement eventual lockstep execution and clock synchronization with bounded precision, but synchronization accuracy could not be guaranteed in the presence of arbitrary number of process crashes [84].

An apparently weaker alternative for the Θ-model is explored in [73] with the ABC model. The ABC model imposes a restriction on the ratio of the number of messages that can be exchanged between pairs of processes in certain "relevant" segments of an asynchronous execution. The results from [73] showed that the ABC model is strong enough to implement eventual lockstep execution, and furthermore, algorithms that are correct in the Θ-model, where the value of Θ is unknown, behave

[3]Lockstep execution is defined in Section III.1.2.

[4]See Appendix A for detailed description of the clock synchronization problem.

correctly in the ABC model as well.

The analysis on system models like the timed model, the FAR model, the Θ-model, and the ABC model all focus on modeling a specific deployment of empirical systems and are not known to be general enough to serve as an abstract model for a sufficiently large group of empirical system. This dissertation proposes such a model in Section III.1.4.2.

An alternate mechanism to tolerate crash faults is failure detectors.

II.1.2. Failure detectors

Failure detectors [20] may be viewed as unreliable distributed system services, or oracles, that provide (potentially incorrect) information about process crashes in the system. Each process has access to a local failure detector module which applications can query locally. These failure detectors can make mistakes and provide incorrect information about processes crashes by either failing to suspect crashed processes or (falsely) suspecting correct processes. Despite such mistakes, when certain classes of failure detectors are provided as system services to processes in an asynchronous systems, many classic problems that are otherwise *not* solvable in crash-prone asynchronous systems become solvable.

A significant benefit of failure-detector-based algorithms is that they achieve an essential separation of concerns between abstract detection properties and concrete detection mechanisms. To be implemented in practice, most failure detector classes require some degree of partial or even full synchrony. The timing assumptions and mechanisms for fault detection, however, are encapsulated by the failure detector itself. Since failure-detector-based algorithms depend only on the assertional *properties* of detection, they are effectively decoupled from the underlying implementation mechanisms, network timing parameters, and reliability assumptions.

The canonical investigation into unreliable failure detector oracles [20] proposed eight failure detectors[5] among which three failure detectors gained a significant popularity: *eventually strong* failure detector ($\Diamond\mathcal{S}$), *eventually perfect* failure detector ($\Diamond\mathcal{P}$), and *perfect failure* detector ($\mathcal{P}$). Briefly, the eventually strong failure detector ($\Diamond\mathcal{S}$) guarantees that every crashed process is eventually and permanently suspected by all correct processes, and *some* correct process is eventually and permanently trusted by all correct process. The eventually perfect failure detector ($\Diamond\mathcal{P}$) guarantees that every crashed process is eventually and permanently suspected by all correct processes, and all correct processes are eventually and permanently trusted by all correct processes. The perfect failure detector ($\mathcal{P}$) guarantees that all crashed processes are eventually and permanently suspected by all correct processes, and no process is suspected before it crashes.

Observe that $\Diamond\mathcal{P}$ provides more reliable information than $\Diamond\mathcal{S}$; $\Diamond\mathcal{S}$ may suspect a correct process permanently at all processes, but $\Diamond\mathcal{P}$ is required to stop suspecting correct processes at all processes. Hence, $\Diamond\mathcal{S}$ is said to be *weaker* than $\Diamond\mathcal{P}$. Similarly, since $\Diamond\mathcal{P}$ may wrongfully suspect a correct process (albeit for only a finite duration and only finitely many times), but $\mathcal{P}$ never suspects a process that is not crashed, $\Diamond\mathcal{P}$ is weaker than $\mathcal{P}$. Thus, some failure detector classes may be ordered based on their relative 'strength'. This notion of stronger and weaker failure detector will become clear after we see some explicit examples of problems solvable by failure detectors next.

The eventually strong failure detector, $\Diamond\mathcal{S}$, is arguably one of most popular failure detector. It was shown in [20, 49] that $\Diamond\mathcal{S}$ is powerful enough to solve many problems including consensus, leader election, terminating reliable broadcast, and

[5]These failure detectors are discussed in detail in Chapter VI. For now, an informal description of a few suffices.

atomic commit in crash-prone asynchronous systems where at least a majority (more than half) of the processes are correct. Subsequently, an equivalent failure detector Ω was introduced in [19] to solve consensus. The Ω oracle, when queried, outputs a single process ID; Ω guarantees that after some (potentially unknown) time the outputs at all correct processes at all times is the process ID of some unique correct process. Although Ω provides outputs that are different from $\Diamond\mathcal{S}$, both failure detectors provide the same information about process crashes [19].

We claimed earlier that $\Diamond\mathcal{S}$ is weaker than $\Diamond\mathcal{P}$. To show that $\Diamond\mathcal{P}$ is, in fact, stronger than $\Diamond\mathcal{S}$, we are required to show that $\Diamond\mathcal{P}$ can solve all the problems solvable by $\Diamond\mathcal{S}$, and that there exist problems that can be solved by $\Diamond\mathcal{P}$ but remain unsolvable by $\Diamond\mathcal{S}$. The former task is straightforward because all outputs of $\Diamond\mathcal{P}$ are valid outputs of $\Diamond\mathcal{S}$ as well. For the latter, the separation in solvability between $\Diamond\mathcal{S}$ and $\Diamond\mathcal{P}$ failure detectors was provided in [66] through the dining philosophers problem. The results in [66] showed that any $\Diamond\mathcal{S}$-based solution to the dining philosophers problems cannot guarantee any progress by the neighbors as well as the neighbors of neighbors of a crashed process. However, with $\Diamond\mathcal{P}$, it is possible to ensure that all the processes that are not immediate neighbors of a crashed process make progress[6]. In fact, $\Diamond\mathcal{P}$ was shown to be powerful enough to solve many more problems that are unsolvable with $\Diamond\mathcal{S}$: stable leader election [4], quiescent reliable communication [3], wait-free non-blocking contention management [47], wait-free eventual weak exclusion [69], crash-locality-1 dining philosophers [68], and wait-free eventually k-bounded schedulers under eventual weak exclusion [78][7].

Similarly, we know that $\mathcal{P}$ is stronger than $\Diamond\mathcal{P}$ because we can implement per-

[6]This variant of dining philosophers problem is called crash locality-1 dining.

[7]The problems in this list are described in Appendix A.

petual lockstep execution of processes with $\mathcal{P}$ which is impossible with $\Diamond\mathcal{P}$ whereas all problems solvable by $\Diamond\mathcal{P}$ are also solvable by $\mathcal{P}$ because every output of $\mathcal{P}$ is also a valid output of $\Diamond\mathcal{P}$.

II.2. Comparing failure detectors

The previous section provided an ostensive description for comparing failure detectors based on the notion of "weaker" and "stronger". In this section, we discuss such comparisons in detail. Unlike partial synchrony, which lacks a precise notion of "weaker" and "stronger" system models, failure detectors have a precise notion of what it means for a failure detector (say) $\mathcal{D}_1$ to be weaker (or stronger) than a failure detector (say) $\mathcal{D}_2$.

A failure detector $\mathcal{D}_1$ is said to be weaker than a failure detector $\mathcal{D}_2$ if all the problems solvable by $\mathcal{D}_1$ are solvable by $\mathcal{D}_2$. By definition, if $\mathcal{D}_1$ is weaker than $\mathcal{D}_2$, then $\mathcal{D}_2$ is stronger than $\mathcal{D}_1$. Based on this definition, it is easy to see that $\Diamond\mathcal{S}$ is weaker than $\Diamond\mathcal{P}$, and $\Diamond\mathcal{P}$ is weaker than $\mathcal{P}$. It is also obvious that $\Diamond\mathcal{S}$ is weaker than Ω. However, for $\mathcal{D}_1$ to be *strictly weaker* than $\mathcal{D}_2$, it is necessary that (in addition to $\mathcal{D}_1$ being weaker than $\mathcal{D}_2$) there exist some problem that is solvable by $\mathcal{D}_2$ but not solvable by $\mathcal{D}_1$; an analogous definition of *strictly stronger* holds. Again, from Section II.1.2 we know that $\Diamond\mathcal{S}$ is strictly weaker than $\Diamond\mathcal{P}$, and $\Diamond\mathcal{P}$ is strictly weaker than $\mathcal{P}$. However, $\Diamond\mathcal{S}$ is *not* strictly weaker than Ω. If $\mathcal{D}_1$ is weaker than, but not strictly weaker than, $\mathcal{D}_2$, then $\mathcal{D}_1$ is said to be *equivalent* to $\mathcal{D}_2$. Therefore, $\Diamond\mathcal{S}$ is equivalent to Ω. In fact, if $\mathcal{D}_1$ and $\mathcal{D}_2$ are equivalent, then all the problems solvable by $\mathcal{D}_1$ are solvable by $\mathcal{D}_2$, and vice versa.

Thus far, we have established the "[strictly] stronger than" and "[strictly] weaker than" relations between failure detectors. This allows us to ask the following question:

"given that a failure detector $\mathcal{D}_1$ can solve a problem $\mathcal{A}$, does there exist a failure detector $\mathcal{D}_2$ that is strictly weaker than $\mathcal{D}_1$ and yet solve $\mathcal{A}$?" This question motivates the need for comparing a failure detector $\mathcal{D}$ with problem $\mathcal{A}$ and establish the notion of the "weakest failure detector" for a problem $\mathcal{A}$. We discuss such comparisons next.

II.3. Comparing failure detectors with problems

The Ω failure detector (which is equivalent to $\Diamond\mathcal{S}$) was introduced in [19] and shown to be the *weakest* failure detector to solve consensus in asynchronous systems where a majority of the processes are guaranteed to be correct. Similarly, [28] introduced the quorum failure detector[8] Σ and showed that (Ω, Σ) together are the weakest to solve consensus in asynchronous systems where an arbitrary number of processes may crash. Next, [29] showed that the trusting failure detector[9] $\mathcal{T}$ is the weakest failure detector to solve crash-locality 0 dining philosophers problem in asynchronous systems with a majority of correct processes, and [15] showed that $(\mathcal{T}, \Sigma^l)$ (where Σ^l is a variant of the quorum failure detector in which only outputs at live processes are required to have non-empty intersections) together are the weakest to solve the same problem in environments with an arbitrary number of process crashes. All of the above results use the same methodology (introduced in [19]) to establish their respective results.

---

[8]The quorum failure detector outputs a set of trusted processes when queried. The quorum failure detector guarantees that eventually, the outputs of the failure detector will contain only correct processes, and for every pair of processes (x, y) and every pair of times (t_1, t_2), the output of the failure detector at x at time t_1 and the output of the failure detector at y at time t_2 have a non-empty intersection.

[9]The trusting failure detector $\mathcal{T}$ guarantees that every crashed process is eventually and permanently suspected by all correct processes, every correct process is eventually and permanently trusted by all correct processes, and if a process that was trusted in past is suspected, then that process is guaranteed to have crashed. In other words, $\mathcal{T}$ may make a mistake by falsely suspecting a correct process continuously for a finite period of time from the beginning. However, once a process is trusted by $\mathcal{T}$, then suspicion by $\mathcal{T}$ implies knowledge (and not just suspicion) of crash.

We describe this methodology next.

In order to show that a failure detector $\mathcal{D}$ is the weakest to solve a problem $\mathcal{A}$, we have to demonstrate two results: (1) the problem $\mathcal{A}$ is solvable in an asynchronous system augmented with $\mathcal{D}$, and (2) any failure detector $\mathcal{D}'$ that solves $\mathcal{A}$ is stronger than $\mathcal{D}$. Demonstrating the first result is methodologically straightforward; the task is accomplished by developing an algorithm that queries $\mathcal{D}$ to solve $\mathcal{A}$ in an asynchronous system. The second result can be demonstrated as follows.

For the purpose of contradiction, let there exist a failure detector $\mathcal{D}'$ which is strictly weaker than $\mathcal{D}$ and yet can also solve $\mathcal{A}$. If, using a black-box solution to $\mathcal{A}$, we construct failure detector $\mathcal{D}$, then we argue the following. By hypodissertation, $\mathcal{D}'$ can implement $\mathcal{A}$, but (by construction) $\mathcal{A}$ can, in turn, implement $\mathcal{D}$. By transitivity, $\mathcal{D}'$ can also implement $\mathcal{D}$. Therefore, $\mathcal{D}'$ can solve all the problem solvable by $\mathcal{D}$. Hence, $\mathcal{D}'$ cannot be strictly weaker than $\mathcal{D}$. Thus, we establish the contradiction and conclude that $\mathcal{D}$ is, in fact, the weakest failure detector to solve $\mathcal{A}$.

Thus, demonstrating that $\mathcal{D}$ is the weakest failure detector to solve $\mathcal{A}$ involves two action items: (1) demonstrate that $\mathcal{D}$ can solve $\mathcal{A}$, and (2) demonstrate that $\mathcal{A}$ can solve $\mathcal{D}$.

This methodology has been used to show other weakest failure detector results such as: $\Diamond\mathcal{P}$ is the weakest to solve wait-free contention management [47], wait-free eventual weak exclusion [76], and wait-free eventually-fair mutual exclusion [79].

Thus far, we have established comparisons between failure detectors, comparisons between a failure detector and a problem, and we have seen existing results that show how failure detectors can be used to solve problems otherwise unsolvable in asynchronous systems subject to crash faults. However, all these results risk practical irrelevance unless we can establish realistic implementations of failure detectors, which we explore next.

II.4. Failure detector implementations

Recall that many problems in distributed computing are unsolvable in crash-prone asynchronous message passing systems. However, if the system is partially synchronous (that is, it guarantees appropriate temporal bounds on computation and communication) then many such problems become solvable despite process crashes. Alternatively, if the asynchronous system is augmented with appropriate failure detectors, then the same problems become solvable (in the presence of process crashes). From the above three results we know that most failure detectors cannot be implemented in asynchronous systems. To be implemented in practice, most failure detector classes require some degree of partial or even full synchrony.

Recall from Section II.1.1 that system models with less stringent constraints on communication and computation are said to be 'weaker' with less 'synchronism'. Also recall that 'weaker' system models are less restrictive and hence algorithms that behave correctly in such weak models behave correctly in a larger set of empirical models. Consequently, there has been a significant body of research focusing on increasingly weaker models of communication and computation for implementing failure detectors in empirical systems. This section reviews the results from such research.

A significant amount of the aforementioned body of research focuses on $\Diamond\mathcal{S}$ (or Ω), and $\Diamond\mathcal{P}$.[10] We discuss the results related to each of the above failure detectors in separate subsections.

---

[10]The popularity of $\Diamond\mathcal{S}$ and $\Diamond\mathcal{P}$ is not just incidental. Recall that despite apparently weak guarantees on crash fault detection, $\Diamond\mathcal{S}$ has been shown to solve consensus and other related problems [20], and $\Diamond\mathcal{P}$ has been shown to solve problems including dining philosophers [68, 69], stable leader election [4], quiescent reliable communication [3], and contention management [47].

II.4.1. Implementing $\Diamond\mathcal{S}$

The first practically feasible implementation of a failure detector was proposed in [20] which demonstrated that the $\mathcal{M}_*$ models with unknown and eventual bounds on relative process speeds and message delay possess sufficient 'synchronism' to implement $\Diamond\mathcal{P}$, and since $\Diamond\mathcal{P}$ is strictly stronger than $\Diamond\mathcal{S}$, the above result applies to $\Diamond\mathcal{S}$ as well. Subsequently, the system models described in Section II.1.1.1 were shown to be sufficient to implement $\Diamond\mathcal{S}$. We briefly revisit these models.

It was shown in [5, 7] that $\Diamond\mathcal{S}$ can be implemented in a system model where all processes execute in lock-step synchrony and there exists some correct process whose outgoing links eventually timely; that is, eventually there is an upper bound on the message delay on these links. Subsequently, focus shifted to the weakest system model to implement $\Diamond\mathcal{S}$ (or the failure detector Ω [19] which is equivalent to $\Diamond\mathcal{S}$) in environments where up to f processes may crash for some known f. Results in [61] showed that Ω (and hence, $\Diamond\mathcal{S}$) can be implemented in system models where eventually some correct process has f bidirectional links at all times. Note that the set of f timely links need not be fixed and may vary throughout the execution. Independently, it was shown in [6] that Ω can be implemented in systems where some f outgoing links at some correct process is eventually timely. These two results were superseded by [51] which showed the Ω can be implemented in systems where eventually some correct process has f timely outgoing links and the set of f timely links can vary throughout the execution.

II.4.2. Implementing $\Diamond\mathcal{P}$

Recall that in [20], $\Diamond\mathcal{P}$ was implemented in the $\mathcal{M}_*$ models with unknown and eventual bounds on relative process speeds and message delay over all communication

links. In [4], the system model was weakened to permit arbitrary message delay on all links except the links to/from some correct process which are required to be eventually timely. Independently, [58] provided an alternate system model, weaker than $\mathcal{M}_*$, to implement $\Diamond\mathcal{P}$ in which relative processes speeds are eventually bounded and only some subset of the links that form a virtual ring of correct processes are required to be eventually timely. Subsequently it was shown in [39, 40] that it is sufficient if there is some upper bound on the *average* transmission delay of messages in the system with an unknown upper bound on absolute process speed. Note that none of these system models permit infinite message loss. In fact, to the best of our knowledge, there is no proposed system model that permits an infinite subset of messages to be lost or arbitrarily delayed and yet is sufficient to implement $\Diamond\mathcal{P}$.

II.4.3. On the weakest system models

As is evident from the above results, there has been a concerted effort to identify the 'weakest' system models to implement the various failure detectors that solve relevant problems in crash-prone distributed systems. This effort is motivated, in part, by the conjecture that axiomatic properties of a failure detector codify the temporal guarantees provided by an 'equivalent' partially synchronous system. Many recent results on the weakest system models for failure detectors have met with limited success partly because the proposed system models assume real-time based bounds on communication (and possibly computation too), whereas failure detectors do not encapsulate real time.

II.4.3.1. Failure detectors and real time

In order to understand why failure detectors do not encapsulate real time, for the sake of the following argument, let us assume that the outputs of the failure detector

contain information concerning the real-time properties of the underlying partially synchronous system. Such information can be either (1) explicitly embedded in the output of the failure detector itself, or (2) implicitly expressed by guaranteeing certain real-time bounds on the delay between a crash and its suspicion, the delay between a false suspicion and the subsequent correction, and so on (cf. [24]). Next, we see why failure-detector based algorithms cannot use this information to advantage.

Consider the system model assumptions within which failure detectors operate. Failure detectors are augmented to asynchronous systems: systems where process speeds and message delays can vary arbitrarily, and processes do not have access to (global or local) clocks. Since algorithms executing in asynchronous systems do not have access to a clock, the algorithms cannot take advantage of any explicit real-time information embedded in the failure detector output.

If failure detectors that provide real-time information implicitly, algorithms could still not use it to advantage. Note that (1) the real-time duration between two consecutive steps by a process is unbounded in asynchronous systems, and (2) the failure-detector based algorithm is required to behave correctly in executions where the failure detector is queried increasingly infrequently. Therefore, a process may not query the failure detector throughout the duration that the failure detector output changes with real-time bounds on suspicion and trust accuracy. The algorithm is not guaranteed to query the failure detector within any bounded duration of time, and the algorithm could potentially 'miss' all the 'timely' changes in the outputs of the failure detectors. Hence, the algorithm cannot make use of any implicit information expressed in the real-time bounds on the changes in the failure detector output.

Hence, failure detector can, at best, quarantine the real-time based properties of the underlying partially synchronous systems; failure detectors do not encapsulate or codify real-time information in a usable manner. For a more detailed analysis of the

properties of failure detectors in relation to partial synchrony and real time, see [23].

II.4.3.2. Fairness-based system models

Since failure detectors do not encapsulate real-time, the focus on the so-called 'weakest' system models for implementing failure detectors moved away from real-time based systems to the so-called *fairness*-based system models. These models do not impose real-time based bounds on computation and communication. Instead, the temporal bounds on computation and communication are expressed as bounds on the relative ordering of computation and communication events in the systems. These time-free models are better explained with the examples provided below.

The first fairness-based system model was proposed in [63] for implementing $\Diamond\mathcal{S}$. The system model consists of n processes with at most f crash faults in which executions progress in "rounds" (the notion of a round is local to each process, not global), and processes send messages to all other processes in each round. A round terminates at a process when the process has received messages from $n - f$ processes for that round. The model guarantees that there exists some correct process i such that eventually some fixed subset consisting of f processes receive a message from i in each of their rounds. Subsequently, a weaker system model (and weakest-to-date) was proposed in [8] which permits this subset consisting of f processes to vary over time, as long as (eventually) at all times such a subset exists.

For implementing $\Diamond\mathcal{P}$, the weakest fairness-based message passing model known to date are sufficient for implementing $\Diamond\mathcal{P}$ in environments with at least two correct processes are the Θ-model [50] and the ABC model [73]. Recall that the Θ-model bounds the ratio of the end-to-end communication delay of messages that are simultaneously in transit, while the ABC model imposes a restriction on the ratio of the number of messages that can be exchanged between pairs of processes in certain

"relevant" segments of an asynchronous execution.

Note that all the above proposed system models, while claiming to be weakest to-date to implement their respective failure detectors, do not claim to be *the weakest* to do so. The closest result to the 'weakest' message-passing system model to implement $\Diamond\mathcal{S}$, $\Diamond\mathcal{P}$, and other eventually accurate failure detectors is [16] which follows an approach intermediate between the real-time-based and fairness-based approaches. The results in [16] demonstrate that with respect to solvability $\Diamond\mathcal{S}$, $\Diamond\mathcal{P}$, and other failure detectors are "equivalent" to various partially synchronous models. The authors of [16] are aware that their transformations do not preserve bounds on real-time message delay. They claim that the bounds on message delay is preserved in a 'relativistic' sense (in the extended technical report [17]), but they do not expound on the interpretation of the term 'relativistic'.

II.5. Open questions

The state of the art in partial synchrony and failure detectors leave several open questions. In section outlines the specific open questions that are addressed and resolved by this dissertation.

The first open question is the issue of fidelity and relevance of the idealized models of partial synchrony favored by theoreticians and algorithm designers. We will see in Section III.2 that the primary reason for lack of fidelity between the idealized models and empirical systems is that the temporal constraints in these idealized models are denominated in real-time units. The results in this dissertation demonstrate that adopting a fairness-based specification of temporal constraints allows us to establish the fidelity of the idealized partially-synchronous models with empirical systems.

The second open question is the issue of the 'weakest' system models to im-

plement various failure detectors. As noted earlier, while there has been significant work on determining the 'weakest' system models to implement $\Diamond\mathcal{S}$ and $\Diamond\mathcal{P}$, existing results provide only a proximate specification for such system models. In this dissertation, we argue that the reason for this apparent lack of success in determining such models is that the so-called 'weakest to-date' system models the temporal constrains on communication and computation is denominated in real-time and not fairness. If we adopt a fairness-based approach to specifying partial synchrony, then establishing the weakest system models for implementing $\Diamond\mathcal{S}$ and $\Diamond\mathcal{P}$ becomes straightforward, and we demonstrate such weakest system models in this dissertation. As a corollary, we show that failure detectors encapsulate fairness in the ordering of computation and communication events, and not the real-time constraints (if any) guaranteed by the partially synchronous system.

However, in order to understand the basis for these open questions and issues, we must first understand the nature of distributed system models and partial synchrony, which is discussed in the next chapter.

CHAPTER III

DISTRIBUTED SYSTEM MODELS

A theory has only the alternative of being right or wrong. A model has a third possibility: it may be right, but irrelevant.

–*Manfred Eigen* [36]

This chapter provides an informal, but detailed, understanding of the distributed system models used in solving problems in message passing systems. We start with a description of three families of distributed system models: asynchrony, full synchrony, and partial synchrony. We invest a lot of the chapter on partial synchrony because partially synchronous models are believed to approximate the behavior of empirical systems. Based on the observed behavior of a non-trivial set of empirical systems, we propose a distributed system model that we believe describes empirical systems adequately. Then we argue why the idealized partially synchronous models, while popular in the theory of distributed systems, fail to adequately describe empirical systems.

III.1. Distributed system model classes

Various distributed system models are specified by imposing temporal bounds, varying in nature and degree, on the behavior of the processes and communication links in a system. These bounds may be absolute (specified in real time units) or relative (specified by the number of events of some particular type). First, we describe these constraints and use them to describe the different distributed system model classes. Formal definitions of these distributed system models is provided are Chapter V.

III.1.1. Temporal constraints

Temporal constraints in distributed systems are of two types: computational and communicational. Computational constraints restrict the behavior of the processes whereas communicational constraints restrict the behavior of communication links. We explore the nature of these constraints next.

III.1.1.1. Computational constraints

Computational constraints dictate the frequency with which processes execute the steps of their algorithms. This frequency may change with respect to real time, or relative to the number of steps executed by other processes. There are many variations of such computational constraints. Below are informal definitions of the constraints employed in this dissertation:

- **Lower Bound on Absolute Process Speed.** Process i has a lower bound l on absolute process speed if i executes one step at least once every l time units.
- **Upper Bound on Absolute Process Speed.** Process i has an upper bound u on absolute process speed if i executes at most one step every u time units.
- **Bounded Relative Process Speeds.** A system is said have a bound Φ on relative process speeds if, in the duration that any process executes two consecutive steps, every process that is not crashed executes no more than Φ steps.

III.1.1.2. Communicational constraints

Communicational constraints dictate the delay experienced by messages while in transit. Like computational constraints, there are many variations of communicational

constraints. Below is an incomplete list of such variants:

- **Fair-Lossy Channels.** This is the weakest communicational constraint we consider. It simply states that if an infinite number of messages are sent from process i to process j, then an infinite subset of these messages are delivered to j, a message is delivered only after it has been sent, and messages may not be duplicated or corrupted. An infinite subset of messages may be arbitrarily delayed or even dropped.

- **Reliable Channels.** Such channels deliver messages without loss, duplication, fabrication, or corruption. Note that reliable channels are a subset of fair-lossy channels.

- **Correct-Restricted Reliable Channels.** (or Correct-reliable Channels, for short) Such channels are a variant of reliable channels that behave as reliable channels only if the sender and the recipient do not crash.[1] Otherwise, the channels may behave as fair-lossy channels.

- **Bounded-Delay Channels.** These channels are reliable channels with the additional constraint that message delay never exceeds some bound Δ. Depending on the variant of the bounded-delay channel, the bound Δ may or may not be known.

Distributed system models may be arranged in a hierarchy ordered by the aforementioned temporal constraints on communication and computation. Next, we de-

[1]The concept of "correct restrictedness" was first introduced in [46] to describe problems whose specifications impose restrictions on the behavior of correct processes, but not on the behavior of faulty processes. Similarly, here we impose restriction on the behavior of channels connecting correct processes, but not on the behavior of channels connecting faulty processes.

scribe the end-points of this hierarchy and delve into the many models that occupy the space in between.

III.1.2. Synchronous system model

The strongest system model in the aforementioned hierarchy is *synchrony*. A synchronous system model guarantees that channels are always reliable, there exists a *known* bound Φ on relative process speeds, and there exists a *known* bound Δ on message delay. The knowledge of Φ and Δ enables algorithms in synchrony to use these value of these bounds to advantage. The strength of synchrony in solving distributed problems in the presence of faults is demonstrated in its ability to implement the so-called *lockstep execution*.

In a lockstep execution, processes execute their computational steps in *rounds*. All processes share a common view of the round. In each round, each process receives all the messages sent to it in the previous round, executes a single computational step, and sends at most one message to each process in the system. In effect, lock-step execution ensures that a message sent by a correct process in a given round is reliably delivered in the next round. Hence, the failure to receive an expected message, or the receipt of an unexpected message, during a given round is indicative of a fault in the system. Specifically, the sender of the putative message is faulty.

Such a strong fault detection mechanism enables synchronous systems to deploy fault-tolerant solutions to many distributed problems [10, 60]. But unfortunately, the strong guarantees provided by synchronous systems is not reflected in the guarantees provided in many empirical distributed systems. Consequently, distributed algorithms designed for synchronous systems are not guaranteed to behave correctly in empirical systems.

Since the focus of this dissertation is to solve problems in empirical systems, we

are forced to consider weaker model from the hierarchy of distributed system model. We explore the other end of this hierarchy, the asynchronous system model, next.

III.1.3. Asynchronous system model

In contrast, the asynchronous system model is the weakest system model in the aforementioned hierarchy. The only temporal guarantee that asynchrony provides is that eventually every correct process executes another step and channels are fair-lossy. Apart from the aforementioned restriction, there are no temporal bounds on absolute process speeds, relative process speeds, or message delays.

It is well known that reliable channels can be constructed using unreliable channels [1–3, 11, 37]. Therefore, it could be argued that by replacing fair-lossy channels with reliable channels, we get an equivalent specification for the asynchronous system model. However, the results from [37] clarify that the results from [1, 2, 11, 37] are valid only in fault-free executions. Therefore, in the presence of process crashes, implementing reliable channels on top of fair-lossy channels becomes impossible. However, the results from [3] inform us that, while reliable communication may be impossible, it is possible to implement correct-reliable channels using fair-lossy channels. Consequently, in this dissertation we define an asynchronous system model as a system in which every correct process executes infinitely many steps and channels are correct-reliable.

The absence of temporal guarantees makes asynchrony an attractive candidate for modeling empirical systems. Since algorithms designed for asynchrony cannot assume the existence of any guarantees on communication and computation, these algorithms execute correctly in any system that provides stronger temporal guarantees than asynchrony. Specifically, these algorithms, when deployed in empirical systems, are guaranteed to behave correctly.

However, a significant drawback of this approach is that such asynchronous algorithms have extremely limited crash-fault tolerance capabilities. In fact, asynchrony is too weak to solve many problems even under the relatively benign fault environment that permits no more than one process crash [41].

Since the focus of this dissertation is to solve these problems in the presence of crash faults, we look to other distributed system models: models whose temporal guarantees are not so strong that they fail to model empirical systems and yet not so weak that they fail to permit crash-fault tolerant solutions. Such models are collectively said to be *partially synchronous*.

III.1.4. Partial synchrony

Informally, partial synchrony refers to the multitude of system models that occupy the landscape between synchrony and asynchrony. These are system models that provide temporal guarantees on communication and computation. But these guarantees and/or their knowledge may be imprecise or unknown. This section describes the different partially synchronous system models.

Partial synchrony was introduced in [35][2] with the precise intention to circumvent the impossibility of (crash) fault-tolerant consensus in asynchrony [42] by considering system models stronger than pure asynchrony. Although originally aimed at solving consensus, partial synchrony has since become a cornerstone for developing crash-tolerant algorithms for a variety of problems (an incomplete list is available in Appendix A).

This dissertation explores partial synchrony of two kinds: fairness based, and

[2]This paper won the prestigious Edsger W. Dijkstra Prize in 2007, which is *"awarded for an outstanding paper on the principles of distributed computing, whose significance and impact on the theory and/or practice of distributed computing has been evident for at least a decade."* [82].

real-time based. We briefly explain the distinction between them before exploring fairness-based models in Section III.1.4.1 and real-time based models in Section III.1.4.2. Informally, fairness is a measure of the number of steps executed by one process relative either to the number of steps taken by another process or relative to the duration for which a message is in transit. Fairness-based partially-synchronous systems provide temporal guarantees on computation and communication relative to other computation and communication events. For instance, the bound on relative process speeds described in Section III.1.1.1 is a classic example of a fairness-based constraint. Another example of such fairness-based constraint is the Θ-model [83] which states that for all times t, there exists a bound Θ on the ratio of the end-to-end delay of all messages in transit system-wide between correct processes at time t.

In contrast, real-time based partially-synchronous systems provide temporal guarantees with respect to an external global time base that ticks or progresses independently of the events and actions within the distributed system. Examples of such real-time based constraints include bounds on absolute process speeds defined in Section III.1.1.1 and bounded-delay channels specified in Section III.1.1.2.

III.1.4.1. Fairness-based partial synchrony

As mentioned earlier, fairness-based partially synchronous systems restrict the process speeds and message delays based on the occurrence of certain actions in executions of the system. Such system models are fairly popular, and several fairness-based partially-synchronous system models have been proposed in the literature (*e.g.*,. [33, 35, 73, 83]).

Computational fairness. Here we introduce the notion of a *proc-fair* process. A process x is said to be *k-proc-fair* (where k is a non-negative integer), if, for all processes y in the system, in all intervals of time in which y takes exactly $k+1$ steps,

and x is not crashed at any instance in that duration, then x executes at least one step.

Similarly, a process x is said to be *eventually k-proc-fair*, if there exists a time after which x is *k-proc-fair*; that is, there exists a time after which, for all processes y in the system, in all intervals of time in which y takes exactly $k+1$ steps, and x is not crashed at any instance in that duration, then x executes at least one step.

Note that while other processes may be 'fair' with respect to a proc-fair process i, process i need not be 'fair' with respect to other processes; *i.e.*, a proc-fair process may execute an unbounded number of steps in the duration between two consecutive steps of a non-proc-fair process. This is an important distinction between computational fairness and bounded relative process speeds defined in [33, 35]. Bounded relative process speeds may be viewed as a special case of computational fairness where every process is (eventually) k-proc-fair.

Communicational fairness. Here we introduce the notion of a *com-fair* process. A process x is said to be *d-com-fair* (where d is a non-negative integer) if, for each process y in the system, while a message m is in transit from x to y, either (1) y takes no more than d steps, or (2) x is crashed[3].

Similarly, a process x is said to be *eventually d-com-fair* (where d is a non-negative integer) if there exists a time after which x is d-com-fair; that is, there exists a time after which, for each process y in the system, while a message m is in transit from x to y, either (1) y takes no more than d steps, or (2) x is crashed

Informally, if x is a d-com-fair process, then all the outgoing links from x are 'timely'. Note that the outgoing links from x are 'timely' only for as long as x is live; if x crashes while a message m from x to (say) y is in transit, then the recipient y may

[3]Note that the condition (b) states that the *sender* (not the recipient) of message m is crashed.

take an arbitrary number of steps before m is delivered, or m may even be dropped.

Another point of interest is that in the traditional models of partial synchrony [33, 35] bounds on message delay are measured as the number of steps executed by the *sender*; the liveness of the recipient is irrelevant to the bound on message delay. We model message delay (communicational fairness) differently for the two important reasons.

First, the traditional models of partial synchrony assume that relative process speeds are bounded. Consequently, if some live process executes a bounded number of steps while a message is in transit, then all processes execute a bounded number of steps while a message is in transit. Hence, asserting the existence of a bound on the number of steps executed by the sender while a message is in transit is equivalent to asserting the existence of a bound on the number of steps executed by the recipient while the message is in transit. However, in our case, computational fairness is not a symmetric property. Consequently, measuring and bounding message delay is subtler under our proposed fairness framework.

Second, we denominate message delay (communicational fairness) as the number of steps taken by the recipient because the notion of a message being timely or late is relevant only to the recipient of the message and not the sender. Furthermore, we bound the number of steps taken by the recipient only while the sender is live because: while the sender is not crashed, it can successfully maintain an operational communication link with the recipient, and the link can ensure that messages are delivered before the recipient takes 'too many steps'. However, if the sender crashes, then the link is no longer guaranteed to stay operational, and hence, no guarantees can be provided on message delay and delivery.

Fairness-based partially-synchronous system models. The above-described notions of com-fairness and proc-fairness can be combined to specify the following

four partially-synchronous system models:

1. *All Fair* ($\mathcal{AF}$) is an asynchronous system with the following restriction on fairness: all processes are both k-proc-fair and d-com-fair, for known k and d.

2. *Some Fair* ($\mathcal{SF}$) model is an asynchronous system with the following restriction on fairness: some correct process x is both k-proc-fair and d-com-fair, for known k and d.

3. *Eventually All Fair* ($\Diamond\mathcal{AF}$) is an asynchronous system with the following restriction on fairness: there exists some (potentially unknown) time after which all processes are both k-proc-fair and d-com-fair, for known k and d. That is, *eventually* the system behaves like $\mathcal{AF}$.

4. *Eventually Some Fair* ($\Diamond\mathcal{SF}$) is an asynchronous system with the following restriction on fairness: there exists a (potentially unknown) time after which some correct process x is both k-proc-fair and d-com-fair, for known k and d. That is, *eventually* the system behaves like $\mathcal{SF}$.

III.1.4.2. Real-time based partial synchrony

As mentioned earlier, real-time based partially-synchronous systems provide temporal guarantees on communication and computation with respect to an external global time base (real time) that ticks or progresses independently of the events and actions within the distributed system. Real-time based systems are particularly useful in modeling empirical distributed systems owing to the latter's specification with respect to real time. Examples of such real-time based models have already been discussed in Section II.1.1.2, and hence, will not be repeated here. However, recall that Section II.1.1.2 concluded that the models are not general enough to serve as an

abstract model for a sufficiently large group of empirical systems. We address this issue by specifying a new system model that describes a significant subset of empirical systems based on their real-time constraints.

The empirical distributed systems are one of two types: *physical systems* and *overlay systems*. Informally, physical systems are essentially the communication and processor infrastructure at the lowest physical (hardware) level[4]. Overlay systems are distributed (software) architectures built on top of physical systems[5].

It should be noted that the list of empirical distributed systems is too diverse for any single family of system models to describe all of them adequately. Since one of the goals of the dissertation is to abstract multiple distributed systems into a single family of system models, we limit the collection of distributed systems to a subset that shares sufficiently common properties and can be abstracted by a single family of system models. We do so by first considering a subset of physical systems which satisfies a given set of properties and include all overlay systems that satisfy the same properties.

The physical systems under consideration satisfy the following properties:

Computational constraints. The system consists of a finite fixed set of processes. Processes may crash at any time without warning and never recover. While absolute process speeds of correct processes may be arbitrary, but finite, the ratio of process speeds is always bounded above; that is, there exists an upper bound (that is potentially unknown) on relative process speeds. Additionally, all processes have *local real-time clocks* that are not necessarily synchronized, but they can approximately

[4]Examples of physical systems are multi-core processors, network hardware infrastructure, and tightly and loosely coupled computing clusters.

[5]Examples of overlay systems include the Internet, distributed operating systems, distributed databases, and Grid computing applications.

measure intervals of real time with an (unknown) upper bound ρ on the *drift rate*.

Communication constraints. Communication links transport messages between the processes they connect. These links are assumed to be point-to-point links. That is, communication links connect exactly two processes with each other. The links are also assumed to be *fair lossy* with some additional constraints on message loss and delay, which are described next.

While messages might be lost in empirical communication links, the operational parameters of the system dictate that such behavior is limited to some proper subset of messages sent on the links. Furthermore, in empirical systems, a significant subset of messages are delivered reliably and 'on time', and such timely messages are 'not too sparse'. The notions of 'on time' and 'not too sparse' merit explanation.

Being 'on time'. In empirical distributed systems, communication interfaces often have hardcoded real-time bounds on the expected duration for a message to be delivered and an acknowledgment received at the sender. If the acknowledgment does not arrive within that duration, the message is assumed to be undelivered and is resent. In reality, the maximum delay experienced by an 'on-time' message depends on parameters like link bandwidth, signal propagation delay, and queuing delays. The link bandwidth and signal propagation delay on a link are determined by the hardware and are usually static[6]. Queuing delay depends on the number of messages in transit in the system. Often, priority-based routing is used to ensure that some subset of the messages (usually, control messages) in transit experiences little or no queuing delay. For the purposes of our characterization of communication links, such high-priority messages may be considered to be 'on time' messages as well. Consequently, accept-

[6]In a well maintained and administered system hardware upgrades may increase the link bandwidth and decrease the propagation delay. These changes serve to decrease the delay of messages, and hence do not result in an otherwise 'on time' message being late.

able delay of a message to be 'on-time' is a moving target whose value depends on all the aforementioned parameters. While the value of the target may be unknown and variable, it nevertheless exists. Based on the above arguments, we characterize the notion of being 'on time' as follows: the average message delay of all 'on time' messages sent within every bounded window of time (the size of this window, although fixed, can vary from hours to days to weeks, and need not be known *a priori*) does not exceed some unknown real-time value δ.

Being 'not too sparse'. It is not uncommon to observe 'on time' messages in empirical distributed systems. In other words, if an 'on time' message is witnessed at time t, then it will not be too long after t that another 'on time' message is witnessed. However, the maximum gap between two consecutive 'on time' messages is not fixed and varies depending on various factors like channel capacity, network congestion, denial-of-service attacks, unplanned outages, and so on. But in short, the maximum gap between 'on time' messages is smaller during 'good periods' and larger during 'bad periods'. Also, transitions between 'good' and 'bad' periods are not instantaneous; there is always a transient period between good and bad periods. Finally, the 'good periods' are not too far apart; that is, 'bad periods' are relatively short-lived and bounded in duration. Based on the above observations, we can characterize the notion of 'on time' messages being 'not too sparse' as follows: the average number of arbitrarily delayed or dropped messages over every fixed window of time (the size of this window, although fixed, can vary from hours to days to weeks, depending on the empirical system and the deployed environment, and need not be known *a priori*) does not exceed some unknown positive integer r.

Typically the size of the windows described above are large enough to encompass the 'good', 'bad', and transient periods, as well periods of high and low system loads.

ADD channels. The aforementioned constraints introduce a new communication

channel model called Average Delay/Drop (ADD) Channels[7]. Intuitively, ADD channels combine the message-loss property of fair-lossy channels with the timeliness of bounded-delay channels. These channels can be viewed as fair-lossy channels with additional constraints on message delays of some subset of messages. All messages sent on an ADD channel can be logically partitioned into two disjoint sets: *privileged* and *non-privileged*. ADD channels provide no guarantees on delivery or delay of non-privileged messages. As such, infinitely many messages may be arbitrarily delayed or even dropped. By contrast, ADD channels provide the following guarantees for privileged messages:

1. If a process i sends infinitely many messages to a correct process j on an ADD channel, then some infinite subset of those messages will be privileged. All such privileged messages will be delivered reliably to j.

2. There exists an unknown window size $w \in \mathbb{N}^+$, an unknown message delay $\delta \in \mathbb{R}^+$, and an unknown message ratio $r \in \mathbb{N}^+$, such that for every interval of time I in which exactly w privileged messages are sent on the channel:

   (a) The average delay of the w privileged messages is at most δ.

   (b) The average number of non-privileged messages sent between any consecutive pair of privileged messages in I is at most r.

Intuitively, privileged messages are delivered reliably and are neither too late nor too sparse. Privileged and non-privileged messages can be interleaved. These constraints on privileged messages can be simplified as follows. The unknown window size w and the unknown bound δ on *average* privileged delay actually induce an

[7]The specification of ADD channels as a communication channel model for empirical systems is published at [74].

unknown bound on the *absolute* delay of privileged messages; specifically, no privileged message can be delayed more than $w \times \delta = \Delta$ time units. Similarly, the unknown window size w and the unknown bound r on *average* message ratio actually induce an unknown bound on the maximum number of non-privileged messages which can be sent between any consecutive pair of privileged messages; specifically, at most $w \times r = R$ non-privileged messages can be sent between any consecutive pair of privileged messages.

The foregoing observations yield a simplified specification of the timeliness and reliability properties of ADD channels:

1. If a process i sends infinitely many messages to a correct process j on an ADD channel, then some infinite subset of those messages will be privileged. All such privileged messages will be delivered reliably to j.

2. There exist two unknown bounds $\Delta \in \mathbb{R}^+$ and $R \in \mathbb{N}^+$ such that:

   (a) The absolute delay of all privileged messages is at most Δ.

   (b) The maximum number of non-privileged messages sent between any consecutive pair of privileged messages is R.

III.2. The $\mathcal{M}_*$ models: a critique

Although there are many problem-specific partially synchronous system models mentioned in Section II.1.1.1, for the many algorithms presented in each of these models to be of practical relevance, it is important that these system models possess justifiable fidelity to empirical systems so that the algorithms are deployable in the target empirical systems (albeit with some modifications). Note that all the models from Section II.1.1.1 are derived by weakening the $\mathcal{M}_*$ models introduced in [35].

Therefore, all the algorithms in these models work correctly in the $\mathcal{M}_*$ models as well. Therefore, to understand the practical relevance of these algorithms, we first investigate the practical relevance of the $\mathcal{M}_*$ models.

Before proceeding to determining the practical relevance of the $\mathcal{M}_*$ models, we have to first determine if the $\mathcal{M}_*$ models are real-time based system models or fairness-based system models.

III.2.1. Real time versus fairness based interpretation

Interestingly, the classification of the $\mathcal{M}_*$ models of partial synchrony introduced in [35] as either fairness based or real-time based is a subject of justifiable debate. Recall that the $\mathcal{M}_*$ models assert the existence of an upper bound Δ on the maximum delay of messages and an upper bound Φ on relative process speeds.

The debate on whether the $\mathcal{M}$-models are real-time based or fairness based arises from the interpretation of Δ and Φ, which depends on the interpretation of the term *real time* in [35] which is defined as follows: "...there is a real-time clock outside the system that measures time in discrete numbered steps. At each tick of real time some processors [*sic*][8] take one step of their protocol." We first present two possible interpretations of the term *real time* and then derive the possible interpretations of Δ and Φ. In order to distinguish the vernacular notion of real time from the term *real time* defined in [35], we will refer to the former as *Newtonian time* and the latter as *real time*.

The term *real time* in [35] can be interpreted in two different ways: (1) at each tick of real time zero or more processes take exactly one step of their protocols, or (2) at each tick of real time at least one process takes exactly one step of its protocols.

[8]The term *processor* in [35] refers to the entity that is defined as a *process* in this dissertation.

Both interpretations imply that a process cannot take more than one step at each tick of real time, but the first interpretation permits real-time ticks where no process takes a step whereas the second interpretation prohibits such behavior. This is a subtle, but significant, difference between the interpretations and underpins the nature of real time, process speeds, and message delay as elucidated next.

The first interpretation allows real time to tick independently of whether or not processes take steps, and thus permits real time to be identical to Newtonian time. This allows processes to take decelerate arbitrarily with respect to real time as well as Newtonian time. However, it does not allow processes to accelerate arbitrarily. Since no more than 1 step can be taken per real-time tick, absolute process speeds is bounded above by 1 step per Newtonian-time unit. Thus, constraints on computation are inextricably linked to the progression of Newtonian time.

Similarly, the bound Δ on message delay now applies to both real time and Newtonian time. Consequently, process acceleration and deceleration do not affect the maximum Newtonian time delay that can be experienced by a message. Thus, constraints on communication are inextricably linked to the progression of Newtonian time.

Evidently, the first interpretation of real time in [35] argues that the $\mathcal{M}_*$-models are (Newtonian-time, or what we call) real-time based models of partial synchrony.

The second interpretation mandates a process-driven progression of real time. That is, real time does not move forward until a process executes a step. An extremal consequence of such an interpretation is that if all but one process in the system crash, then real time moves at the rate at which the solitary process takes steps; that is, the solitary process executes at the rate of 1 step per real-time tick regardless of the Newtonian time duration between two consecutive steps by that process. In other words, real time moves at a different rate from Newtonian time.

Within the purview of this interpretation, the bound Φ on relative process speeds requires that "in any contiguous subinterval I containing Φ real-time steps, every correct processor [*sic*] must take at least one step." [35] That is, processes may accelerate and/or decelerate arbitrarily with respect to Newtonian time, but at every instant of Newtonian time, the ratio of the speeds of the fastest process to the slowest process is guaranteed to not exceed Φ. Thus, constraints on computation are decoupled from Newtonian time completely.

Similarly, consider the bound Δ on message delay which requires that, given an interval I of real time, "if message m is placed in p_j's buffer by some $Send(m, p_j)$ at a time s_1 in I, and if p_j executes a $receive(p_j)$ at a time s_2 in I with $s_2 \geq s_1 + \Delta$, then m must be delivered to p_j at time s_2 or earlier." [35] Intuitively, it says that if a process p_i sends a message m to p_j at real time s_1, then p_j is guaranteed to receive it by real time $s_1 + \Delta$ or the earliest real time past $s_1 + \Delta$ when p_j takes a step in which it receives messages. The bound Δ on message delay applies only when Δ denotes real time duration, not Newtonian time duration. For example, suppose p_i and p_j take steps at the rate of 1 step per Newtonian time unit, and the bound on message delay is Δ real time units. Then a message sent from p_i to p_j will be delivered within Δ real-time units and Δ Newtonian-time units as long as the process speeds of p_i and p_j remain unchanged. However, if p_i sends a message to p_j and immediately p_i and p_j change their speeds to 1 step per k Newtonian-time units, then the bound on message delay is still Δ in real-time units, but it now becomes $k \times \Delta$ in Newtonian-time units. That is, as processes execute faster, messages must be delivered sooner, and as processes slow down, message may be delivered later. Thus, much like computation, constraints on communication are decoupled from Newtonian time completely.

It can be verified that this interpretation is equivalent to the $\Diamond\mathcal{AF}$ system model specified in Section III.1.4.1.

Both interpretations yield plausible and reasonable, but different and irreconcilable, models of partial synchrony.

A point of clarification: henceforth the term *real time* will refer to Newtonian time.

III.2.2. Real-time based $\mathcal{M}_*$ and empirical systems

We explore the fidelity of $\mathcal{M}_*$ models to empirical systems while adopting the real-time based interpretation $\mathcal{M}_*$ models. Although the $\mathcal{M}_*$ models are intuitively seen as an idealized formalization of empirical system behavior with 'somewhat timely' computation and 'somewhat timely' communication, the exact subset (or even a non-trivial sound subset) of empirical systems modeled by $\mathcal{M}_*$, to my knowledge, is undetermined. In order for $\mathcal{M}_*$ to be a viable candidate as a prescribed system model which can be built on empirical systems, it is imperative that $\mathcal{M}_*$ models either have high fidelity to the empirical systems, or be constructible on top of the empirical systems.

This section illustrates the inability of (real-time based) $\mathcal{M}_*$ to describe empirical systems or the systems constructible on top of empirical systems.

III.2.2.1. Absolute process speed

The definitions of $\mathcal{M}_*$ directly couple real time with absolute process speed. The definitions of $\mathcal{M}_*$ state that real time is measured in discrete integer numbered steps, and that a process can execute at most one atomic step at each time tick. This forces a *de facto* upper bound on the absolute process speed *viz.*, one atomic step per real-time tick. However, historically, we have seen process speeds increase continually. It is unclear whether assuming such a *de facto* upper bound on absolute process speed will have implications for systems whose processors are continually upgraded to higher

speeds. Ideally, system model specifications should be able to model systems whose process speeds may increase infinitely often as the systems are upgraded. Unfortunately, current $\mathcal{M}_*$ models fail to model such continual increases in absolute process speed.

III.2.2.2. Message loss

The $\mathcal{M}_*$ models assume that channels are eventually reliable: there exists a time after which every message sent to a correct process is eventually delivered. However, message loss occurs in physical systems due to various (uncontrolled) factors such as, traffic congestion, signal collision, electro-magnetic interference, high signal-to-noise ratio, and misrouting of messages; insofar as these factors are not controlled, they can precipitate message loss infinitely often, thus undermining communication reliability for an infinite suffix. In fact, message loss occurs infinitely often in many empirical overlay systems, even when the network is stable and operating as specified. Consider the following examples:

- In broadcast systems (like Ethernet), message loss results from collisions. Although broadcast communication protocols employ collision avoidance techniques, these techniques can minimize collisions, but not prevent them altogether. Therefore, when multiple processes attempt to send messages infinitely often, there is a non-zero probability of message collision occurring infinitely often, and hence potentially infinite message loss.

- In protocols like TCP, message loss is essential for congestion control and maintaining fairness in bandwidth allocation. In TCP, a process floods the network with messages until the sending process begins to timeout on acknowledgments. These timeouts are interpreted as congestion due to bandwidth hogging. There-

fore, the sending process reduces the rate of message transmission. In any network of significant size, multiple processes can have concurrent TCP sessions infinitely often, and hence can lose messages infinitely often.

Thus, message loss can occur infinitely often even in well-tuned physical and overlay systems. Therefore, unlike $\mathcal{M}_*$, partially synchronous models that capture the behavior of the physical systems characterized in Section III.1.4.2 have to accommodate some form of infinite message loss.

Note that there are overlay systems constructible on top of message-lossy empirical systems that provide reliable communication. However, such overlay systems do not guarantee bounds on message delay (as discussed next in Section III.2.4).

III.2.3. Message size

The definitions of $\mathcal{M}_*$ assume real-time upper bounds on message delay, but do not restrict message size. This implies that algorithms can send messages of unbounded size and the system guarantees a bounded real-time message delay. However, consider the observed behavior in empirical systems. In many empirical systems, the time taken to transmit a message of size $2k$ units is typically twice the time it takes to transmit a message of size k units. In fact, every communication link in empirical systems specifies the maximum data transfer rate that the link can achieve. For example, a Gigbit-Ethernet interface can transmit no more than 1 Gigabit per second. Therefore, sending messages of unbounded size on communication links which have a bounded bandwidth cannot yield an upper bound on message delay. Based on the above argument, we cannot expect empirical systems to satisfy an unconditional upper bound on message delay (especially when message size increases without bound). They may, at best, guarantee upper bounds on message delay that are proportional

to the message size.

III.2.4. Communication reliability and message delay

Recall that the definitions of $\mathcal{M}_*$ assume reliable communication *and* upper bounds on message delay. As discussed earlier, many empirical systems do not guarantee reliable communication natively. However, reliable communication may be simulated on top of empirical systems which inherently lose messages. All known implementations of reliable communication on top of unreliable channels (cf. [1,2,12]) retransmit messages sufficiently many times to ensure the delivery of at least one copy of each message.

Deterministic techniques of message retransmission to achieve communication reliability have a potential problem of message duplication. If a process p sends multiple copies of a message m_i to process q, then after q receives the first copy to message m_i, process q is expected to discard all additional copies of m_i it may receive in the future. However, if process p subsequently sends multiple copies of another message m_j which is incidentally identical to m_i, then when q receives copies of m_j, process q has no way of distinguishing a copy of m_i from a copy of m_j. Since the protocol to implement reliable communication is deterministic, process q has to take the same decision when it receives a copy of m_i and when it receives a copy of m_j because they are identical. If process q discards copies of m_i and m_j, then process p will never be able to successfully transmit m_j. On the other hand, if q accepts copies of m_i and m_j, then q may receive more than two distinct messages with identical payload when process p sent only two such distinct messages.

In order to circumvent the above problem of message duplicity, the retransmission techniques for achieving reliable communication resort to tagging each distinct message with a unique identifier (usually a sequence number). These unique identifiers enable the recipient to distinguish copies of an older message from a new message

with identical payload. However, since the sending process has no way of reliably determining if all the copies of a given message have either already been received by the recipient or dropped by the channel, the sending process cannot reuse the unique identifiers. Consequently, if two processes send messages to each other infinitely often, the unique identifiers grow without bound. Such unbounded growth in the unique identifiers results in unbounded message sizes. However, from the earlier discussion in Section III.2.3, we know that empirical systems cannot be expected to satisfy an upper-bound on message delay if the message size is unbounded. Therefore, techniques that achieve reliable communication on top of unreliable channels suffer from the potential tradeoff of losing the timeliness of the communication. Consequently, empirical systems which inherently lose messages at the physical level cannot simultaneously achieve the $\mathcal{M}_*$ behavior of reliable communication and a real-time upper bound on message delay.

III.2.4.1. Channel capacity

The definitions of $\mathcal{M}_*$ make no explicit mention of channel capacity. However, channel capacity has a significant impact on message reliability, message delay, and relative process speeds in empirical systems. In fact, empirical systems with finite channel capacity and processes running at different speeds cannot be modeled by $\mathcal{M}_*$.

All empirical systems have finite channel capacity. Channel capacity, however, could be bounded, or unbounded. We now explore the system behavior under both assumptions.

First, assuming that channel capacity is bounded, we analyze the system behavior with respect to message delay and reliability. From the definitions of $\mathcal{M}_*$, we know that the model allows processes to run at different speeds while maintaining their relative process speeds. Consider an instance of an empirical system with bounded

channel capacity where a process p is running faster than a process q. Let process p send a message to q every k time units, and let process q (being slower than p) consume a message every $2k$ time units. Let the channel capacity be C messages. Let the system start execution at time $t = 0$ with empty channels. Clearly, at time $t = 2k$, p would have sent 2 messages whereas q would have consumed just 1. Therefore, at the end of time $2k$, the channel contains 1 message in transit. The number of messages in transit increases by 1 every $2k$ time units. Therefore, at time $2k \times C$, there at C messages in transit, and the channel is now full. At time $2k \times C + 1$, process p sends another message. However, since the channel is already full, some message has to be dropped. Since p sends messages faster than q consumes them, messages will have to be dropped infinitely often. This implies that in systems with bounded channel capacity, if faster processes are allowed to send messages at any time, then channel reliability cannot be guaranteed.

Alternatively, if the sender could be blocked from sending when the channel is full, then it may be possible to guarantee channel reliability and timeliness even when processes run at different rates. However, empirical systems support asynchronous sends and buffer messages until delivered. Therefore, for $\mathcal{M}_*$ to model empirical systems, it cannot block the sender. Consequently, empirical systems with bounded channel capacity and processes running at different speeds cannot be modeled by $\mathcal{M}_*$[9].

[9]It can be argued that 'ping-pong' style protocols — where processes defer sending the next message until an ack for the previous message has been received — may be used to send and receive messages in order to guarantee message reliability in systems with bounded channel capacity. Such 'ping-pong' transactions act as flow control mechanisms to maintain a bounded number of messages in transit, thus guaranteeing zero message loss in channels that are reliable with bounded channel capacity (provided the bound on the channel capacity is sufficient). However, such 'ping-pong' protocols cannot be used in defense of $\mathcal{M}_*$ system models because not all applications are designed to work under such communication mechanisms. The specification of $\mathcal{M}_*$ does not restrict communication behavior to such 'ping-pong'

Now consider empirical systems with unbounded channel capacity; that is, at any given time the channels can accommodate a finite, but unbounded, number of messages. Revisiting the example in the previous paragraph, except that the channel capacity is unbounded, we see the following: Since the channel is never full, messages need not be dropped. However, note that after $2k \times C$ time units, there are c messages in transit. Among the C messages in transit, some message will experience a delay of at least $2k \times C$ time units (because q consumes only 1 message per $2k$ time units). Therefore, among the messages in transit at time t, some message experiences a delay of at least t time units. In other words, over the lifetime of the network, there exists no upper bound on message delay[10]. Therefore, any empirical system with unbounded capacity with processes running at different speeds cannot be modeled by $\mathcal{M}_*$.

III.2.4.2. Final analysis

Based on the above preliminary illustrations, we can conclude that real-time based $\mathcal{M}_*$ models do not model empirical systems, or any systems constructible on top of existing empirical systems. Therefore, in order to build idealized models of partial synchrony *à la* $\mathcal{M}_*$, our only remain option is to view $\mathcal{M}_*$ and its derivative models as fairness-based models of partial synchrony[11].

---

based communication, hence the applications designed for $\mathcal{M}_*$ models need not be designed to work under a 'ping-pong' protocol. Therefore, the 'ping-pong' protocols have limited applicability under $\mathcal{M}_*$, and hence are not a viable defense for $\mathcal{M}_*$.

[10]Note that the alternative is to block the sender, but we have already established that empirical systems do not block the sender, and hence cannot be used as a defense to achieve the upper bound on message delay.

[11]Recall that the system model $\Diamond\mathcal{AF}$ is identical to a fairness-based interpretation of $\mathcal{M}_*$.

III.2.5. Fairness-based $\mathcal{M}_*$ and empirical systems

It is fairly straightforward to show that the fairness-based interpretation of $\mathcal{M}_*$ cannot describe any empirical systems. For instance, the fairness-based interpretation allows a message to take an arbitrary amount of real time as long as the recipient does not execute too many steps. In empirical systems, message delays are specified in real-time units, and it is unreasonable to expect communication to change depending on process speeds.

However, when we consider systems implementable on top of empirical systems, fairness-based $\mathcal{M}_*$ appears to be a viable candidate system model. This dissertation pursues this possibility to its logical conclusion. That is, we construct fairness-based $\mathcal{M}_*$ models on top of empirical systems. The methodology used to achieve this is described in the following chapter.

CHAPTER IV

METHODOLOGY

You know my method. It is founded upon the observation of trifles.

The Boscombe Valley Mystery, 1892
The Adventures of Sherlock Holmes
–Sir Arthur Conan Doyle

In Chapter I, we discussed the limited impact of algorithms and results for idealized models of partial synchrony on deploying empirical distributed systems. In Chapter III, we saw how one of the reasons for such limited impact is that these idealized models are often specified with real-time based temporal constraints on computation and communication, and such real-time based partially synchronous models do not model empirical systems, or any systems constructible on top of existing empirical systems. However, fairness-based partial synchrony does not seem to suffer from such issues. Therefore, a reasonable conjecture is that while real-time based models of partial synchrony may be unrealizable on empirical systems, fairness-based models of partial synchrony can be implemented on top of empirical systems. This chapter provides the methodology used in this dissertation to establish the aforementioned conjecture.

Our first task is to define what it means to *implement* a system model on top of a distributed system. By borrowing the concepts from the virtualization community, we define *implementing* $\mathcal{M}_*$ system models to mean creating a virtual execution environment which satisfies the properties of (fairness-based) $\mathcal{M}_*$. Such a virtual environment consists of correct-reliable communication channels and eventually fair schedulers (both of which are described next). Briefly, the methodology involves implementing correct-reliable communication channels and the scheduler by employing

an appropriate failure detector in an otherwise asynchronous environment, and then implementing the appropriate failure detector in empirical systems.

IV.1. Correct-reliable communication channels.

The constructions of reliable communication from [1–3, 11, 37] may be used to implement a *correct-reliable* communication channel on top of empirical systems. However, the $\mathcal{M}_*$ specification also requires an upper bound on the number of steps executed by the recipient of a message while the message is in transit (until the sender crashes). In order to guarantee such an upper bound, the scheduler (described next) sends messages on these correct-reliable channels only when the recipient process requests for messages, and stall the recipient process until all pending messages are received. Such a 'pull' mechanism for communication ensures that the steps taken by the recipient process is bounded for the duration that each message is in transit. However, if a message is sent from a process that has crashed, then the recipient need not wait to receive these pending messages.

IV.2. Scheduler

The scheduler forms the basic execution infrastructure of the virtual $\mathcal{M}_*$ environment. The role of the scheduler is to schedule processes such that the bound Φ on relative process speeds and the upper bound Δ on message delay is maintained. The scheduler should be able to guarantee these bounds despite process crashes in the underlying system model. Essentially, the above requirement translates to each process (say) i in the system establishing local synchronization points in time every Φ steps and ensuring that between any two consecutive synchronization points at process i, every process that is not crashed has executed at least one step. If not, process i waits

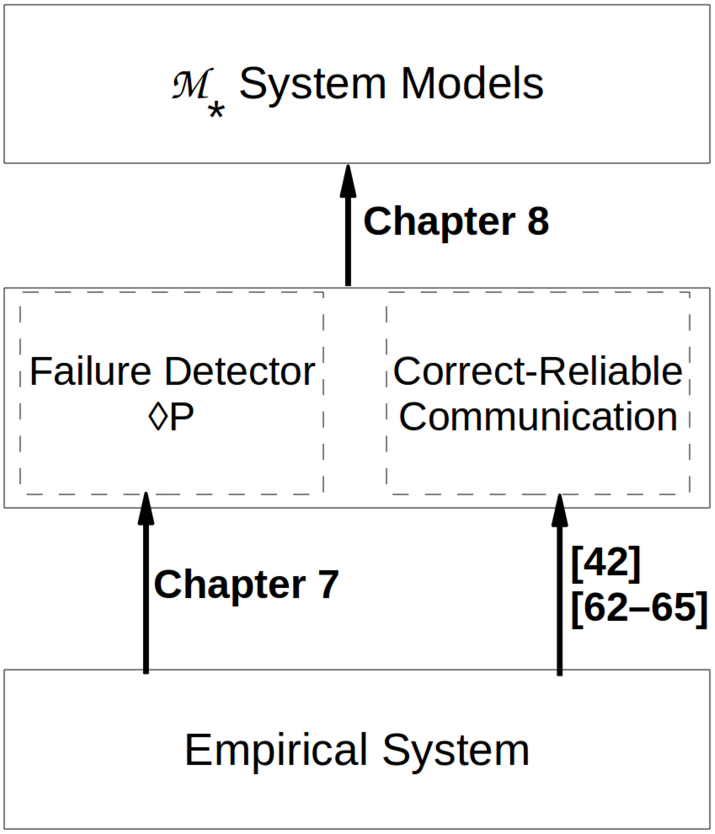

Fig. 1 The combination of constructions to implement $\mathcal{M}_*$ on top of the empirical system model.

until every process that is not crashed has executed at least one step. Similarly, every time the scheduled application at process i sends a message to process j, process i establishes a synchronization point at j such that the application at j stalls after taking Δ steps if the message from i has not arrived at j. If i crashes before the message is delivered at j, then j is required to not continue stalling. In order to determine the set of crashed processes on whom process i need not wait to execute steps, we employ a failure detector.

IV.3. Failure detectors

Recall that a *failure detector* is a distributed oracle that can be queried for (potentially unreliable) information about process crashes. Failure detectors will be employed by the scheduler implementation. A particular failure detector of interest is the eventually perfect failure detector $\Diamond\mathcal{P}$.

The eventually perfect failure detector ($\Diamond\mathcal{P}$), mentioned earlier, can give arbitrarily unreliable information about process crashes for a finite duration. However, eventually it provides perfect information about crash faults forever. We show that $\Diamond\mathcal{P}$ is necessary and sufficient to implement the $\mathcal{M}_*$ system models. Therefore, in order to implement $\mathcal{M}_*$ on top of empirical system, we need to implement $\Diamond\mathcal{P}$ on empirical systems, which completes the set of algorithmic transformations necessary to implement $\mathcal{M}_*$ models on top of empirical systems.

The combination of constructions described in this chapter are shown in Fig. 1. The $\Diamond\mathcal{P}$ failure detector construction is described in Chapter VIII, and the virtual $\mathcal{M}_*$ system model is constructed in Chapter VII.

CHAPTER V

FORMAL FRAMEWORK AND MODELING

He who would do good to another, must do it in Minute Particulars,
General Good is the plea of the scoundrel, hypocrite, and flatterer:
For Art & Science cannot exist but in minutely organized Particulars,
And not in generalizing Demonstrations of the Rational Power.

Jerusalem: The Emanation of The Giant Albion, 1820
–*William Blake*

This chapter presents the theoretical framework that will be used for definitions, specifications, analysis, algorithms, and proofs in the remainder of this dissertation. First, we start with informal descriptions of various terms and components within timed I/O automata (TIOA) model in Section V.2. Subsequently, we define the TIOA that constitute a distributed system: processes, protocols, program actions, schedulers, communication links, and fault environment in Section V.3. Then, we specify constraints on a TIOA that models the system behavior of a distributed system in Section V.4. Finally, we define various models of distributed systems specified as various restrictions on admissible system behavior in Sections V.5.

V.1. Motivation

There is a strong need to characterize failure detectors and partial synchrony within a theoretically-consistent formal framework, and unfortunately, such a framework is currently lacking. All existing frameworks either characterize partial synchrony or failure detectors in isolation, but not both. In this section, we will review existing theoretical frameworks that characterize partial synchrony, failure detectors, algo-

rithms that use failure detectors, or implementations of failure detectors. We will see how all the existing approaches are deficient, and finally, provide an overview of the new framework proposed in this chapter.

V.1.1. Existing frameworks

The first formal framework to characterize partial synchrony was proposed in [33,35]. In this framework, every process executes a single sequential thread of execution subject to the constraints imposed by the system model. This mechanism works well for executing a single algorithm to solve (say) consensus or failure detection, but the framework does not extend to the possibility or the mechanics of executing multiple threads of execution where each thread may either be subject to partial synchrony or subject to asynchrony. Recall that in failure-detector based system models, the process is assumed to have access to a failure detector module which is (ostensibly) implemented in partial synchrony, but the application that employs the failure detector executes within an asynchronous environment. Therefore, the framework in [33,35] is inadequate for the purpose of this dissertation.

The first formal framework to consider failure detectors was proposed in [20]. The framework assumed the existence of a failure-detector module at each process in an asynchronous system with reliable channels. The definition of an execution in this framework depends on (1) the times at which processes crash, (2) the outputs of the failure detector, and (3) the algorithm being executed. While this framework overcomes the issue of multiple threads of execution within a single process, the framework is silent on the actual implementation of the failure detector itself. Furthermore, the framework mandates that a failure detector provide an output at every instant of time and respond to a query instantaneously. However, no practical implementation of a failure detector can (1) provide an output at each time instant, and (2) have

zero latency between query and response. Other issues with the framework in [20] have been explored in [23]. The exposition in [23] illustrates how the requirement of valid outputs at every instant of time and an instantaneous response to a query makes it impossible for an arbitrary failure detector to be reflexive; that is, it is not always possible for a failure detector output to be repeated by a simple asynchronous algorithm and yet maintain correctness. Briefly, the problem is as follows: since the algorithms that query a failure detector execute under asynchrony, these algorithms could 'miss' a set of changes in the output of failure detector between times (say) t and t' if the algorithm does not take any steps between t and t'. Furthermore, note that $[t, t']$ could encompass an unbounded duration of time. Therefore, failure detectors $\mathcal{D}$ whose specifications require some real-time sensitive change in their output (like, for instance, instantaneous suspicion of a crashed process) cannot be implemented even in systems augmented by $\mathcal{D}$.

The issue of failure detectors' irreflexivity and the issue of implementability of failure detectors is addressed in [53]. The system model framework in [53] by Jayanti et al. (henceforth called the *Jayanti framework*) augments the traditional failure-detector framework in [20] with input and output queues for algorithms to interact with the 'external world'. Thus, the definition of an execution is augmented and depends on (1) the times at which processes crash, (2) the outputs of the failure detector, (3) the contents of the input queue at each process, and (4) the algorithm being executed. The failure detectors are made reflexive by allowing the duration between a query and its response to be non-zero. Thus, the framework disallows failure detectors $\mathcal{D}$ whose specifications require some real-time sensitive change in their output (like, for instance, instantaneous suspicion of a crashed process). The issue of implementability of failure detectors is addressed by essentially multiplexing the actions of the failure detector implementation and the applications querying the

failure detector. Every query to the failure detector is treated as an addition to the input queue of failure detector.[1] A similar multiplexing is claimed to permit applications and the failure detector to share the same communication channels.

Unfortunately, the Jayanti framework is not sufficiently detailed to eliminate ambiguity and undesirable side effects of certain permissible, but undesirable, behavior of algorithms within the framework. Additionally, the framework can be too restrictive to permit unrestrained asynchrony among applications querying the failure detector. We explain these issues next. The Jayanti framework states that the failure detector algorithm and other application algorithms are multiplexed 'fairly', but no formal definition of such fair multiplexing is mentioned. A round-robin scheduling of steps of each application (including the failure detector) is provided as an illustrative example. Therefore, despite a partially synchronous system model's guarantees on computation, an 'insufficiently fair' scheduler that multiplexes actions of various algorithms could deny the failure detector algorithm the necessary guarantees to behave correctly. On the other hand, if round-robin scheduling is accepted as the definition of a 'fair' schedule, then the system behavior, when restricted to applications, is no longer completely asynchronous. Round-robin scheduling ensures all applications execute steps at approximately the same rate. If the underlying system is partially synchronous, then the execution of each application is partially synchronous as well. Therefore, the Jayanti framework could potentially be too restrictive to model asynchronous systems augmented with failure detectors.

The multiplexing of messages from various applications also introduces similar issues in the Jayanti framework. If the failure detector application sends messages less

[1]Recall that the input and output queues are used by the failure-detector algorithm to interact with the 'external world', and, in this case, the 'external world' are the applications querying the failure detector.

frequently than other applications, then other applications could potentially flood the channels with their messages, and thus increase the delay experienced by the failure detector messages. Such inflation in message delay beyond the guarantees provided by the underlying system model could compromise the correctness of the failure detectors. This issue becomes more acute if we assume that the underlying system model permits message loss. Traditional system model assumptions for failure-detector based algorithm includes reliable communication. There are several mechanisms available to implement reliable communication on top of unreliable channels [1–3, 11, 37]. However, all such mechanisms involve repeated message retransmissions that potentially increase the end-to-end delay experienced by each message. While this is not problematic for the applications themselves, the correctness of failure detectors depends crucially on message delay. Therefore, such multiplexing of messages from multiple applications introduces non-trivial issues that have not been addressed in the Jayanti framework.

We overcome the above issues by introducing a new system model framework that is based on Timed I/O Automata and ensure that the specifications guarantee that these issues do not occur within the proposed framework.

V.1.2. Overview of the proposed Timed I/O Automata framework

Timed I/O Automata (TIOA) [54] is a mathematical framework for specifying distributed systems whose behavior is constrained by the duration of real time that elapses between events within the system. In TIOA, every entity within a distributed system is a (possibly infinite) state automaton with input and output actions which interface with other entities within the system. For instance, each process is an automaton, and each communication link is an automaton. The process automaton interfaces with the communication-link automaton with the actions *send* and *receive*

which send and receive messages, respectively. The process automaton sends a message by invoking its output action *send* with triggers the input action *send* of the communication-link automaton. Similarly, the process automaton receives a message when its input action *receive* is triggered by the output action *receive* of the communication-link automaton.

A process automaton contains multiple *protocols*. The algorithms that solve various problems are executed within these protocols. Each process has a *control protocol* whose executions are constrained by the partial synchrony of the underlying system model. All other protocols, called *application protocols*, execute without any timing constraints, and hence are completely asynchronous. Failure detector algorithms, which require partial synchrony to behave correctly, are typically executed within a control protocol, and all other applications which query the failure detector are executed within application protocols. While such a structure addresses the issues associated with managing concurrent threads of execution, one thread for each algorithm, the issue of multiplexing messages from multiple applications still remains, and the latter is addressed next.

The messages sent by all the application protocols are stored locally within the control protocol of a process. The control protocol periodically dequeues one message from each application protocol to a specific recipient, concatenates them together with its own message (typically a failure detector message) to that recipient, and sends out a single composite message on the communication link. When such a composite message is received by the control protocol at the recipient process, the message is split into its constituent messages for each protocol, and these messages are enqueued into their respective protocols' message buffers to be processed by the respective application or failure detector. This mechanism ensures that a failure detector message is never delayed due to application protocol traffic, thus the communication bounds

guaranteed by the underlying system model are preserved for failure detector traffic as well. For applications that require reliable communication, the application protocol may implement reliable communication using one the many algorithms available for implementing reliable communication on top of unreliable channels [1–3, 11, 37] without affecting the communication delay experienced by the failure detector protocol.

The failure detector updates its suspicion information by updating a local variable called suspectList, and this updating is assumed to happen instantaneously. Thus the latest output of the failure detector is available at each time instant. The application protocols can read this variable at any time through the actions *query* and *response* which is assumed to take zero time.

The proposed framework addresses all the issues of existing frameworks for failure-detector based systems within partially synchronous systems. Before providing a detailed description of this framework, we provide a detailed overview of Timed I/O Automata next, and follow it up with the framework description in Section V.3.

V.2. Timed I/O Automata

Recall that timed I/O Automata (TIOA) is a mathematical framework for specifying distributed systems whose behavior is constrained by the duration of real time that elapses between events within the system. For a complete and formal description of the theory of TIOA, see [54]. An informal description follows.

V.2.1. Definitions

A TIOA is a state machine that consists of a set of *variables* V which determine the state of the automaton, a set of discrete *actions* A which change the state of the automaton by changing the value of (some) variables, and a set of *trajectories* $\mathcal{T}$

which model the continuous change in the values of (some) variables over intervals of time.

V.2.1.1. Time

In TIOA, a unidirectional time axis T is modeled by a subgroup of $(\mathbb{R}, +)$ — the real numbers with addition. An interval of time J is a nonempty, convex subset of T.

V.2.1.2. Variables

Variables may be *discrete* or *analog*. The values of discrete variables are typically manipulated by actions, whereas the values of analog variables are typically manipulated by evolution functions (*i.e*, by passage of time). Examples of discrete variables include variables that store information like user input, messages in-transit, and such. Examples of analog variables include variables that store local clock values, global time values, and such. The set of variables is denoted V.

V.2.1.3. States

The state of an automaton is uniquely determined by the values of the variables in the set V. The set of all possible valuations of V (denoted $val(\mathsf{V})$) represents the state space Q of the automaton. The size of this state space may be finite or even infinite. Some subset of Q is denoted Θ, the start states. Intuitively, the state of an automaton upon initialization is a member of Θ.

V.2.1.4. Actions

Actions cause instantaneous changes in the values of a set of variables V (and consequently, change the state of the automaton). A TIOA has a fixed set of actions. An action may change the state of the automaton only when the action is *enabled*. Every

action a has a predicate on the state of the automaton called a *precondition* (denoted $a.prec(s)$ where s is the state of the automaton) which determines if a is enabled or not. If the precondition evaluates to $true$, then the corresponding action (a) is said to be enabled. The effect of executing action a in state s is to change the state of the automaton to a new state s' (which may be the same as s) denoted $a.eff(s)$.

Actions are partitioned into two sets: *external* and *internal*. An automaton interacts with other automata through external actions. For example, a TIOA modeling a processor interacts with the TIOA modeling a communication link through external actions *send* and *receive* which send and receive messages, respectively. An internal action changes the state of the automaton but does not interact with other automata.

External actions are further partitioned into two sets: *input* and *output*. Input actions 'receive' information from the 'outside world', that is, from other automata (for example, the *receive* action at a processor), and output actions 'send' information to the 'outside world' (for example, the *send* action at a processor). While the preconditions associated with internal and output actions may be arbitrary predicates on the state of the automaton, the precondition associated with input actions must be *true*; that is, input actions are always enabled.

V.2.1.5. Trajectories

Trajectories may be viewed as functions that map intervals T of time to $val(V')$ where $V' \subseteq \mathsf{V}$. If V' is the empty set for a trajectory over an interval T, then the trajectory simply denotes the passage of time over the interval T. If the interval T is $[t, t]$ (for $t \in \mathsf{T}$), then the trajectory is called a *point* trajectory for the values V' and it denotes an instantaneous change in the values of the variables in V'. Given a trajectory τ_i, the state of the automaton at the start of τ_i is called the *first valuation* of τ_i and is denoted $\tau_i.fval$. The time associated with $\tau_i.fval$ is denoted $\tau_i.ftime$. If τ_i is closed

and spans a finite interval of time then the state of the automaton at the end of τ_i is called the *last valuation* of τ_i and is denoted $\tau_i.lval$. The time associated with $\tau_i.lval$ is denoted $\tau_i.ltime$. As a corollary, note that if the interval T associated with τ_i is finite, then $T = [\tau_i.ftime, \tau_i.ltime]$. Let $\mathcal{T}$ denote the set of trajectories over the valuation of the set V; that is, let $\mathcal{T}$ denote the set of trajectories over Q — the state space of the automaton. In addition, we assume that for all times $t \in \mathsf{T}$, the point trajectories for all states in Q over all intervals $[t, t]$ are in $\mathcal{T}$.

V.2.1.6. Executions

The behavior of a TIOA is described by *executions*. An execution α of a TIOA is a sequence of alternating trajectories and actions of the form $\alpha = \tau_0 a_1 \tau_1 a_2 \tau_2 \ldots$, with the following properties:

1. $\tau_0.fstate \in \Theta$. That is, $\tau_0.fstate$ is a start state.
2. Each τ_i is a trajectory in $\mathcal{T}$.
3. If $i < j$, then $\tau_i.ltime \leq \tau_j.ftime$.
4. If τ_i is not the last trajectory in α, then:
   (a) $\tau_i.ltime = \tau_{i+1}.ftime$, and
   (b) $a_{i+1}.pre(\tau_i.lstate) = true$ and $a_{i+1}.eff(\tau_i.lstate) = \tau_{i+1}.fstate$ — that is, action a_{i+1} is enabled and executed at time $\tau_i.ltime$ and the state of the automaton changes instantaneously from $\tau_i.lstate$ to $\tau_{i+1}.fstate$.
5. If τ_i is the last trajectory, then $\tau_i.ltime = \infty$. We consider only infinite executions.

6. Let a be an action specified in the TIOA. If $\forall i \in \mathbb{N} : a_i \neq a$, then $\nexists j \in \mathbb{N}$ such that $\forall k \geq j, \forall t \in \tau_k : a.pre(\tau.fstate) = true$ where $dom(\tau) = [t, t]$ (and, by assumption $\tau \in \mathcal{T}$). That is, every continuously enabled action is eventually executed. There exists no action a such that a is eventually and continuously enabled in some suffix of α and is never executed. This property is also called *weak fairness*.

V.2.2. Operations

Three important operations on TIOA are composition, hiding, and restriction. Composition allows us to combine multiple TIOA into a single TIOA, and hiding allows us to replace certain input and output actions of the constituent TIOA with an internal action in the composite TIOA. Restriction allows us to project an execution of a TIOA on a subset of actions and variables. Restriction can be used in conjunction with the composition operation to analyze the behavior of a single TIOA in isolation while it is composed with other TIOA. We describe these three operations in this section.

V.2.2.1. Composition

As mentioned earlier, composition allows a complex automaton to be constructed by multiple simpler automata. For example, the entire distributed system can be represented as a single TIOA by composing the process TIOA for each process and the system model TIOA consisting of a scheduler and the communication links. The composition operation among a set of automata $\mathcal{A}_{set} = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n\}$ identifies each action a_i such that a_i is an unique output action of some automaton $\mathcal{A}_k \in \mathcal{A}_{set}$, and a_i is an output action of the automata in $\mathcal{A}_{a_i} \subset \mathcal{A}_{set}$ (where $\mathcal{A}_{a_i} \neq \emptyset$) . When $\mathcal{A}_k$ executes its output action a_i, the automata in $\mathcal{A}_{a_i}$ execute their input action a_i.

V.2.2.2. Hiding

As mentioned earlier, hiding operations allow us to reclassify external actions of constituent automata as an internal action of the composed automaton. Given two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ that are composed to form an automaton $\mathcal{A}$, the automaton $\mathcal{A}$ is equivalent to an automaton $\mathcal{B}$ where $\mathcal{B}$ is identical to $\mathcal{A}$ except that: every pair of actions (a_i, a_i), one from $\mathcal{A}_1$ and another from $\mathcal{A}_2$, that are 'composed' by the composition operator are replaced by a single internal action a' with the same precondition as the output action a_i (note that the precondition of the input action a_i is simply *true*).

V.2.2.3. Restriction

Restriction is a projection operator with respect to set of actions A' and a set of variables V'. Given an execution α of a TIOA, let A' be a subset of the actions of the TIOA and V' be a subset of the variables in the TIOA. An (A', V')-*restricted* execution of α is obtained by first projecting all the trajectories of α on the variables in V', then removing all the actions not in A', and finally concatenating all the adjacent trajectories.

V.2.3. Special kind of TIOA

We define a special kind of TIOA for the automata that are only nominally timed. That is, such automata are not constrained by passage of time, but only constrained by the ordering of actions within the automaton. Such automata are said to be *timing-independent* and are defined as follows: A timed automaton is said to be timing-independent if and only if all of its state variables are discrete and its set of trajectories is exactly the set of constant-valued functions over $[0, t)$ for all $t \in \mathbb{R} \cup \{\infty\}$. That is,

the state of the system does not change simply through passage of time, it changes only when an action is executed.

With timing-independent automata, since all trajectories are constant-value functions, for every finite trajectory τ, we have $\tau.fstate = \tau.lstate$. Therefore, all executions of a timing-independent TIOA can be replaced with *runs* consisting of alternating sequences of states and actions such that for every execution $\alpha = \tau_0 a_1 \tau_1 a_2 \tau_2 \ldots$ there exists a run $\alpha_{run} = s_0 a_1 s_1 a_2 s_2 \ldots$ where $s_i = \tau_i.fstate$.

V.3. Constituents of a distributed system

This section defines the constituting elements of a distributed system and their modeling as TIOA. A distributed system has multiple constituent components: *step scheduler*, *communication schedulers*, and *processes*. Informally, step schedulers dictate when processes can execute certain critical actions (called *program actions*), and communication schedulers dictate when processes can send messages to other processes and receive messages from other processes within the system. A detailed specification of the aforementioned terms follows.

V.3.1. Step scheduler and communication scheduler

The step scheduler and communications scheduler are modeled as a single TIOA [54, 60] M (called, simply as, *scheduler*). However, for ease of understanding, we describe the step scheduler separately and independently from communication scheduler.[2]

[2]Note that we could alternatively specify step scheduler and the communication scheduler as separate TIOA and then compose them to form M, but this is laborious and ultimately wasteful because in the entire dissertation we treat a system model as a single entity.

V.3.1.1. Step scheduler

The step scheduler triggers a process to perform a specific operation through an output action *takeStep*. The step scheduler has a different *takeStep* action for each process. The behavior of a process when the scheduler invokes the output action *takeStep* is discussed later in Section V.3.2. The step scheduler may impose constraints on the relative ordering of invocations of *takeStep* for various processes in the system. For instance, the step scheduler could constrain the sequence of invocations of *takeStep* for various processes such that between two consecutive executions of *takeStep* for each process i, the step scheduler is guaranteed to invoke *takeStep* for each process in the system no more than (say) Φ times. Such a step scheduler is said to impose an upper bound Φ on relative process speeds. The step scheduler is guaranteed to invoke *takeStep* for each process infinitely often, except if the process is *crashed*; crashed processes are discussed in Section V.3.3.

V.3.1.2. Communication scheduler

Processes communicate with each other through communication links. The communication scheduler takes receipt of the messages to be sent to other processes, transports them through communication links, and determines when messages may be delivered to the recipient processes. Each pair of processes (i, j) are assumed to be connected by two unidirectional communication links that send and receive messages between i and j in both directions. Process i sends a message to process j by invoking its output action *send(i,j)*. The output action *send(i,j)* is also the input action *send(i,j)* for the system model's communication scheduler. The communication scheduler transports that message from i to j through the respective communication link between i and j. The message is delivered to j through the communication scheduler's output

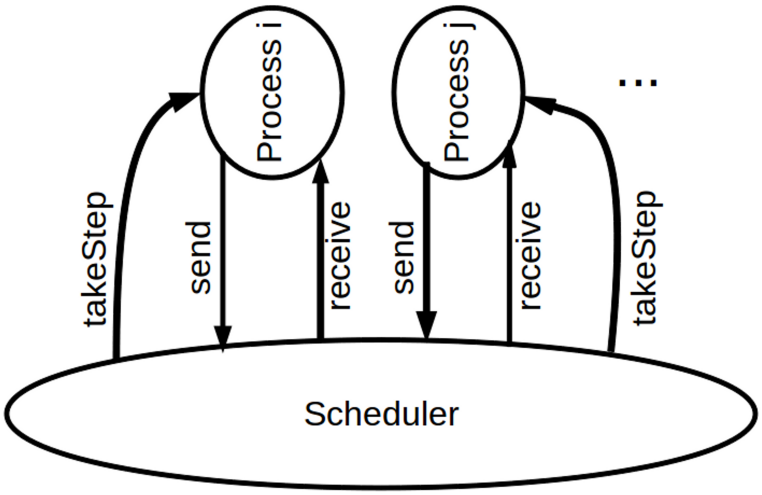

Fig. 2 Schematic of the Timed I/O Automata framework for a message-passing distributed system.

action *receive*. The scheduler M's output action *receive(j,i)* is also j's input action *receive(j,i)*, and the message is delivered to j.

The communication scheduler may satisfy some constraints on reliability and temporal guarantees for communication. For instance, the scheduler may guarantee that every message sent on a communication link is delivered, but there may be an arbitrary time between a send and its corresponding receive action. Alternatively, the scheduler may guarantee that some subset of the messages sent on a communication link are received within some bounded delay. That is, for some subset of messages, there exists an upper bound on the duration between the send and the corresponding receive action while other messages may be arbitrarily delayed or dropped.

A schematic illustrating the interaction between the TIOA components of a message-passing distributed system is shown in Fig. 2.

V.3.2. Processes

The system has a fixed set $\Pi = \{p_1, p_2, \ldots, p_n\}$ of n processes. A *process* within a system is modeled by a TIOA [54, 60]. Typically, and in this dissertation, processes perform multiple tasks concurrently. These tasks may include crash fault detection(cf. [20]), consensus(cf. [43]), reliable communication (cf. [3]), mutual exclusion [30], and such. In order to model each of these tasks separately within a single process, we assume that each process executes a fixed set $\rho = \{r_0, r_1, r_2, \ldots, r_m\}$ of one or more *protocols* concurrently. Each protocol corresponds to a single task of the kind mentioned earlier and is modeled as a TIOA as well. Consequently, a process TIOA may be viewed as the composition of its constituent protocol TIOA.

Although there are many protocols within a process, only one protocol can interact with communication links (via the scheduler)[3]. Consequently, among the many protocols within a process, we designate one protocol to be the *control* protocol, and it is denoted r_0. All other protocols are called *application protocols*. The control protocol acts as a gateway between application protocols and the scheduler $\mathcal{M}$. While application protocols' actions are unconstrained by the scheduler, the control protocol may execute certain actions (including sending messages on the communication links) only when permitted by the scheduler; that is, the control protocol performs certain actions (these actions are the takeStep actions and will be discussed later) only in response to the step scheduler's *takeStep* action. Also, all the messages sent and received by application protocols are handled by the control protocol which, in turn, interfaces with the communication links through the *send* and *receive* actions.

The structure of a protocol TIOA is as follows. Each protocol has two local

[3]In empirical systems, this task is performed by the network layer protocol.

message buffer queues called *send buffer* and *receive buffer*.[4] Any message that the protocol wants to send to another process is placed in the *send buffer*, and all the messages received from another process are retrieved from the *receive buffer*. A protocol also has at least two external actions: *send* and *receive*. The *send* action is an output action that sends a message from the local *send buffer* either to the control protocol (at that process) or to the communication link to be delivered to its counterpart protocol automaton at another process. Similarly, the *receive* action is an input action that receives a message from either the control protocol (at that process) or the communication link and places the message in its local *receive buffer*.

Since the control protocol acts as a gateway between application protocols and the communication scheduler, we label the *send* and *receive* actions for the application protocol differently from the *send* and *receive* actions for the control protocol. In the case of application protocols, the *send* output action is labeled rSend and the *receive* input action is labeled rReceive. Again, since the control protocol acts as a gateway between application protocols and the communication scheduler, the action rSend becomes an input action, and the action rReceive becomes an output action, to the control protocol. In the case of the control protocol, the *send* output action is labeled send and the *receive* input action is labeled receive. The send action at the control protocol becomes an input action for the communication scheduler, and the receive action at the control protocol becomes an output action for the communication scheduler.

Apart from the *send* and *receive* actions, each protocol also has an action labeled

[4]Note that these message buffers are a part of the protocol's local state. They may be viewed as local variables that store messages. Also, the names of these variables differ between application protocols and control protocols. The names of these variables are explained in Sections V.3.2.1 and V.3.2.2 for control protocols and application protocols, respectively.

takeStep. The action takeStep is an internal action in application protocols. In the control protocol for, say, a process i, the action takeStep is an input action which is executed with the output action takeStep by the step scheduler for process i. Note that takeStep actions are executed only while the hosting process is not crashed (we discuss process crashes in detail in Section V.3.3). In the case of application protocols, every application protocol has an input action crash which is invoked when the hosting process crashes. Upon being invoked, crash disables the program action takeStep permanently. In the case of control protocols, since takeStep is an input action, takeStep cannot be disabled. Consequently, in order to model the control protocol at a crashed process not executing its program action, the step scheduler stops invoking the action *takeStep* to cease a crashed process from taking control-protocol program actions.

Recall that the step scheduler and communication scheduler are modeled as a single TIOA denoted (simply as) *scheduler*. For the rest of this dissertation, we will not longer refer to step scheduler or communication scheduler in isolation. Instead, we simply refer to them together as the scheduler.

The TIOA schematic for the constituent protocols within a process is shown in Fig. 3. A detailed description of the structure of control protocols, application protocols, and program actions follows.

V.3.2.1. Control protocols

Recall that a control protocol acts as a gateway between application protocols, and the system scheduler and communication links. Hence, it contains the minimal set of variables and actions shown in Algorithms 1–2, and these variables and actions are described in this section.

The control protocol maintains two arrays of message buffers: rSendBuff and

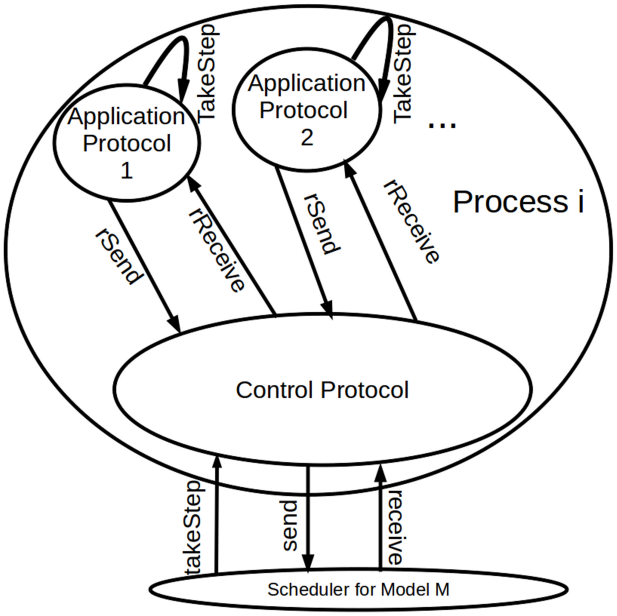

Fig. 3 Schematic of the TIOA for a process consisting of multiple protocols (ignoring process crashes).

rReceiveBuff. All message buffers are modeled as unbounded-size FIFO queues. The message buffer rSendBuff[r,j] in the control protocol of process i contains the messages that protocol r (at process i) has sent to (protocol r at) process j. Similarly, the message buffer rReceiveBuff[r,j] in the control protocol of process i contains the messages for protocol r (at process i) sent from (protocol r at) process j, and yet to be delivered to protocol r. These message buffers are manipulated by the input actions rSend and receive, and the output actions rReceive and send.

Note that we make no assumptions about the contents of messages sent and received by the control protocol. This is denoted in Algorithm 1 by declaring each instance of message msg to be of arbitrary Type.

The action rSend(msg,r,j) enqueues the message msg from protocol r to process j in the message buffer rSendBuff[r,j]. The action send(msg,i,j) at process i (1) dequeues one message from each message buffer rSendBuff[r,j] corresponding to protocol r's messages

to process j, (2) appends these messages together (with markers and identifiers so that the individual messages may be retrieved by the recipient) to construct msg and sends msg to j (through the communication links). Such a message is received by the control protocol at j through the action receive(msg,j,i) which, in turn, retrieved the individual messages within msg and for each individual message m' sent by protocol r, m' is added to rReceiveBuff[r,i] message buffer. Every protocol r at j take receipt of a message m' through the action rReceive(m',r,i) in which protocol r receives a message m' from i by dequeuing m' from rReceiveBuff[r,i].

We also define aliases sendBuffer[j] and receiveBuffer[j] for rSendBuff[r_0, j] and rReceiveBuff[r_0, j] for reasons described later in Section V.3.2.2.

Apart from the above actions, the control protocol also has the input action takeStep which enables the control protocol to take a program action. After executing a program action, the control protocol immediately invokes the action send for all processes j to ensure that messages are promptly sent over the communication links.

When a process crashes, the scheduler stops executing the action takeStep for the crashed process. Consequently, upon crashing, the control protocol stops executing its program action and ceases to send messages over the communication links.

While the above description outlines the minimal variables and actions of the control protocol, it is permissible for control protocols to have additional actions and variables that enable the control protocol to interact with the application protocols. In fact, this dissertation augments control protocols with specific actions and variables to enable such interaction.

V.3.2.2. Application protocols

As mentioned earlier, an application protocol consists of an input action receive, an output action send, and an internal action takeStep (which executes the program

Algorithm 1 Signature for minimal states and actions of a control protocol automaton. For descriptions of the automaton, see Sections V.3.2.1 and V.3.2.3.

automaton controlProtocol(i: pIndex) *Control Protocol for process i*
type pIndex = **enumeration of** p_1,p_2,...,p_n **where** $\Pi = \{p_1, p_2, \ldots, p_n\}$
type rIndex = **enumeration of** r_0, r_1, r_2,...,r_m **where** $\rho = \{r_0, r_1, r_2, \ldots, r_m\}$
and r_0 refers to the control protocol itself

signature
input rSend(msg: Type, r: rIndex\{r_0}, j: pIndex)
//Procotol r sends message msg to process j
output send(msg: Type, i: pIndex, j: rIndex)
//Sends message msg from process i to process j
output rReceive(msg: Type, r: rIndex\{r_0}, j: pIndex)
//Delivers message msg to protocol r from process j
input receive(msg: Type, i: pIndex, j: pIndex)
//Process i receives message msg from process j
input takeStep(i: pIndex) *//Scheduler enables process i to take a step*

states
now: Real ← 0 *//Current time*
rSendBuff: Array[rIndex, pIndex, Queue[Type]] ← constant(∅)
//Messages sent by protocols
rReceiveBuff: Array[rIndex, pIndex, Queue[Type]] ← constant(∅)
//Messages received for protocols
pause: Array[pIndex, Boolean] ← constant(false) *//Device to stop time evolution*

aliases
sendBuffer[j] ≡ rSendBuff[r_0,j]
receiveBuffer [j] ≡ rReceiveBuff[r_0,j]

Algorithm 2 Minimal state transitions for the control protocol automaton. For descriptions of the automaton, see Sections V.3.2.1 and V.3.2.3.

```
transitions for automaton controlProtocol(i: pIndex)
 input rSend(msg, r, j)
  effect
   enqueue msg in rSendBuff[r,j]

 output send(msg,i,j)
  precondition
   (pause[j] = true) ∧ (msg = join(∀r∈ ρ: ⟨(front rSendBuff[r,j]) else ⊥,r⟩))
                          //Dequeue one message from each protocol send buffer (or ⊥, if
                         //the buffer is empty) and join them together to form message m
  effect
   ∀r∈ ρ: dequeue from rSendBuff[r,j]
   pause[j] ← false

 output rReceive(msg, r, j)
  precondition
   msg = ((front rReceiveBuff[r,j]) else ⊥)
  effect
   dequeue msg from rReceiveBuff[r,j]

 input receive(msg, i, j)
  effect
   ∀r∈ ρ: enqueue m′ in rReceiveBuff[r,j] where ⟨m′,r⟩ is in msg

 input takeStep(i)
  effect
   execute an enabled program action
   ∀j∈ Π::pause[j] ← true

trajectories
 stop when
  ∃j∈ Π::pause[j] = true  //Pause time to send msgs immediately after a program action
 evolve
  d(now) = 1                    //Time evolves at the rate of 1 time unit per real-time unit
```

actions). In this section, we describe the structure of an application protocol in detail. An application protocol TIOA contains the variables and actions shown in Algorithm 3. The application protocol maintains two message buffers: sBuffer and rBuffer. The message buffer rBuffer[j] in an application protocol r of process i contains the messages that protocol r (at process i) has sent to (protocol r at) process j. Similarly, the message buffer rBuffer[j] in an application protocol r of process i contains the messages for protocol r (at process i) sent from (protocol r at) process j, and delivered to protocol r. These message buffers are manipulated by the actions rSend and rReceive which interface with the control protocol. The output action rSend(m,r,j) dequeues the message m from the message buffer sBuffer[j] and sends it to the control protocol. The input action rReceive(m,r,j) receives a message m sent by (protocol r at) process j delivered by the control protocol. The message buffers sBuffer and rBuffer are also manipulated by the when the latter sends messages to j by adding them to sBuffer[j] and receives messages from j by de-queuing a message from rBuffer[j].

Similar to the control protocol, we make no assumptions about the contents of the messages sent and received by the application protocols. This is denoted in Algorithm 3 by declaring each instance of message msg to be of arbitrary Type.

The input action crash(i,r) is invoked when process i crashes (process crashes are discussed in Section V.3.3). Upon being invoked, crash(i,r) sets the variable live to $false$ which disables the program at the protocol, thus effectively ceasing operations at the protocol.

Like control protocols, it is permissible for application protocols to have additional actions and variables that enable the application protocol to interact with the control protocol and other application protocols. Again, this dissertation augments application protocols with specific actions and variables to enable such interaction.

Algorithm 3 Application protocol automaton framework. For description of the automaton, see Sections V.3.2.2 and V.3.2.3.

```
automaton applicationProtocol(r: rIndex, i: pIndex)        //App. Protocol r at process i
type pIndex = enumeration of p1,p2,...,pn where Π = {p1, p2, ..., pn}
type rIndex = enumeration of r0, r1, r2,...,rm where ρ = {r0, r1, r2, ..., rm}
                                          and r0 refers to the control protocol itself
 signature
  output rSend(msg: Type, r: rIndex, j: pIndex)        //sends message msg to process j
  input rReceive(msg: Type, r: rIndex, j: pIndex)      //Delivers msg sent by process j
  internal takeStep()                                  //Executes a program action
  input crash(i: pIndex, r: rIndex)                    //Disables program actions
 states
  now: Real ← 0                                        //Current time
  sBuffer: Array[pIndex, Queue[Type]] ← constant(∅)    //Messages to be sent
  rBuffer: Array[pIndex, Queue[Type]] ← constant(∅)    //Messages received
  live: Boolean ← true                                 //Set to false upon crashing
 aliases
  sendMsg(m,j) ≡ enqueue m in sBuffer[j]
  receiveMsg(m,j) ≡ dequeue m from rBuffer[j]
 transitions
  output rSend(m, r, j)
   precondition
    m = ((front sBuffer[j]) else ⊥)
   effect
    dequeue m from sBuffer[j]
  input rReceive(m, r, j)
   effect
    enqueue m in rBuffer[j]
  input crash(i,r)
   effect
    live ← false
  internal takeStep()
   precondition
    live = true
   effect
    execute an enabled program action
 trajectories
  evolve
   d(now) = 1
```

V.3.2.3. Program actions

Recall that every protocol automaton has a special action takeStep. When takeStep is invoked, the protocol executes the state transition specified by its *program action.*

A program action is a sequence of guarded commands [32]; a guarded command is a pair $\langle guard \rangle \rightarrow \langle command \rangle$ consisting of a predicate on the local state (the guard) followed by a finite sequence of executable statements (the command). A guarded command is enabled when its guard is true. Upon being selected for execution, an enabled command is executed atomically (without interleaving any other operations). Since several guarded commands may be enabled simultaneously, the order of execution is non-deterministic. However, an enabled guarded command is selected for execution subject to *weak fairness* which states that a continuously enabled guarded command must be eventually selected for execution.

Each guarded command retrieves at most one message sent from each process j from its local *receive buffer* for j, manipulates other local variables, and sends at most one message to each process j in the system by adding them to its local *send buffer* for j.

While all other actions within a protocol perform the functions of coordinating the interaction with the scheduler, communication links, and other processes and protocols in the system, the program action is responsible for performing the actual sequence of operations necessary for the distributed system to solve a given problem. In the vernacular, program actions contain the code necessary for the distributed system to accomplish its 'actual purpose', or to 'get the job done'.

Although the behavior of program actions in control protocols and application protocols is semantically identical, they are syntactically different. The local *send buffer* for j at the program action in a control protocol is rSendBuff[r_0,j] whereas the

local *send buffer* for j at the program action in an application protocol is sBuffer[j]. Similarly, the local *receive buffer* for j at the program action in a control protocol is rReceiveBuff[r_0,j] whereas the local*send buffer* for j at the program action in an application protocol is rBuffer[j]. In order to establish a syntactic similarity for program actions at all protocols, we introduce the following aliases: sendBuffer and receiveBuffer.

In control protocols, sendBuffer[j] is an alias for rSendBuff[r_0,j], whereas in application protocols, sendBuffer[j] is an alias for sBuffer[j]. Similarly, receiveBuffer[j] aliases to rReceiveBuff[r_0,j] in control protocols and to rBuffer[j] in application protocols.

Based on the description of control protocol, application protocols, and program actions, the interaction among the three entities is illustrated in Fig. 4.

V.3.3. Fault environment

In this dissertation, we assume that processes can fail only by crashing. A *crash fault* occurs when a process ceases execution without warning and never recovers [26]. We represent crash faults as a TIOA *faultPattern* that, in order to crash process i, invokes an output action *crash(i)* to the model M and to process i (and its constituent protocols). When *crash* is invoked at M, the scheduler stops invoking the action *takeStep*, thus permanently ceasing all program actions at the control protocol at i, and the communication links could potentially delay or drop messages to and/or from i. When *crash* is invoked at process i, all the program actions at all application protocols in i are disabled. Ideally, a crash fault at process i is expected to cease all actions (input, output, and internal actions) at process i completely. Note that while ceasing to invoke the action *takeStep* halts the program actions of the control protocol, it does not halt the input and internal actions of the control protocol, and while the action *crash* at process i halts the program actions of all application protocols, it

Algorithm 4 Fault pattern automaton.

automaton faultPattern(f: Integer) *//No more than f processes crash*
type pIndex = **enumeration of** p_1,p_2,...,p_n **where** $\Pi = \{p_1, p_2, \ldots, p_n\}$
type rIndex = **enumeration of** r_0, r_1, r_2,...,r_m **where** $\rho = \{r_0, r_1, r_2, \ldots, r_m\}$

signature
output crash(i: pIndex[, r: rIndex]) *//Crashes a process at the appropriate time*

states
now: Real $\leftarrow$ 0 *//Current time*
crashNum: Integer $\in$ [0, f] *//Non-deterministic choice of the total number of crashes*
crashList: Set[pIndex] $\subset \Pi \wedge$ |crashList| = crashNum *//Set of faulty processes*
crashTimes: Array[pIndex, Real] $\leftarrow$ non−determinstic times *//The fault pattern*
crashStatus: Array[pIndex, Boolean] $\leftarrow$ constant(false)
//If true, crash events have been sent

transitions
output crash(i[, r]) *//Crash process i and inform the scheduler*
precondition
(crashTimes[i] = now)$\wedge$ (i$\in$crashList) $\wedge$ (crashStatus[i] = false)
effect
crashStatus[i] $\leftarrow$ true *//Send crash event*

trajectories
stop when
$\exists$i$\in \Pi$:(crashTimes[i] = now) $\wedge$ ($\neg$crashStatus[i])) *//Process i to be crashed*
evolve
d(now) = 1

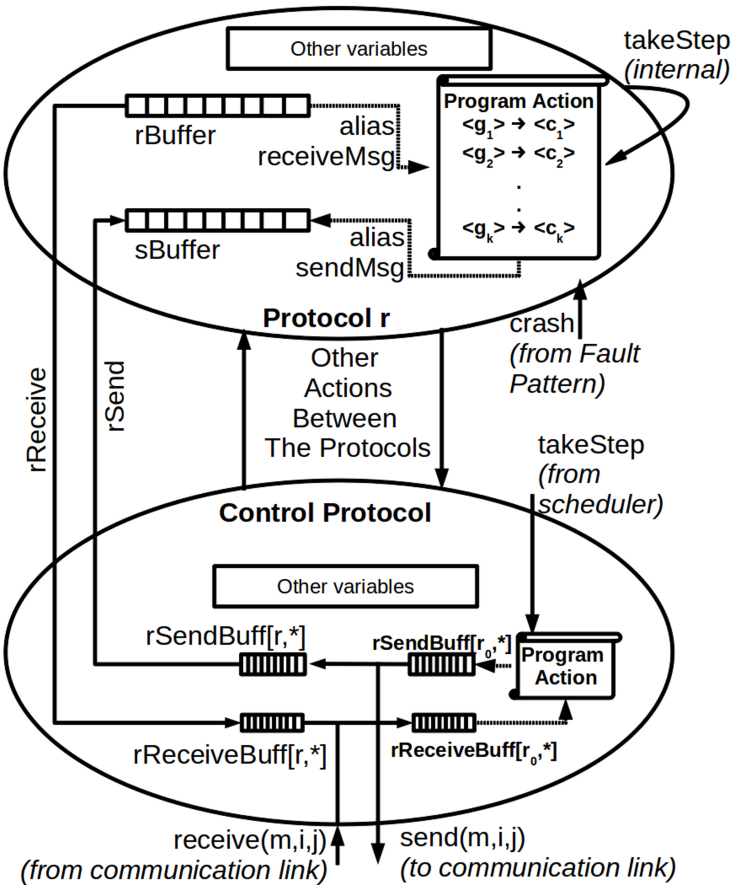

Fig. 4 Interaction between the control protocol automaton and an application protocol automaton at a process i. See Sections V.3.2.1, V.3.2.2, V.3.2.3,and V.3.3 for detailed descriptions.

does not halt the input and other internal actions of the application protocols. This leaves open the issue of whether later inputs to process i are ignored, or cause the same changes to the state of process i that they would if i has not crashed (that is, there were additional invocations of the action *takeStep* at i), or cause some other state changes. These distinctions become irrelevant as long as the effects of these state changes are never communicated to any other processes[5]. However, in order to

[5]A similar modeling approach is advocated in [60, p. 251].

simplify analysis, we simply assume that after a process i has crashed, no actions are executed at i.

A TIOA that specifies the aforedescribed fault environment is given in Algorithm 4. The automaton faultPattern generates a fault environment in which at most f processes (among n) crash.

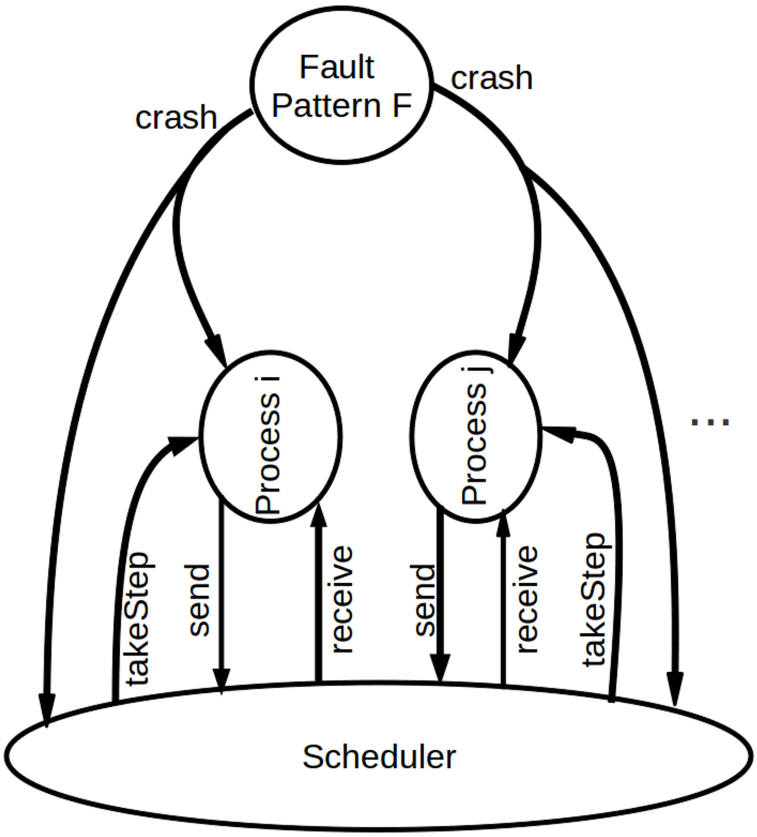

Fig. 5 TIOA schematic for a message-passing distributed system subject to process crashes.

The TIOA schematic of a distributed system model is shown in Fig. 5.

V.4. Executions of a distributed system

Recall that an execution of a TIOA is a sequence of trajectories and states of the form $\alpha = \tau_0 a_1 \tau_1 a_2 \ldots$ with additional restrictions described in Section V.2.1.6. The additional restrictions imposed by the semantics of the scheduler, communication links, processes, and the fault environment (as specified in Sections V.3.1, V.3.2, and V.3.3) are summarized below:

- For each $i \geq 1$, if a_i is an action at process x, then for all $j \leq i$ the fault environment automaton must not have executed $\mathsf{crash}(x)$ in action a_j.

- For each $i \geq 1$, if action a_i is a program action executed by protocol r at process y, and a_i sends a message m to process z by en-queuing m in sendBuffer[z], then:

  - There exists $j > i$ such that action a_j is the output action $\mathsf{rSend}(m', r, z')$ executed by r at y where $m' = \mathsf{m}$ and $z' = \mathsf{z}$.
  - There exists $k > j$ such that action a_k is the output action $\mathsf{send}(m'', y, z'')$ executed by the control protocol at y where m'' contains $\langle \mathsf{m}, r \rangle$ and $z'' = \mathsf{z}$.

  That is, the control protocol does not lose any messages while sending them on behalf of itself or the application protocols.

- For each $i \geq 1$, if action a_i at is receive(m,y,z) executed by the communication link automaton, then:

  - There exists $j > i$ such that action a_j is the output action $\mathsf{rReceive}(m', r, z')$ executed by the control process at y where m contains $\langle m', r \rangle$ and $z' = \mathsf{z}$.
  - There exists $k > i$ such that action a_k is a program action executed by protocol r at y, and a_k de-queues m'' from receiveBuffer[z''] where $m'' = m'$ and $z'' = \mathsf{z}$.

That is, the control protocol does not lose any messages while receiving them on behalf of itself or the application protocols.

Thus an execution of the TIOA formed by the composition of the scheduler, processes (which, in turn, is formed by the composition of a control protocol and potentially multiple application protocols), and fault pattern TIOAs is said to be an execution of the *system*.

V.4.1. Protocol-restricted executions and runs

Recall from Section V.3.2.3 that it is the execution of the commands within program actions that ensures that distributed systems can solve certain problems. Consequently, while analyzing the correctness of a distributed algorithm (or protocol), we focus on the sequence of program actions executed in the system. Specifically, we focus on the program actions of a specific protocol that is expected to satisfy the specifications of a given problem. In order to facilitate the analysis of such problems and algorithms, we introduce protocol-restricted variants of executions and runs.

Let α be an execution of a distributed system. Let $a_{i.r}$ denote the program action of protocol r at process i. Let $V_{i.r}$ denote the set of variables accessed by the program action of protocol r at process i. We define $A'_r = \{a_{j.r} | \forall j \in \Pi\}$, and $V'_r = \cup_{\forall j \in \Pi} V_{j.r}$. The (A'_r, V'_r)-restriction of α is called the *r-restricted execution* $\alpha|_r$ of α.

If an r-restricted execution $\alpha|_r$ is timing independent, then the corresponding run is said to be an r-restricted run.

Such r-restricted definitions allow us to focus our analysis on the execution of program actions of a specific protocol, and hence, on the execution of a specific distributed application. When discussing and analyzing specific distributed applications, the prefix "r-restricted" may be dropped from terms in future chapter to favor brevity.

All such shorthand notations will be explicitly stated in the beginning of each chapter.

V.5. Distributed system models

This section revisits the distributed system models from Section III.1 and specifies the models within the TIOA framework. We start with the formal specification of temporal constraints on computation and communication from Section III.1.1 in Sects. V.5.1 and V.5.2. Then we specify fairness-based partially synchronous system models from Section III.1.4.1 within the TIOA framework in Section V.5.4, and finally specify the empirical system model from Section III.1.4.2 within the TIOA framework in Section V.5.5

V.5.1. Revisiting computational constraints

Recall from Section III.1.1.1 that computational constraints dictate the frequency with which processes execute the steps of their algorithms. Within the TIOA framework, it translates to the frequency with which the control protocols at different processes execute their program actions. Since the program actions of control protocols are invoked by the scheduler, the computational constraints in effect dictate the frequency with which schedulers can invoke takeStep for a process i either with respect to real time, or relative to invocations of takeStep for processes other than i. There are many variations of such computational constraints. Below are definitions of the constraints employed in this dissertation:

- **Lower Bound on Absolute Process Speeds.** Process i has a lower bound l on absolute process speed if i executes its control-protocol program action at least once every l time units. That is, a system is said to have a lower bound l on absolute process speed of a process i if and only if the following is true.

In each execution $\alpha = \tau_0 a_1 \tau_1 a_2 \ldots$ of the system, consider the subsequence $a_{c(1)} a_{c(2)} \ldots$ of α that contains all the instances of takeStep(i) invoked by the scheduler. Then $\tau_{c(1)}.ftime \leq l$, and for every pair of actions $(a_{c(i)}, a_{c(i+1)})$, $\tau_{c(i+1)}.ftime - \tau_{c(i)}.ftime \leq l$.

- **Upper Bound on Absolute Process Speeds.** Process i has an upper bound u on absolute process speed if i executes its control-protocol program action at most once every u time units. That is, a system is said to have an upper bound u on absolute process speed of a process i if and only if the following is true. In each execution $\alpha = \tau_0 a_1 \tau_1 a_2 \ldots$ of the system, consider the subsequence $a_{c(1)} a_{c(2)} \ldots \ldots$ of α that contain all the instances of takeStep(i) invoked by the scheduler. Then $\tau_{c(1)}.ftime \geq u$, and for every pair of actions $(a_{c(i)}, a_{c(i+1)})$, $\tau_{c(i+1)}.ftime - \tau_{c(i)}.ftime \geq u$.

- **Bounded Relative Process Speeds.** A system is said have a bound Φ on relative process speeds if every process that is not crashed executes its control-protocol program action at least once in a duration where all other processes (that have not crashed) execute their control-protocol program action $\Phi + 1$ times. That is, a system is said to have a bound Φ on relative process speeds if and only if the following is true. In each execution $\alpha = \tau_0 a_1 \tau_1 a_2 \ldots$ of the system, consider the control-protocol-restricted execution $\alpha|_{r0}$ of α. Note that every action in $\alpha|_{r0}$ is a control-protocol program action, and hence, corresponds to an invocation of takeStep by the scheduler. Consider every segment that ends with a program action by i and contains exactly $\Phi + 1$ instances of takeStep(i) for some process i. Let the last action of the segment be a_ϕ. Every such segment consists of at least one instance of takeStep(j) for every process j that is not crashed until time $\tau_\phi.ftime$.

V.5.2. Revisiting communicational constraints

Recall that communicational constraints dictate the delay experienced by messages while in transit. Within the TIOA framework, it translates to the duration between invocations of action send(m,i,j) and action receive(m,i,j) for given m, i, and j. Formally, communicational constraints may be viewed as properties of a function L that maps instances of the actions of form send(m,i,j) to instances of actions of the form receive(m,i,j) in an execution α of the system. Below are the formal definitions of communicational constraints from Section III.1.1.2:

- **Fair-Lossy Channels.** A channel $\mathcal{L}$ from process i to process j is said to be *fair-lossy* if and only if the following are true in each every execution $\alpha = \tau_0 a_1 \tau_1 a_2 \ldots$ of the system. Consider a (partial) function L that maps actions of the form send(m,i,j) in α to actions of the form receive(m′,j′,i′) in α:
  - If the domain of L is an infinite set, then the range of L is an infinite set as well. That is, if an infinite number of messages are sent, then an infinite number of messages are received.
  - The function L is 1-1 (one-to-one) and may be partial but the inverse function L^{-1} is total. That is, messages cannot be fabricated and every received message is associated with a unique message that was sent.
  - Let x be an action in α of the form send(m,i,j) that is in the domain of L, and let y be an action in α of the form receive (m′,j′,i′) that is in the range of L. Let t_x be the time associated with x and t_y be the time associated with y in α. If $L(x) = y$, then $t_x \leq t_y$, $\mathsf{m} = \mathsf{m}'$, $\mathsf{i} = \mathsf{i}'$, and $\mathsf{j} = \mathsf{j}'$. That is, no message is delivered before it is sent, messages are not misrouted, and messages are not corrupted.

- **Reliable Channels.** Recall that reliable channels do not lose any messages. More precisely, a channel $\mathcal{L}$ from process i to process j is said to be reliable if the function L that maps actions of the form $\mathsf{send(m,i,j)}$ in α to actions of the form $\mathsf{receive(m',j',i')}$ in α satisfies the following properties:
  - L satisfies the properties of a fair-lossy channel
  - L is an onto function. That is, every message sent to j is delivered.[6]
- **Correct-Reliable Channels.** Recall that correct-reliable channels behave as reliable channels only if the sender and the recipient do not crash. More precisely, a channel $\mathcal{L}$ from process i to process j is said to be reliable if the function L that maps actions of the form $\mathsf{send(m,i,j)}$ in α to actions of the form $\mathsf{receive(m',j',i')}$ in α satisfies the following properties:
  - L satisfies the properties of a fair-lossy channel
  - L is a total function over the set of actions of the form $\mathsf{send(m,i,j)}$ for pairs of processes i and j such that actions $crash(i)$ and $crash(j)$ do not occur in α. That is, if the sender and the recipient do not crash, then every message sent to j is delivered.
- **Bounded-Delay Channels.** Recall that bounded-delay channels are reliable channels where message delay never exceeds some bound Δ. More precisely, a channel $\mathcal{L}$ from process i to process j is said to be a bounded-delay channel if the function L that maps actions of the form $\mathsf{send(m,i,j)}$ in α to actions of the form $\mathsf{receive(m',j',i')}$ in α satisfies the following properties:
  - L satisfies the properties of a reliable channel

[6]Note that if j is crashed, then j will not process this message. However, a reliable channel still guarantees the delivery of the message.

– There exists a positive integer Δ such that for every action x in α that contains send(m,i,j), and every action $y = L(x)$ in α that contains receive(m, j,i), the following is true. If t_x be the time associated with x and t_y be the time associated with y in α, then $t_y - t_x \leq \Delta$

V.5.3. Asynchronous system model

Recall that asynchronous system model guarantees that every correct process executes infinitely many steps and communication channels are correct-reliable. Apart from the aforementioned restriction, there are no temporal bounds on absolute process speeds, relative process speeds, or message delays. The TIOA for such an asynchronous system is given in Algorithms 5–6.

Algorithm 5 TIOA for the asynchronous system model (signature and states).

```
automaton asynchronousSystem()
type Packet = tuple of message: Type, sent: Real, deadline: Integer
type Index = enumeration of p1,p2,...,pn
 signature
  output takeStep(i: Index)                          //Enable process i to take a step
  input crash(i: Index)                                        //Process i is crashed
  input send(m: M, sender: Index, receiver: Index)                    //Send message
  output receive(m: M, receiver: Index, sender: Index)             //Receive message
 states
  now: Real ← 0                                                          //Current time
  crashSet: Set[Index] ← ∅                                  //The set of crashed processes
  transit: Array[Index, Index, Set[Packet]] ← ∅                 //Correct-reliable, link
```

V.5.4. Fairness-based partial synchrony

As mentioned earlier, fairness-based partially synchronous systems restrict the process speeds and message delays based on the occurrence of certain actions in executions of the system. We revisit the terms from Section III.1.4.1 and define them with the

Algorithm 6 TIOA for the asynchronous system model (state transitions).

```
transitions for automaton asynchronousSystem()
 output takeStep(i)
  precondition
   (i ∉ crashSet)                                          //Process i is not crashed
  effect
                //Nothing. Just let process i execute its control-protocol program action
 input crash(i : Index)
  effect
   crashSet ← crashSet ∪ {i}

 input send(m, s, r)          //The message sent is to be reliably delivered if r is correct
  effect
   if (s ∈ crashSet ∨ r ∈ crashSet)              //If either sender or recipient is crashed
    non-deterministically choose deadline (time)  //Unbounded message delay or dropped
    if (deadline ≠∞) then transit[s, r] ← transit[s, r] ∪ {[m, now, deadline]}
    else drop the message
   else                                                  //Both sender and recipient are live
    non-deterministically choose deadline (time) such that
     deadline ≠∞                                        //Unbounded, but finite message delay
    transit [s, r] ← transit[s, r] ∪ {[m, now, deadline]}

 output receive(m, r, s)                                          //Receive a message
  precondition
   ∃p ∈ transit[s, r]:: (p.message = m)
  effect
   transit [s, r] ← transit[s, r]\{p}

trajectories
 evolve
  d(now) = 1
```

formal TIOA framework.

V.5.4.1. Proc-fairness

A process x is said to be *k-proc-fair* (where k is a non-negative integer) in an execution α, if and only if the following is true. For all processes y in the system, in every interval of time in which y executes its control-protocol program action exactly $k+1$ times in α, and x is not crashed at any instance in that duration, then x executes its control-protocol program action at least once in the same interval in α. Similarly, a process x is said to be *eventually k-proc-fair* in α, if and only if the following is true. There exists a (potentially unknown) time t_{gst} after which, for all processes y in the system, in all intervals of time (that begin after t_{gst}) in which y executes its control-protocol program action exactly $k+1$ times in α, and x is not crashed at any instance in that duration, x executes its control-protocol program action at least once in the same interval in α; that is, x is *k-proc-fair* in α from time t_{gst} onwards.

V.5.4.2. Com-fairness

A process x is said to be *d-com-fair* (where d is a non-negative integer) in an execution α if and only if the following is true. For each process y in the system, while a message m is in transit from x to y in α, either (1) y executes its control-protocol program action no more than d times, or (2) x is crashed. Similarly, a process x is said to be *eventually d-com-fair* (where d is a non-negative integer) in α if and only if the following is true. There exists a (potentially unknown) time t_{gst} such that, for each process y in the system, while a message m that is sent after time t_{gst} is in transit from x to y, either (1) y executes its control-protocol program action no more than d times, or (2) x is crashed. That is, x is d-com-fair from time t_{gst} onwards.

In the case of both proc-fairness and com-fairness, although the constraint is

on the execution of control-protocol program action at all processes, note that the control-protocol program action is executed in the input action takeStep at each control protocol, and the execution of takeStep at a control protocol is determined by the execution of takeStep by the scheduler. Therefore, the constraints imposed by proc-fairness and com-fairness are, in fact, restrictions imposed upon a scheduler.

Based on the definition of proc-fairness and com-fairness, the fairness-based partially-synchronous system models from Section III.1.4.1 are restated next.

V.5.4.3. Fairness-based partially-synchronous system models

1. *All Fair* ($\mathcal{AF}$) is an asynchronous system subject to the following constraint: in all executions, all processes are both k-proc-fair and d-com-fair, for known k and d.

2. *Some Fair* ($\mathcal{SF}$) model is an asynchronous system subject to the following constraint: in each execution, some correct process x is both k-proc-fair and d-com-fair, for known k and d.

3. *Eventually All Fair* ($\Diamond\mathcal{AF}$) is an asynchronous system subject to the following constraint: in each execution, all processes are both eventually k-proc-fair and eventually d-com-fair.That is, *eventually* the system behaves like $\mathcal{AF}$. Recall that this system model is identical to the $\mathcal{M}_2$ model from the $\mathcal{M}_*$ system models. Since the $\mathcal{M}_*$ models (and specifically the $\mathcal{M}_2$ model) are of specific interest to us in this dissertation, the TIOA specification for $\mathcal{M}_*$ (in fact, the $\mathcal{M}_2$ variant) is given in Algorithms 7–8.

4. *Eventually Some Fair* ($\Diamond\mathcal{SF}$) is an asynchronous system subject to the following constraint: in each execution, some correct process x is both eventually k-proc-fair and eventually d-com-fair, for known k and d. That is, *eventually* the system

behaves like $\mathcal{SF}$.

Algorithm 7 TIOA for the fairness-based $\mathcal{M}_*$ system models (signature and states).

```
automaton M*System(Phi, Delta)
type Packet = tuple of message: M, sent: Real, deadline: Integer
type Index = enumeration of p1,p2,...,pn
 signature
  output takeStep(i: Index)                          //Enable process i to take a step
  input crash(i: Index)                                          //Process i is crashed
  input send(m: M, sender: Index, receiver: Index)                    //Send message
  output receive(m: M, receiver: Index, sender: Index)             //Receive message
  internal reset()          //Reset the count for number of steps by each process
 states
  now: Real ← 0                                                          //Current time
  Phi: Integer                                    //Known bound on relative process speeds
  Delta: Integer                                          //Known bound on message delay
  GST: Real ← random(ℝ)                                 //Unknown global stabilization time
  stepCount: Array[Index, Integer] ← constant(0)     //Steps by each process since reset
  totalCount: Array[Index, Integer] ← constant(0)     //Total no. of steps by a process
  crashSet: Set[Index] ← ∅                                 //The set of crashed processes
  transit: Array[Index, Index, Set[Packet]] ← ∅                    //Communication link
```

V.5.5. Empirical system model

Recall that the empirical system model from Section III.1.4.2 satisfies the following properties:

- There exists an unknown upper bound on relative process speeds. This property has already been formally specified in Section V.5.1.
- All processes have *local real-time clocks* that are not necessarily synchronized, but they can approximately measure intervals of real time with an (unknown) upper bound D on the *drift rate*. This will be described in Section VIII.4 when the local clock will be used to solve the problem of process celeration (see Chapter VIII for details).

Algorithm 8 TIOA for the fairness-based $\mathcal{M}_*$ system models (state transitions).

```
transitions for automaton M*System(Phi, Delta)
 output takeStep(i)
  precondition
   (i ∉ crashSet) ∧                                          //Process i is not crashed, and
   (((stepCount[i] < Phi) ∧        //i has taken fewer than Phi steps since last reset and
    ∀j ∈ Π, ∃p: Packet p ∈ transit[j,i] ∧ (p.deadline = totalCount[i]))
   ∨ (now < GST))                  //no message is past deadline, or it is prior to GST
  effect
   stepCount[i] ← stepCount[i] + 1          //Account for process i taking another step
   totalCount[i] ← totalCount[i] +1                //Update number of steps taken by i
 internal reset()
  precondition
   (∀i ∈ Π−crashSet: stepCount[i] ≠0)                  //If all live processes have taken
  effect                                              //at least one step since last reset()
   ∀i: Index stepCount[i] ← 0                                              //Reset counters
 input crash(i: Index)
  effect
   crashSet ← crashSet ∪ {i}
 input send(m, s, r)          //The message sent is to be reliably delivered if r is correct
  effect
   if (now < GST)                                                          //If before GST
    non-deterministically choose deadline (time) such that
     deadline > totalCount[r]                    //Unbounded message delay or dropped
    if (deadline ≠∞) then transit[s, r] ← transit[s, r] ∪ {[m, now, deadline]}
    else drop the message
   else                                                                    //At or after GST
    choose deadline such that                                  //Delay not exceeding Delta
     (totalCount[r]+Delta ≥ deadline > totalCount[r])
     transit[s, r] ← transit[s, r] ∪ {[m, now, deadline]}
 output receive(m, r, s)                                                 //Receive a message
  precondition
   ∃p ∈ transit[s, r]: ∀p′ ∈ transit[s, r]\{p}: (p.message = m)
  effect
   transit[s, r] ← transit[s, r]\{p}

trajectories
 evolve
  d(now) = 1
```

- Every pair of processes are connected to each other via ADD channels. We provide a formal specification of the ADD channels next.

Algorithm 9 TIOA for the empirical system TIOA (signature and states).

```
automaton empiricalSystem()
type Packet = tuple of message: M, sent: Real, deadline: Real
type MsgClassification = enumeration of privileged, unprivileged
type Index = enumeration of p_1,p_2,...,p_n where Π = {p_1, p_2, ..., p_n}

 signature
  input send(m: M, sender: Index, receiver: Index)
  output receive(m: M, receiver: Index, sender: Index)
  output takeStep(i: Index)                          //Enable process i to take a step
  internal reset()                  //Reset the count for number of steps by each process
  input crash(i: Index)                                          //Process i is crashed

 states
  transit: Array[Index, Index, Set[Packet]] ← ∅          //Timely, but unreliable, link
  now: Real ← 0                                                        //Current time
  Δ: Real ← random(ℝ)                 //Bound on message delay for privileged messages
  R: Integer ← random(ℕ) + 1             //No. of consecutive unprivileged messages
  sparsity: Integer ← R                //No. of msgs since the last privileged message
  MsgClassification: coinflip                                          //A coin
  Phi: Integer                                          //Bound on relative process speeds
  stepCount: Array[Index, Integer] ← constant(0)     //Number of steps by each process
  crashSet: Set[Index] ← ∅                                //The set of crashed processes
```

ADD channel properties can be specified as restrictions on executions of the system as follows. A channel $\mathcal{L}$ from process i to process j is an ADD channel if the following are true in each every execution $\alpha = \tau_0 a_1 \tau_1 a_2 \ldots$ of the system:

- $\mathcal{L}$ is fair lossy.
- If i and j are correct and there are an infinite number of actions of the form send(m,i,j) in α (for fixed i and j, but various values of m), then the set of (infinite) number of actions of the form send(m,i,j) can be partitioned into two

sets *privileged* and *non-privileged*, and there exist $\Delta \in \mathbb{N}^+$ and $R \in \mathbb{N}^+$ such that:

1. The set *privileged* contains an infinite number of actions.
2. For every action a_x in *privileged* (and time t_x associated with a_x) which contains $\mathsf{send}(\mathsf{m}_x,\mathsf{i},\mathsf{j})$, there exists an action a_y (and time t_y associated with a_y) in α such that a_y contains $\mathsf{receive}(\mathsf{m}_y, \mathsf{j}, \mathsf{i})$, $\mathsf{m}_x = \mathsf{m}_y$, and $0 < t_y - t_x \leq \Delta$.
3. Let a_0 be an action in *privileged* and let t_0 be the time associated with a_0: (1) there exists at least one action a_1 in *privileged* such that $t_1 \leq t_0 + R$ where t_1 is the time associated with a_1, and (2) if $t_0 > R$, then there exists at least one action a_{-1} in *privileged* such that $t_{-1} \geq t_0 - R$ where t_{-1} is the time associated with a_{-1}.

A Timed I/O Automaton that specifies the aforedescribed properties of the ADD channel and bounded relative process speeds in empirical systems is given in Algorithms 9–10.

V.6. Application protocols as TIOA

Note the apparent asymmetry in the effect of computational and communicational constraints in the behavior of protocols within a distributed system model. Specifically, computational constraints in a distributed system model affect the program actions only in the control protocol and not the application protocols; in contrast, communicational constraints affect the messages from both the control protocol and the application protocols. However, this asymmetry is only superficial. Consider messages sent by an application protocol r from process i to process j. The commu-

Algorithm 10 TIOA for the empirical system TIOA (state transitions).

```
transitions for automaton empiricalSystem()
 output takeStep(i)
  precondition
   (i ∉ crashSet) ∧ (stepCount[i] < Phi)          //If i has taken fewer then Phi steps
  effect                                          //since the last time reset() was invoked
   stepCount[i] ← stepCount[i] + 1                //Account for process i taking another step
 internal reset()
  precondition
   (∀i ∈ Π/crashSet: stepCount[i] ≠0)             //If all live processes have taken
  effect
   ∀i: Index stepCount[i] ← 0                                          //Reset counters
 input crash(i: Index)
  effect
   crashSet ← crashSet ∪ {i}
 input send(m, s, r)          //The message sent is either timely or delayed or dropped
  effect
   coinflip ← either privileged or unprivileged          //Choose message classification
   if (sparsity ≠ 0) and (coinflip = unprivileged) then          //Msg is unprivileged
    sparsity ← sparsity − 1          //Decrement the number of unprivileged messages
    either                          //Either, the message is in transit with unbounded delay
     choose deadline such that
      (deadline > now) ∧ (∀i,j ∈ Π,∀p ∈ transit[i,j]:: (p.deadline ≠deadline))
     transit [s, r] ← transit[s, r] ∪ {[m, now, deadline]}
    or drop the message                                   //Or, the message is dropped
   else                                                   //Else, the message is privileged
    choose deadline such that
     (deadline∈ (now, now+Δ]) ∧ (∀i,j∈ Π, ∀p∈ transit[i,j]:: (p.deadline ≠deadline)
     transit [s, r] ← transit[s, r] ∪ {[m, now, deadline]}          //Bounded delay
     sparsity ← R                                  //The next R messages may be unprivileged
 output receive(m, r, s)          //Receive a message whose deadline has not expired
  precondition
   ∃p ∈ transit[s, r]: ∀p′ ∈ transit[s, r]/{p}: (p.message = m) ∧ (p′.deadline ≠ now)
  effect
   transit [s, r] ← transit[s, r]/{p}          //A message whose deadline has not expired
trajectories
 stop when
  ∀i,j ∈ Π, ∃p: Packet p ∈ transit[i,j] ∧ (p.deadline = now)
 evolve
  d(now) = 1
```

nicational constraints affect the delay of these messages only while the messages are in transit between the control protocol at i and the control protocol at j. On the other hand, the delay incurred by these messages in the message buffer sBuffer in the application protocol at i, rReceiveBuffer in the control protocol at j, and the message buffer rBuffer in the application protocol at j remain unconstrained. Consequently, the total delay experienced by application-protocol messages remain unconstrained (except for the minimal constraint of *fair-lossy*).

Recall that program actions at all application protocols are subject to the minimal temporal constraint of *weak fairness* (cf. Section V.2.1.6) much like the *weak fairness* computational constraint of the control protocol in the asynchronous system model (cf. Section III.1.3). Therefore, application protocols are viewed as always being asynchronous whereas control protocols are constrained by the temporal bounds of the distributed system model. Consequently, such differentiated behavior of control protocols and application protocols (and their corresponding messages, respectively) within the TIOA framework provides us with a mathematically sound mechanism to model co-existence of a partially-synchronous control protocol and asynchronous application protocols within a single distributed system.

V.7. Algorithms and solvability

A problem P is defined by a set of properties (predicates) on the sequence of input and output actions that must be satisfied in all executions of a distributed system. A problem P is said to be solvable in system model $\mathcal{M}$ if there exists a set of guarded commands $\mathcal{G}$ such that if $\mathcal{G}$ is executed as the program actions of a protocol r (either a control protocol or an application protocol) at each process, then all executions of $\mathcal{M}$ satisfy the properties of P. Such a set of guarded commands $\mathcal{G}$ is said to be an

algorithm that *solves* P in $\mathcal{M}$, and is denoted $P \preceq \mathcal{M}$.

If the system model $\mathcal{M}$ is partially synchronous and the protocol r is a control protocol, then then $\mathcal{G}$ is said to be a *partially synchronous algorithm.* On the other hand, if the protocol r is an application protocol, then $\mathcal{G}$ is said to be an *asynchronous algorithm* (because application protocols are viewed as being asynchronous).

CHAPTER VI

FAILURE DETECTORS

The study of error is not only in the highest degree prophylactic, but it serves as a stimulating introduction to the study of truth.

Public Opinion, 1922
– *Walter Lippman*

In Chapter V, we explored partially-synchronous models as a plausible framework for expressing many empirical systems insofar as these models provide an adequate description of the empirical systems while retaining sufficient temporal guarantees to admit crash fault tolerant solutions to classic problems in distributed computing. We focused on two classes of partially synchronous models in Chapter V, the $\mathcal{M}_*$ models which represent idealized descriptions of partial synchrony, and the *empirical system model* (ADD channels with bounded relative process speeds and local clocks) which represent physical (and many overlay) systems found 'in the wild'.

Recall that our goal is to implement $\mathcal{M}_*$ on top of empirical systems. We accomplish our goal through algorithmic constructions that preserve sufficient synchronism to enable implementing $\mathcal{M}_*$ on top of an otherwise asynchronous system. To this end, we need a mechanism that can 'capture' the necessary and sufficient synchronism from the empirical system model. Failure detectors [20] are such a mechanism. This chapter provides a detailed overview of failure detectors and demonstrates the use of failure detectors by application protocols within the TIOA framework specified in Chapter V.

Informally, an unreliable failure detector [20] can be viewed as a system service that is available to protocols, and when queried, it provides a list of processes, called

a *suspect list*, that it suspects as having crashed[1]. Each process has access to its own local detector module that provides (potentially unreliable) information about process crashes when queried.

VI.1. Formal specification

In order to provide a clearer understanding of failure detectors and how they are used by processes, we define and specify them formally as functions that map *fault patterns* to *histories*. Fault patterns are a formal conceptualization of crash faults, and failure detector histories are a formal notation to describe the value of the suspect list of a failure detector at each process at each instant of time.

VI.1.1. Fault patterns

A *fault pattern* is a function F that returns the set of crashed processes at any given time. That is, $F : \mathbb{R} \rightarrow 2^{\Pi}$; $F(t)$ denotes the set of processes that are crashed at time t.[2] Since we assume that crashed processes never recover, $\forall t, t' \in \mathbb{R}, t < t' : F(t) \subseteq F(t')$. We define $faulty(F) = \cup_{\forall t \in \mathbb{R}} F(t)$ and $correct(F) = \Pi - faulty(F)$; that is, $faulty(F)$ denotes all the processes that crash in F and $correct(F)$ denotes all the processes that are correct in F. A process that has not crashed at time t is said to be *live* at time t. We consider only fault patterns F in which at least one process is

[1]The canonical output of a failure detector is a subset of process ids, called a *suspect list*, which corresponds to the list of processes that the failure detector suspects as having crashed. This was later generalized to admit outputs of arbitrary formats [19] ranging from a single process id (in [19]) or a set of integers [3] to entire algorithms [53]. In this dissertation, we consider only those failure detectors whose output is a suspect list.

[2]Canonically, in [20] and subsequent work on failure detectors, time is modeled as moving in integer valued steps, and therefore the signature of a fault pattern function is $F : \mathbb{N} \rightarrow 2^{\Pi}$. However, since we model time as a continuous function (see Section V.2.1.1), we adopt a different signature for a fault pattern.

correct; that is, $correct(F) \neq \emptyset$; let the set of all such fault patterns be denoted $\mathcal{F}$.

VI.1.2. Histories

A *failure detector history* is a function H that maps $\Pi \times \mathbb{R}$ to 2^{Π}. $H(p_i, t)$ is the value of the suspect list of the failure detector module at process p_i at time t. If $p_j \in H(p_i, t)$, then we say process p_i suspects process p_j at time t in H. Note that it is possible for failure detector modules at two processes to have different suspect lists at the same instant; that is, if $p_i \neq p_j$, then $H(p_i, t) \neq H(p_j, t)$ is possible. Let $\mathcal{H}$ denote the set of all possible failure detector histories.

VI.1.3. Failure detectors

A *failure detector* is a function $\mathcal{D}$ that maps each fault pattern F to a subset of histories; that is $\mathcal{D} : \mathcal{F} \rightarrow 2^{\mathcal{H}}$. The value $\mathcal{D}(F)$ denotes the set of failure detector histories that may be output by a failure detector $\mathcal{D}$ for a fault pattern F. In different executions of a system with the same fault pattern F, the failure detector $\mathcal{D}$ may output different histories from $\mathcal{D}(F)$. Different definitions of the function $\mathcal{D}$ specifies different *failure detector classes*. Traditionally, $\mathcal{D}$ is defined through two abstract properties: *completeness* and *accuracy*.

Informally, *completeness* and *accuracy* denote the kinds of unreliability in the suspect list of a failure detector: false negatives (*i.e.*, failing to suspect a crashed process) or false positives (*i.e.*, wrongfully suspecting a correct process). *Completeness* restricts the false negatives, while *accuracy* restricts the false positives.

VI.1.3.1. Completeness

We consider two kinds of completeness properties from [20]: *weak* and *strong* completeness.

Weak Completeness requires that every crashed process is eventually and permanently suspected by *some* correct process.

Strong Completeness requires that every crashed process is eventually and permanently suspected by *all* correct processes. Note that weak and strong completeness are known to be computationally equivalent [20], and hence the remainder of this dissertation will only consider failure detectors that satisfy strong completeness.

Formally, a failure detector $\mathcal{D}$ satisfies strong completeness if and only if:

$\forall F \in \mathcal{F}, \forall H \in \mathcal{D}(F), \forall i \in correct(F), \forall j \in faulty(F), \exists t \in \mathbb{R} : \forall t' \geq t : j \in H(i, t')$

Strong completeness is easy to satisfy in isolation. Any failure detector whose output is always the set Π at all processes at all times trivially satisfies strong completeness. However, such a failure detector provides no useful information about process crashes. In order for a failure detector to be useful, in additional to satisfying completeness, it also needs to satisfy an *accuracy* property.

VI.1.3.2. Accuracy

We consider four kinds of accuracy properties from [20]: *perpetual strong accuracy*, *eventual strong accuracy*, *perpetual weak accuracy*, and *eventual weak accuracy*.

Perpetual strong accuracy requires that for every pair of processes i and j, i is not suspected by j before i crashes. The word 'perpetual' is often dropped to denote this property Strong Accuracy.

Formally, a failure detector $\mathcal{D}$ satisfies strong accuracy if and only if:

$\forall F \in \mathcal{F}, \forall H \in \mathcal{D}(F), \forall t \in \mathbb{R}, \forall i, j \in \Pi - F(t) :: i \notin H(j, t)$

Eventual strong accuracy requires that eventually no correct process is ever suspected by any other correct process.

Formally, a failure detector $\mathcal{D}$ satisfies eventual strong accuracy if and only if:

$\forall F \in \mathcal{F}, \forall H \in \mathcal{D}(F), \forall i, j \in correct(F), \exists t \in \mathbb{R} : \forall t' \geq t : i \notin H(j, t')$

Perpetual weak accuracy requires that there exist some correct process that is never suspected by any other process. Again, the word 'perpetual' is often dropped to denote this property Weak Accuracy.

Formally, a failure detector $\mathcal{D}$ satisfies weak accuracy if and only if:

$\forall F \in \mathcal{F}, \forall H \in \mathcal{D}(F), \exists i \in correct(F), \forall t \in \mathbb{R}, : \forall j \in \Pi - F(t) : i \notin H(j, t)$

Eventual weak accuracy requires that there exist some correct process that is eventually never suspected by any other correct process.

Formally, a failure detector $\mathcal{D}$ satisfies eventual weak accuracy if and only if:

$\forall F \in \mathcal{F}, \forall H \in \mathcal{D}(F), \exists i \in correct(F), \exists t \in \mathbb{R} : \forall t' \geq t, \forall i \in correct(F) : i \notin H(j, t')$

VI.1.4. Chandra-Toueg failure detector classes

Based on the aforementioned properties, Chandra and Toueg introduced four classes of failure detectors in [20] which form the Chandra-Toueg hierarchy [3]: *perfect failure detector*, *eventually perfect failure detector*, *strong failure detector* and *eventually strong failure detector*.

Perfect failure detectors (denoted $\mathcal{P}$) are the class of failure detectors that satisfy strong completeness and (perpetual) strong accuracy. That is, a perfect failure detector never suspects processes before the latter crash, and it upon suspected a crashed process, the suspicion is permanent.

Strong failure detectors (denoted $\mathcal{S}$) are the class of failure detectors that

[3] In fact, [20] introduces eight failure detector classes in the original Chandra-Toueg hierarchy of failure detector. But the computational equivalence between weak and strong completeness collapses this hierarchy to just four failure detectors.

satisfy strong completeness and (perpetual) weak accuracy. That is, a strong failure detector never suspects *some* correct process, and a strong failure detector eventually and permanently suspects all crashed processes. The correct process that is never suspected may vary from one execution to another.

Eventually perfect failure detectors (denoted $\Diamond\mathcal{P}$) are the class of failure detectors that satisfy strong completeness and eventual strong accuracy. That is, $\Diamond\mathcal{P}$ may wrongfully suspect a correct process, but only during a finite prefix of an execution. Eventually, $\Diamond\mathcal{P}$ never suspects any correct process and permanently suspects crashed processes. Although every execution is known to contain a suffix in which $\Diamond\mathcal{P}$ never suspects correct processes and always suspects crashed processes, the starting time of such a suffix is unknown (and can vary from one execution to another).

Eventually strong failure detectors (denoted $\Diamond\mathcal{S}$) are the class of failure detectors that satisfy strong completeness and eventual strong accuracy. That is, there exists *some* correct process that $\Diamond\mathcal{S}$ may wrongfully suspect during a finite prefix of an execution. Eventually, $\Diamond\mathcal{S}$ stops suspecting *that* correct process [4], and $\Diamond\mathcal{S}$ eventually and permanently suspects all crashed processes. Although every execution is known to contain a suffix in which $\Diamond\mathcal{S}$ never suspects some correct process and always suspects all crashed processes, the starting time of such a suffix is unknown. The starting time of such a suffix may vary from one execution to another, and furthermore, the correct process that is never suspected in such a suffix may vary from one execution to another as well.

---

[4] $\Diamond\mathcal{S}$ provides no guarantees for other correct processes in the system. They may be suspected infinitely often, finitely often, or never.

VI.2. Failure detectors within the formal TIOA framework

Recall from Chapter V that a distributed system is modeled by a TIOA where each process communicates with other processes through communication links and consists of multiple protocols of which one protocol is designated to be the *control* protocol, and the control protocol's program actions are scheduled by the system model scheduler through the action takeStep. Within this framework, a failure detector may run as a collection of protocols where one protocol at each process behaves as the local failure detector module and all other protocols interact with the local failure detector module through actions *query* and *response*. The remainder of this section describes the behavior, role, and placement of a local failure detector module within the TIOA framework.

VI.2.1. Failure detector protocol

We implement failure detectors as program actions of the control protocol in a partially synchronous system. The foregoing decision is based on three observations: (1) Chandra-Toueg failure detectors cannot be implemented in asynchronous systems (which follows from the impossibility of consensus in asynchrony [42], the solvability of consensus in partial synchrony [35], and the solvability of consensus using $\Diamond\mathcal{S}$ [19]), (2) Chandra-Toueg failure detectors can be implemented in various models of partial synchrony [45], and (3) failure detectors supplant the assumptions about partial synchrony within a distributed system; that is, algorithms that use failure detectors are assumed to execute under asynchrony. Consequently, in order to utilize the partial synchronism in the underlying system and supplant the former within the framework, the failure detector actions are placed within the control protocol.

Note that properties of a failure detector are defined with respect to its *suspect*

list which is a subset of processes that the putative failure detector module at a given process currently suspects. Hence, we mandate the existence of a variable suspectList — a set of process IDs — in the failure detector (control) protocol at each process, and the value of suspectList should be identical to the *suspect list* of the putative local failure detector module implemented by the protocol.

Application protocols that use the failure detector obtain the suspect list information from the failure detector by querying the latter. We model this behavior through two actions: query and response. The action query is an output action of the application protocol and an input action of the failure detector, and the action reponse is an output action of the failure detector protocol and an input action of the application protocol.

When an application protocol r invokes the action query of the failure detector protocol r_0, a variable queryInProgress[r] is set to $true$. The time evolution function at r_0 is stopped when queryInProgress[r] is $true$. Also, the precondition for the action response in r_0 is queryInProgress[r] $= true$ and response sends the current value of suspectList; that is, the failure detector responds to the query immediately with the current suspect list[5].

The TIOA specification of a failure detector (control) protocol is given in Algorithm 11.

---

[5]In a strict sense, within an asynchronous system, it is not necessary for a failure detector to respond with the current suspect list immediately. It is sufficient if the latency of the response is finite and order of outputs respect the real-time order. However, a common argument (based on private communications with other researchers in the field) for an immediate response to a query is that failure detectors are expected to periodically write the current value of the suspect list to a shared variable that is readable by all protocols.

Algorithm 11 Failure Detector TIOA.

```
automaton controlProtocol(i: pIndex, r_0: rIndex)
                                        //Failure Detector Protocol r_0 at process i
type pIndex = enumeration of p_1,p_2,...,p_n where Π = {p_1, p_2, ..., p_n}
type rIndex = enumeration of r_0, r_1, r_2,...,r_m where ρ = {r_0, r_1, r_2, ..., r_m}
                                        and r_0 refers to the control protocol itself
type Packet = tuple of message: Type, sent: Real, deadline: Real

 signature (in addition to the actions specified in Algorithm 1)
  input query(i: pIndex, r: rIndex)
  output response(i: pIndex, r: rIndex, suspectList: set[Index]
 states (in addition to the variables defined in Algorithm 1)
  now: Real ← 0                         //As defined in Algorithm 1: Current time
  queryInProgress: Array[rIndex, Boolean] ← constant(false)   //'true' ⇒ stop time
  suspectList : Set[pIndex] ← ∅,                              //Suspect List

 transitions
  input query(i, r)                              //Client queries the failure detector
   effect
    queryInProgress[r] ← true
  output response(i, r, suspectList)     //Failure detector responds to the client's query
   precondition
    queryInProgress[r] = true
   effect
    queryInProgress[r] ← false           //Return the set suspectList as the response
  input takeStep()
   effect
    //The suspectList is updated by the program action
    //The program action is implementation specific, and it will be discussed when
    //we investigate specific failure detector implementations
 trajectories
  stop when
   ∃r∈ ρ::queryInProgress[r] = true     //Ensure 0 time between query and response
  evolve
   d(now) = 1
```

VI.2.2. Application protocols

Since application protocols are assumed to execute their program action based on the recently received value of the suspect list, we have to ensure that, in every execution, the failure detector is queried at least once between two consecutive program actions of an application protocol. We accomplish this with the following modifications to the application protocol.

An application protocol that uses the failure detector information is assumed to have the variable suspectList that stores the value returned in the input action response. The effect of the input action response is to set a variable slRefreshed to *true*. The variable slRefreshed indicates that the suspect list information has been refreshed since the last time the application protocol executed a program action. The precondition for the program (internal) action takeStep is augmented from live = true to (live = true) $\wedge$ (slRefreshed = true) and the effect of takeStep is augmented with slRefreshed $\leftarrow$ false. Since slRefreshed is set to *true* only by the action response, and the failure detector module invokes response only when query is invoked, we ensure that the suspect list if refreshed by response at least once between two consecutive program actions by the application protocol.

The modified framework of the application protocol is shown in Algorithm 12.

VI.3. Solvability

Recall from section V.7 that a problem P is a predicate on the sequence of input and output actions that must be satisfied in all executions of a distributed system. If there exist a set of guarded commands $\mathcal{G}$ such that executing $\mathcal{G}$ as the program action of a protocol r in a system model $\mathcal{M}$ guarantees that all the executions of $\mathcal{M}$ satisfy the predicate P, then the problem P is said to be solvable in $\mathcal{M}$. We now

Algorithm 12 TIOA for a Failure-Detector-Querying Application Protocol.

automaton applicationProtocol(r: rIndex, i: pIndex) *App. Protocol r at process i*
type pIndex = **enumeration of** p_1,p_2,...,p_n **where** $\Pi = \{p_1, p_2, \ldots, p_n\}$
type rIndex = **enumeration of** r_0, r_1, r_2,...,r_m **where** $\rho = \{r_0, r_1, r_2, \ldots, r_m\}$
and r_0 refers to the control protocol itself
type Packet = **tuple of** message: M, sent: Real, deadline: Real

signature
output rSend(m: Type, r: rIndex, j: pIndex) *//As defined in Algorithm 3*
input rReceive(m: Type, r: rIndex, j: pIndex) *//As defined in Algorithm 3*
internal takeStep() *//Executes a program action*
input crashProcess(i: pIndex, r: rIndex) *//As defined in Algorithm 3*
input query(i: pIndex, r: rIndex)
output response(i: pIndex, r: rIndex, suspectList: set[Index])
states
now: Real ← 0 *//Current time*
sBuffer: Array[pIndex, Queue[Type]] ← constant($\emptyset$) *//As defined in Algorithm 3*
rBuffer: Array[pIndex, Queue[Type]] ← constant($\emptyset$) *//As defined in Algorithm 3*
live: Boolean ← true *//Set to false upon crashing*
slRefreshed: Boolean ← false *//Set to true by* response
suspectList : Set[pIndex] ← $\emptyset$, *//Suspect List*

transitions
internal takeStep()
precondition
(live = true) $\wedge$ (slRefreshed = true)
effect
execute an enabled program action
slRefreshed ← false
output query(i, r)
effect
//empty
input response(i, r, suspectList)
effect
slRefreshed ← true
trajectories
evolve
d(now) = 1

define what it means for a problem P to be *solvable* by a failure detector $\mathcal{D}$.

Informally, we say that a problem P is solvable by failure detector $\mathcal{D}$ if there exists an algorithm $\mathcal{G}$ that potentially queries $\mathcal{D}$ to satisfy the predicate P in an asynchronous system. More precisely, consider a system $\mathcal{M}$ where the program action of the control protocol r_0 satisfies the properties of a failure detector $\mathcal{D}$. A problem P is said to be solvable by $\mathcal{D}$ if and only if there exists an *asynchronous algorithm* $\mathcal{G}$ that solves P in $\mathcal{M}$ while the program actions of the control protocol r_0 satisfies $\mathcal{D}$. Recall that program actions of an application protocol are executed at arbitrary times and are subject only to weak fairness. That is, program actions of an application protocol do not provide any temporal guarantees. Therefore, the asynchronous algorithm $\mathcal{G}$ can be executed as the program action of an application protocol.

VI.4. Reducibility

Failure detectors and distributed system models are compared with each other through the notion of *reducibility*. Informally, a failure detector $\mathcal{D}$ is reducible to a failure detector $\mathcal{D}'$ with respect to the asynchronous system model if and only if an asynchronous system augmented with $\mathcal{D}$ can implement $\mathcal{D}'$. That is, $\mathcal{D}$ is reducible to $\mathcal{D}'$ with respect to the asynchronous system if and only if for every system model $\mathcal{M}$ whose control protocol satisfies the properties of $\mathcal{D}$, there exists an asynchronous algorithm $\mathcal{G}$ such that, if $\mathcal{G}$ is executed as the program action of an application protocol r in $\mathcal{M}$, then r satisfies the properties of $\mathcal{D}'$. We denote this $\mathcal{D}' \preceq \mathcal{D}$.

Similarly, we also define reducibility between failure detectors and system models. A failure detector $\mathcal{D}$ is *reducible* to a system model $\mathcal{M}$ if there exists a set of guarded commands $\mathcal{G}$ executed as the program action of an application protocol r in every system $\mathcal{M}'$ where the program action of the control protocol satisfies the properties

$\mathcal{D}$ such that: (1) the protocol r has the same TIOA signature as the system model $\mathcal{M}$; that is, protocol r has an output action takeStep, an input action receive, and an output action send, and (2) all the computational and communicational constraints satisfied by $\mathcal{M}$ are satisfied by $\mathcal{G}$. The aforesaid definition is illustrated in (a part of) Fig. 6 where the program action of the control protocol satisfies the properties of the failure detector $\mathcal{D}$ and the application protocol in Fig. 6 has the output action takeStep, the output action receive, and the input action send; if the application protocol satisfies the properties of system model $\mathcal{M}$, then $\mathcal{D}$ is reducible to $\mathcal{M}$.

A failure detector $\mathcal{D}$ is said to be *solvable* in $\mathcal{M}$ if there exists a set of guarded commands $\mathcal{G}$ such that when the control protocol r_0 at each process at $\mathcal{M}$ is as defined in Algorithm 11 and $\mathcal{G}$ is executed as the program action of r_0, then the changes in the value of suspectList and the sequence of values output by the action response satisfies the properties of $\mathcal{D}$. The aforesaid definition is illustrated in (a part of) Fig. 6 where the system model satisfies the properties of $\mathcal{M}$. If the control protocol in Fig. 6, which has an input action query and an output action response, satisfies the properties of the failure detector $\mathcal{D}$, then $\mathcal{D}$ is solvable in $\mathcal{M}$.

If a failure detector $\mathcal{D}$ is reducible to a failure detector $\mathcal{D}'$ and $\mathcal{D}'$ is reducible to $\mathcal{D}$ (with respect to the asynchronous system), then $\mathcal{D}$ and $\mathcal{D}'$ are said to be *equivalent* (with respect to the asynchronous system); denoted $\mathcal{D} \equiv \mathcal{D}'$. Formally, $(\mathcal{D} \preceq \mathcal{D}') \wedge (\mathcal{D}' \preceq \mathcal{D}) \Leftrightarrow \mathcal{D} \equiv \mathcal{D}'$.

Similarly, we can define the notion of equivalence between a failure detector $\mathcal{D}$ and a system model $\mathcal{M}$ as follows: if $\mathcal{D}$ is reducible to $\mathcal{M}$ and $\mathcal{D}$ is solvable in $\mathcal{M}$, then $\mathcal{D}$ is said to be *equivalent* to $\mathcal{M}$, denoted $\mathcal{D} \equiv \mathcal{M}$. Formally, $(\mathcal{D} \preceq \mathcal{M}) \wedge (\mathcal{M} \preceq \mathcal{D}) \Leftrightarrow \mathcal{D} \equiv \mathcal{M}$.

The schematic TIOA-based construction to establish the equivalence between a system model $\mathcal{M}$ and a failure detector $\mathcal{D}$ is shown in Fig. 6.

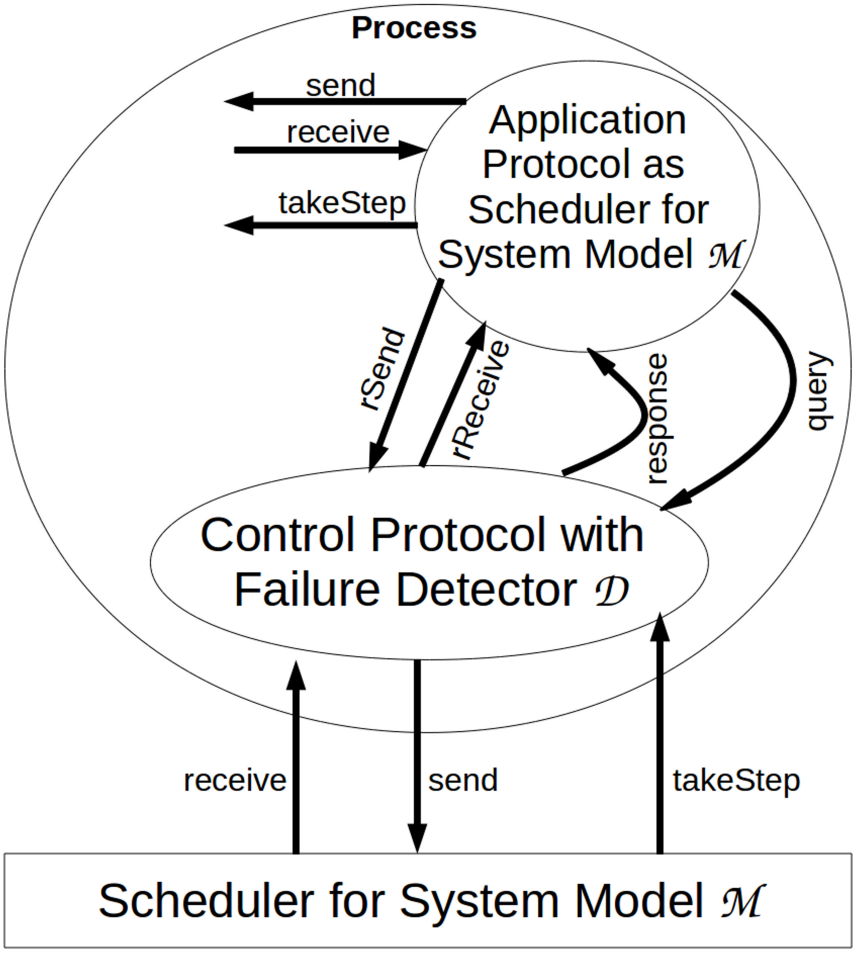

Fig. 6 TIOA construction to show equivalence between failure detector $\mathcal{D}$ and system model $\mathcal{M}$. If the failure detector implemented within the control protocol of system model $\mathcal{M}$ satisfies properties of $\mathcal{D}$, then we say that $\mathcal{D}$ is solvable in $\mathcal{M}$. If the application protocol in the figure implements a scheduler using the failure detector $\mathcal{D}$ (from the control protocol), and the scheduler satisfies the properties of $\mathcal{M}$, then we say that $\mathcal{D}$ is reducible to $\mathcal{M}$.

CHAPTER VII

CONSTRUCTING $\mathcal{M}_*$

Time is an illusion. Lunchtime, doubly so.

The Hitchhiker's Guide to the Galaxy, 1979
– *Douglas Adams*

In this chapter we construct fairness-based $\mathcal{M}_*$ (specifically, fairness-based $\mathcal{M}_2$) using a failure detector.[1] Recall from Chapter IV that the construction of $\mathcal{M}_*$ from empirical systems follows three steps. The first step is to construct an appropriate failure detector on top of empirical systems; the second step is to implement reliable communication on top of empirical systems; and the third step is to construct $\mathcal{M}_*$ using the thus constructed failure detector and reliable channels.[2]

The issue central to the methodology is the choice of the failure detector to be employed in our constructions and algorithmic transformations. Any failure detector $\mathcal{D}$ that we choose must satisfy the following properties for our methodology to work: (1) $\mathcal{D}$ must be solvable in empirical systems, (2) reliable communication must be solvable by $\mathcal{D}$, and (3) $\mathcal{M}_*$ must be solvable by $\mathcal{D}$.

It is unfeasible to explore all possible failure detectors that are solvable in empirical systems and verify if any of these failure detectors solve $\mathcal{M}_*$, or vice versa. Therefore, we have to adopt a different approach. We consider the 'weakest' failure detector that can implement the $\mathcal{M}_*$ system models and check if this failure detector can be solved by empirical systems. If it turns out that this failure detector is

---

[1]The results from this chapter have been published in [67].

[2]Note that we established in Section III.2 that real-time based $\mathcal{M}_*$ is impossible to construct on empirical systems, and we saw that fairness-based $\mathcal{M}_*$ could be plausible on empirical systems. Hence, we focus only on fairness-based $\mathcal{M}_*$ models here.

not solvable by empirical systems, then it proves that implementing $\mathcal{M}_*$ on top of empirical systems is impossible.

Thus, an intermediate goal of this dissertation is to find the 'weakest' failure detector that can solve $\mathcal{M}_*$. Before we find such a failure detector, we need to define what it means for a failure detector to be the 'weakest' to solve $\mathcal{M}_*$. By treating the system model specification of $\mathcal{M}_2$ as a problem specification, we state that a failure detector $\mathcal{D}$ is the 'weakest' to solve $\mathcal{M}_*$ if and only if the following are true: (1) $\mathcal{D}$ is solvable in $\mathcal{M}_*$; that is, $\mathcal{M} \preceq \mathcal{D}$, and (2) all failure detectors that solve $\mathcal{M}_*$ are reducible to $\mathcal{D}$; that is, $(\mathcal{M} \preceq \mathcal{D}') \Rightarrow (\mathcal{D} \preceq \mathcal{D}')$.

An ideal candidate for such a failure detector is one that is equivalent to $\mathcal{M}_*$. This follows from the definition of equivalence. If $\mathcal{D} \equiv \mathcal{M}_*$, then $\mathcal{M}_* \preceq \mathcal{D}$ and for any $\mathcal{D}'$ such that $\mathcal{M}_* \preceq \mathcal{D}'$, we know $\mathcal{D} \preceq \mathcal{M}_*$, from transitivity we know that $\mathcal{D} \preceq \mathcal{D}'$.

Thus, the goal of this chapter is to find a failure detector $\mathcal{D}$ such that $\mathcal{M}_* \equiv \mathcal{D}$.

VII.1. Methodology

We consider a candidate failure detector $\mathcal{D}$. If we design a partially synchronous algorithm $\mathcal{G}$ in $\mathcal{M}_*$ (either $\mathcal{M}_1$, $\mathcal{M}_2$, or $\mathcal{M}_3$ system models) such that the control protocol in $\mathcal{M}_*$ implements $\mathcal{D}$, then we have shown $\mathcal{M}_* \preceq \mathcal{D}$. Similarly, given a system $\mathcal{M}$ whose control protocol implements $\mathcal{D}$, if we design an asynchronous algorithm $\mathcal{G}$ in an application protocol r such that protocol r has an output action takeStep, an input action receive, and an output action send, and all the computational and communicational constraints satisfied by $\mathcal{M}_*$ are satisfied by $\mathcal{G}$, then we have shown $\mathcal{D} \preceq \mathcal{M}_*$. Combining the two results we have $(\mathcal{M}_* \preceq \mathcal{D}) \wedge (\mathcal{D} \preceq \mathcal{M}_*) \Leftrightarrow \mathcal{M}_* \equiv \mathcal{D}$.

Given that the $\mathcal{M}_*$ models are to implemented on top of empirical systems,

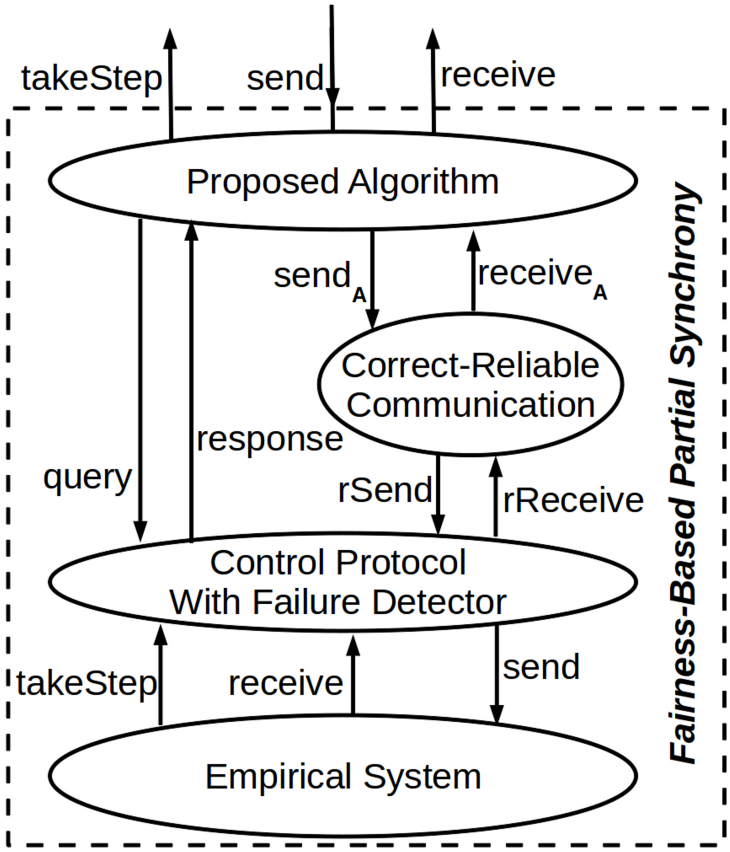

Fig. 7 TIOA schematic for building fairness-based partially-synchronous system on top of the empirical system using a failure detector.

our methodology must also consider the issue of message loss. Specifically, recall that empirical systems permits infinite message loss whereas the $\mathcal{M}_*$ models require eventually reliable links. Therefore, in constructing the $\mathcal{M}_2$ model on using the appropriate failure detector, we address the issue of reliability by employing an application protocol to implement correct-reliable channels on top of the lossy communication links offered by the rSend and rReceive actions of the control protocols at each process. We assume that the application protocol (implementing the correct-reliable communication) utilizes any of the several existing algorithms [1–3, 11, 37] to imple-

ment correct-reliable communication on top of fair-lossy channels. The schematic for implementing fairness-based partial synchrony (including $\mathcal{M}_*$) using a failure detector is shown in Fig. 7.

VII.2. $\Diamond\mathcal{P}$: a candidate failure detector

We show that the candidate failure detector that is equivalent to $\mathcal{M}_*$ is $\Diamond\mathcal{P}$. By following the methodology described in Section VII.1, we show that $\Diamond\mathcal{P}$ is solvable in $\mathcal{M}_*$ in section VII.3, and in section VII.4 we show that $\mathcal{M}_*$ is solvable in $\Diamond\mathcal{P}$.

VII.3. Implementing Chandra-Toueg failure detectors

The guarded commands in Algorithm 13 implement $\Diamond\mathcal{P}$ when executed as the program action of the control protocol in $\mathcal{M}_*$. In fact, the algorithm described in Algorithm 13 is a universal construction that implements the Chandra-Toueg failure detectors $\mathcal{P}$, $\Diamond\mathcal{P}$, $\mathcal{S}$, and $\Diamond\mathcal{S}$, respectively, if the underlying system model satisfies the properties of $\mathcal{AF}$, $\mathcal{SF}$, $\Diamond\mathcal{AF}$ (identical to $\mathcal{M}_*$, see Section III.2.1), and $\Diamond\mathcal{SF}$, respectively.

VII.3.1. Algorithm description

In Algorithm 13, the failure detector module (control protocol) at process i maintains a timer $\mathsf{timerValue}_j$ for each process j in the system which counts down from $k+d+1$ to 0, where, in the various system models described in Section V.5.4, the bounds on fairness are specified by the existence of k-proc-fair and d-com-fair processes. The timer $\mathsf{timerValue}_j$ is decremented by 1 in each step (line 10). In each step process i receives zero or more messages from all other processes (line 2) and sends a heartbeat to each process j in the system (line 4). If i receives a heartbeat from j, then i deletes j from the set $\mathsf{suspectList}$ (line 6) and resets the timer to $k+d+1$ (line 7). If

timerValue$_j$ is decremented to 0, then i adds j to suspectList (line 9). Here we assume that when the failure detector module is queried, it returns suspectList. Hence, when i adds j to suspectList, i is said to suspect j as having crashed; when deletes j from suspectList, i is said to trust j.

Algorithm 13 Implementing Chandra-Toueg failure detectors in system models where (some) processes are k-proc-fair and d-com-fair.

```
constant timeOut ← k + d + 1
set suspectList ← ∅
∀j ∈ Π − {i} :
  integer timerValue_j ← timeOut
-------------------------------------------------------------------------
1 :  {true} ⟶                                                  Action 1
2 :     {\sf msgSet} ← ∪_{∀j∈Π−{i}} {j: receiveMsg (⟨HB⟩, j)}
                                  //Receives zero or more messages from each process
3 :     ∀j ∈ Π − {i} do
4 :       sendMsg (⟨HB⟩, j)                    //Send a heartbeat to each process
5 :       if (⟨HB, j⟩ ∈ msgSet)
6 :         suspectList ← suspectList − {j}    //Trust upon receiving a heartbeat
7 :         timerValue_j ← timeOut                              //Reset timer
8 :       if (timerValue_j = 0)
9 :         suspectList ← suspectList ∪ {j}          //Suspect upon timer expiry
10 :      timerValue_j ← max(timerValue_j − 1, 0)   //Decrement timer for each process
```

VII.3.2. Proof of correctness

We now show that the action system in Algorithm 13 satisfies strong completeness and different accuracy properties depending on the underlying system model. For the purpose of the proofs, we consider an arbitrary control-protocol-restricted run α where the program action of the control protocol executes the guarded commands from Algorithm 13.

Theorem VII.3.1. *The action system in Algorithm 13 satisfies* strong completeness*; that is, there exists a time after which every crashed process is permanently suspected.*

Proof. In α, processes send heartbeats in every step. If a process i crashes at time t in α, i stops taking steps after t, and so stops sending heartbeats. Eventually, all the (finite) heartbeats sent by i are delivered. Let the last such delivery be at time $t' \geq t$. Inspection of the code reveals that the maximum value of $\mathsf{timerValue}_i$ at j at time t' is $k + d + 1$. Thereafter, in every step executed by a process j after time t', $\mathsf{timerValue}_i$ is decremented (if $\mathsf{timerValue}_i$ is not already 0) until j receives another heartbeat from i. Process j resets $\mathsf{timerValue}_i$ to $k + d + 1$ only upon receiving a heartbeat from i. Since we have established that no such heartbeats are received by j after t', it follows that in at most $k + d + 1$ steps, $\mathsf{timerValue}_i$ is decremented to 0 at all processes j, and so j starts suspecting i (in line 9). Since j does not receive any more heartbeats from i, j suspects i permanently. □

We prove accuracy properties in two steps. In the first step (Lemma VII.3.2), we show that a correct process is trusted infinitely often; that is, if a correct process j trusts a correct process i at time t, then there exists a time $t' > t$ such that j trusts i at time t'. Note that this permits j to (falsely) suspect i in the open interval (t, t'). In the second step (Lemma VII.3.3), we show that if a process i is trusted after it is k-proc-fair and d-com-fair, then i will be continuously trusted until i crashes. Lemmas VII.3.2 and VII.3.3 are used to prove the various accuracy properties satisfied by Algorithm 13, depending on the underlying system model.

Lemma VII.3.2. *In α, if process i is correct, then every correct process trusts i infinitely often; that is, $\forall t \in \mathbb{N}$, there exists a time $t' > t$ such that j trusts i at time t'*

Proof. From lines 5–6 of Algorithm 13 we know that a correct process (say) $j \neq i$ trusts process i upon receiving a message from i. We also know that i sends heartbeats to all processes in every step (Algorithm 13). Hence, if i is correct, then i

takes steps infinitely often, and sends heartbeats infinitely often. Correct-reliable communication guarantees that no heartbeat is lost. Therefore, all correct processes receive heartbeats from i infinitely often, and hence, execute lines 5–6 of Algorithm 13 infinitely often. Therefore, all correct processes trust i infinitely often. □

Lemma VII.3.3. *In α, if i becomes k-proc-fair and d-com-fair from time t, and the value of* $\mathsf{timerValue}_i$ *is $k+d+1$ at a process (say) j at time $t' \geq t$, then from time t' onwards, i is never suspected by j until i crashes.*

Proof. Let i become k-proc-fair and d-com-fair in α from time t. Let the value of $\mathsf{timerValue}_i$ be $k+d+1$ at a process (say) j at time $t' \geq t$. We know that the value of $\mathsf{timerValue}_i$ is decremented by 1 in each step until j receives a message from i. We now show that j is guaranteed to receive a message from i before $\mathsf{timerValue}_i$ is decremented to 0.

Note that i sends a heartbeat to j in each action that i executes. Given that i is k-proc-fair and d-com-fair, we know that i will send at least one heartbeat to j before j has taken $k+1$ steps after t', and this heartbeat is received by j before j has taken $k+d+1$ steps after t'. Recall that time t', the value of $\mathsf{timerValue}_i$ is $k+d+1$ and is decremented by 1 at every step taken by j. However, j receives at least one heartbeat from i within $k+d+1$ steps, and so the value of $\mathsf{timerValue}_i$ is reset to $k+d+1$ (in line 7) before it reaches 0.

Note that for j to start suspecting i, $\mathsf{timerValue}_i$ must be 0, and we have shown that if j starts trusting i, then the value of $\mathsf{timerValue}_i$ is reset to $k+d+1$ (in line 7) before it reaches 0. Therefore, from time t onwards, i is never suspected by j until i crashes. □

Theorem VII.3.4. *If the action system in Algorithm 13 is executed on a $\mathcal{AF}$ system model, Algorithm 13 implements the perfect failure detector $\mathcal{P}$.*

Proof. Recall that the $\mathcal{AF}$ system model guarantees that every process is k-proc-fair and d-com-fair from time $t = 0$. Also at time $t = 0$, at each process j, the value of $\mathsf{timerValue}_i = k + d + 1$ for every other processes i in the system. Applying lemma VII.3.3 with $t = t' = 0$, we know that i is never suspected by j until i crashes. Since i and j are arbitrary processes in the system. It follows that no process is suspected before it crashes. This, in conjunction with theorem VII.3.1 shows that every process is distinguished; that is, Algorithm 13 implements the perfect failure detector $\mathcal{P}$. □

Theorem VII.3.5. *If the action system in Algorithm 13 is executed on a $\Diamond\mathcal{AF}$ system model, Algorithm 13 implements the eventually perfect failure detector $\Diamond\mathcal{P}$.*

Proof. Consider a pair of correct processes i and j. Recall that the $\Diamond\mathcal{AF}$ system model guarantees that i process is k-proc-fair and d-com-fair from some (unknown) time t. From Lemma VII.3.2 we know that j trusts i infinitely often, which implies that value of $\mathsf{timervalue}_j$ at i is $k + d + 1$ infinitely often. Applying lemma VII.3.3 we know that eventually j never suspects i. On the other hand, if i is faulty and crashes in finite time, then from theorem VII.3.1 we know that eventually j always suspects i. In other words, i is eventually distinguished. Since i is an arbitrary process in the system, it follows that all the processes are eventually distinguished. That is, Algorithm 13 implements the eventually perfect failure detector $\Diamond\mathcal{P}$. □

Theorem VII.3.6. *If the action system in Algorithm 13 is executed on a $\mathcal{SF}$ system model, Algorithm 13 implements the strong failure detector $\mathcal{S}$.*

Proof. Recall that the $\mathcal{SF}$ system model guarantees that some correct process i is k-proc-fair and d-com-fair from time $t = 0$. Also at time $t = 0$, at each process j, the value of $\mathsf{timerValue}_i = k+d+1$. Applying lemma VII.3.3 with $t = t' = 0$, we know that i is never suspected by j. This, in conjunction with theorem VII.3.1 shows that some

correct process is distinguished and all faulty processes are eventually distinguished; that is, Algorithm 13 implements the strong failure detector $\mathcal{S}$. □

Theorem VII.3.7. *If the action system in Algorithm 13 is executed on a $\Diamond\mathcal{SF}$ system model, Algorithm 13 implements the eventually strong failure detector $\Diamond\mathcal{S}$.*

Proof. Recall that the $\Diamond\mathcal{SF}$ system model guarantees that eventually some correct process i is k-proc-fair and d-com-fair. Let j be a correct process. From Lemma VII.3.2 we know that j trusts i infinitely often, which implies that value of $\mathsf{timervalue}_j$ at i is $k + d + 1$ infinitely often. Applying lemma VII.3.3 we know that eventually j never suspects i. This, in conjunction with theorem VII.3.1 shows that some correct process is distinguished and all faulty processes are eventually distinguished; that is, Algorithm 13 implements the strong failure detector $\mathcal{S}$. □

VII.4. Extracting fairness from Chandra-Toueg failure detectors

In this section, we present a construction that 'extracts' the fairness encapsulated by the Chandra-Toueg failure detectors by implementing fairness-based partially-synchronous models in an asynchronous system augmented with a Chandra-Toueg failure detector. The algorithm presented is a universal construction for the Chandra-Toueg hierarchy in the sense that depending on the failure detector used by the algorithm, the appropriate fairness guarantees are provided by the constructed system model. We present the TIOA framework for such a construction next.

Recall that in a distributed system model the scheduler interacts with the processes in the system through three external actions: takeStep, send, and receive. Therefore, the construction presented in this section must provide these three actions to the processes. Specifically, the construction must provide (1) an output action takeStep that enables the control protocol of each process to execute a pro-

gram action, (2) an input action send for processes to send messages to each other, and (3) an output action receive for processes to receive messages from each other.

The failure detectors used in this construction have two external actions: query and response. The input action query is invoked by a process and as a result of the query, the failure detector returns the current value of *suspectList* through the output action query. We assume that the failure detector is implemented in the control protocol of the process. Consequently, the construction is implemented as an application protocol, and therefore is assumed to be asynchronous.

The asynchronous system interacts with our construction through two external actions: send and receive. Note that these send and receive actions are separate from the ones described earlier in Section V.3.1.2, and for convenience, we rename these actions as send_A and $\mathsf{receive}_A$ (the subscript A stands for 'asynchronous'). The input action send_A is invoked by the construction module at each process to send messages to each other, and the output action $\mathsf{receive}_A$ delivers these messages to the appropriate recipient construction modules. We assume that the communication links formed by these actions are *correct-reliable*; that is, messages send among correct processes is guaranteed to be delivered reliably (see Section V.5.2). Recall from Section VII.1 that we assume the existence of another application protocol that implements a correct-reliable channel on top of the lossy rSend and rReceive communication interface provided by the control protocol. That is, failure detectors are implemented in empirical systems, and an application protocol in the empirical system implements correct-reliable channels on top of the unreliable communication provided by the empirical system without affecting the messages sent among the failure detector modules within each process.

Recall that each application protocol has an internal action takeStep which executes the protocol's respective program action. Also recall that the fairness-based

partially-synchronous model implemented by an application protocol has an output action labeled takeStep. In order to resolve this issue of duplicate labels for these two distinct actions, we rename the internal takeStep action as myTakeStep.

VII.4.1. Fairness constraints

The properties that must be satisfied by the implemented fairness-based system are the following:

- **Local Progress.** The application protocol that implements the fairness-based system model must execute its output action takeStep infinitely at each correct process often regardless of process crashes in the system.
- **Fairness.** If the constituent failure detector in the schematic from Fig. 7 is:
  - $\mathcal{P}$, then all the processes in the fairness-based system are k-proc-fair and d-com-fair. That is, the application protocol implements the All Fair ($\mathcal{AF}$) system model.
  - $\mathcal{S}$, then some correct process is k-proc-fair and d-com-fair. That is, the application protocol implements the Some Fair ($\mathcal{SF}$) system model.
  - $\Diamond\mathcal{P}$, then all the processes in the system are eventually k-proc-fair and eventually d-com-fair. That is, the application protocol implements the Eventual All Fair ($\Diamond\mathcal{AF}$) system model.
  - $\Diamond\mathcal{S}$, then some correct process is eventually k-proc-fair and eventually d-com-fair. That is, the application protocol implements the Eventual Some Fair ($\Diamond\mathcal{SF}$) system model.

Algorithm 14 Signature and states of the TIOA for the partially-synchronous system model using failure detector $\mathcal{D}$. The state transitions are in Algorithm 15.

```
automaton fairnessBasedPartiallySynchronousSystem(pIndex i)
type pIndex = enumeration of p_1,p_2,...,p_n
type rIndex = enumeration of r_0,r_1,...,r_m
type pktType = enumeration of msgReq, msgResponse, permitReq, permitResponse
type pkt = tuple of payload: Type, t: pkType
type schedulerState = enumeration of waiting, active
type takeStepState = enumeration of disabled, enabled, completed
 signature
  output takeStep(i: pIndex)          //Enable control protocol at process i to take a step
  input crashProcess(i: pIndex)       //Process i is crashed. As defined in Algorithm 3
  output send_A(m: pkt, i: pIndex, receiver: pIndex)   //Send via correct-reliable link
  input receive_A(m: pkt, i: pIndex, sender: pIndex)   //Rcv from correct-reliable link
  input send(m: Type, i: pIndex, r: pIndex)    //Client (App.) send msg interface
  output receive(m: Type, i: pIndex, s: pIndex)   //Client (App.) receive msg interface
  internal myTakeStep(i: pIndex)                  //Execute a program action
  input query(i: pIndex, r: rIndex)               //As defined in Algorithm 12
  output response(i: pIndex, r: rIndex, suspectList: set[Index])
                                                  //As defined in Algorithm 12
 states
  live: Boolean ← true                            //As defined in Algorithm 12
  slRefreshed: Boolean ← false                    //As defined in Algorithm 12
  suspectList: Set[pIndex] ← ∅,                   //As defined in Algorithm 12
  now: Real ← 0                                   //Current time
  send_buffer: Array[s: pIndex, r: pIndex, Queue[Type]]   //Msgs sent by app
  receive_buffer: Array[s: pindex, r: pIndex, Queue[Type]] //Msgs to be delivered to app
  sBuffer: Array[r: pIndex, Queue[pkt]]           //Msgs to be sent to other processes
  msgReqQ: Array[s: pIndex, Queue[Type]]          //Queue of message requests
  msgResponseQ: Array[s: pIndex, Queue[Type]]     //Msg-request responses
  permitReq: Array[s: pIndex, Type]               //Contains request for permit
  permit: Array[s: pIndex, Type]          //Contains permit and associated information
  takeStepFlag: takeStepState ← disabled  //Denotes if takeStep should be executed
  s_i.state: schedulerState ← waiting     //Scheduler is initially set to waiting
  s_i.ht: Integer ← 0                             //Initial height is 0
  s_i.permit: Array[j: pIndex, Boolean] ← (i>j)  //Process with higher id holds the permit
  s_i.req: Array[j: pIndex, Boolean] ← (i<j)   //Lower-id process holds the request token
  s_i.ht: Array[j: pIndex, Integer] ← constant(0)          //Process heights
  s_i.seq: Array[j: pIndex, Integer] ← constant(0)   //Sequence no. for msgs from j
  s_i.maxAck: Array[j: pIndex, Integer] ← constant(0)  //The highest seq. no. acked by j
```

Algorithm 15 State transitions for the TIOA for the partially-synchronous system model using failure detector $\mathcal{D}$ shown in Algorithm 14.

```
transitions for fairnessBasedPartiallySynchronousSystem(pIndex i)
 output takeStep(i)
  precondition
   live ∧ (takeStepFlag = enabled)                    //Process i is not crashed
  effect
   takeStepFlag ← completed                  //Denote that takeStep has been executed
 input send(m, s, r)
  effect
   enqueue m in send_buffer[s,r]
 output receive(m, r, s)
  precondition
   m = receive_buffer[s, r]
  effect
   receive_buffer[s, r] ← ∅
 internal myTakeStep(i)
  precondition
   (live = true) ∧ (slRefreshed = true)
  effect
   execute program action in Algorithm 16
   slRefreshed ← false
 output send_A(m, r)
  precondition
   m = front sBuffer[r]
  effect
   dequeue m from sBuffer[r]
 input receive_A(m, s)
  effect
   switch (m.pktType)
    case msgReq: enqueue m.payload in msgReqQ[s]
    case msgResponse: enqueue m.payload in msgResponseQ[s]
    case permitReq: permitReq[s] ← m.payload
    case permitResponse: permit[s] ← m.payload
trajectories
 evolve
  d(now) = 1
```

Algorithm 16 Program action for the scheduler in the partially-synchronous system model using failure detector at process i.

```
1 :  {s_i.state = waiting} ⟶                                                    Action 1
2 :    ∀j ∈ Π − {i} where s_i.req[j] ∧ ¬s_i.permit[j] do                  //Request permit
3 :        enqueue ⟨permitReq, s_i.ht⟩ in sBuffer[j]; s_i.req[j] ← false
4 :  {permitReq[j] ≠null} ⟶                                                     Action 2
5 :    s_i.req[j] ← true; s_i.ht[j] ← permitReq[j].ht                  //Received permit request
7 :    if (s_i.permit[j] ∧ (s_i.state = waiting)∧                      //Send permit if s_i is waiting
         ((s_i.ht[j] < s_j.ht) ∨ ((s_i.ht[j] = s_i.ht) ∧ (i < j)))     //and s_j has higher priority
8 :        enqueue ⟨permitResponse, s_i.ht⟩ in sBuffer[j];
           s_i.permit[j] ← false; permitReq[j] ← null
9 :  {permit[j] ≠null} ⟶                                                        Action 3
10 :   s_i.permit ← true; s_i.ht[j] ← permit[j].ht                      //Received permit
12 :   if (s_i.req[j] ∧ (s_i.state = waiting)∧                          //Send permit if s_i is waiting
         ((s_i.ht[j] < s_j.ht) ∨ ((s_i.ht[j] = s_i.ht) ∧ (i < j)))     //and s_j has higher priority
13 :       enqueue ⟨permitResponse, s_i.ht⟩ in sBuffer[j]
14 :       s_i.permit[j] ← false; permit[j] ← null
15 : {(s_i.state = waiting) ∧ (∀j ∉ suspectList :: s_i.permit[j])} ⟶             Action 4
16 :   s_i.state ← active          //Active upon holding permits from trusted processes
17 :   foreach j in Π − {i}
18 :       increment s_i.seq[j] by 1   //Generate a new seq. no. to tag a request message
19 :       enqueue ⟨msgReq, s_i.seq[j]⟩ in sBuffer[j]   //Send request msgs to all processes
20 : { msgReqQ[j] not empty } ⟶                                                 Action 5
21 :   msgSet ← send_buffer[i, j];send_buffer[i, j] ← ∅      //Received a mesg request.
22 :   dequeue num from msgReqQ[j]
23 :   enqueue ⟨msgResponse, msgSet, num⟩ in sBuffer[j]   //Send the local send buffer
24 : { msgResponseQ[j] not empty } ⟶                                            Action 6
25 :   dequeue ⟨msgSet′, num⟩ from msgResponseQ[j]
26 :   ∀m ∈ msgSet′: enqueue m in receive_buffer[j]      //Add to local receive buffer
27 :   s_i.maxAck[j] ← max(num, s_i.maxAck[j])      //Update max. ack receive so far.
28 : {(s_i.state = active) ∧ (∀j ∈ Π − {i} :: ((s_i.maxAck[j] = s_i.seq[j]) ∨ (j ∈ suspectList)))
     ∧(∀j ∈ Π − {i} :: receive_buffer[j, i] = ∅) ∧ (takeStepFlag = disabled)} ⟶   Action 7
29 :   takeStepFlag ← enabled                          //Enable executing takeStep
30 : {(s_i.state = active) ∧ (∀j ∈ Π − {i} :: ((s_i.maxAck[j] = s_i.seq[j]) ∨ (j ∈ suspectList)))
     ∧(∀j ∈ Π − {i} :: receive_buffer[j, i] = ∅) ∧ (takeStepFlag = completed)} ⟶  Action 8
31 :   takeStepFlag ← disabled          //After executing takeStep, disable the action
32 :   s_i.ht ← min(∀j ∈ Π − {i} :: s_i.ht[j], s_i.ht) − 1   //Reduce height below all neighbors
33 :   ∀j ∈ Π − {i} where (s_i.permit[j])                    //do whose height is known.
34 :     enqueue ⟨permitResponse, s_i.ht⟩ in sBuffer[j];s_i.permit[j] ← false  //Send permits
35 :   s_i.state ← waiting          //Exit the active state after executing an app. step
```

VII.4.2. Algorithm description

The algorithm in Algorithms 14, 15, and 16 implements the scheduler for a distributed system model with dynamic priorities and permits. Algorithm 16 specifies the program action of the model, and Algorithm 14 shows the interface between the failure detector, communication links, the implemented model, and the scheduled application. The idea of dynamic priorities and permits (also called *forks*) is borrowed from the algorithms to solve the dining philosophers problem in [21] and [69]. All the processes are assigned a static id and all the ids are known to all the processes in the system.

The automaton defined in Algorithms 14–15, which we call the *model automaton*, reflects the schematic shown in Fig. 7 that was described earlier. The model automaton sends and receives messages on behalf of the scheduled application; the application sends and receives messages via the model automaton through the actions send and receive, respectively. The model automaton at process i maintains a send_buffer$[i, j]$ and a receive_buffer$[j, i]$ for each process j in the system. As shown in Algorithm 15, the action send simply enqueues the message to be sent to j in send_buffer$[i, j]$, and the action receive simply dequeues a messages from receive_buffer$[j, i]$ and delivers it to the application. The program action of the model (application protocol) automaton (in Algorithm 16) is responsible for removing messages from send_buffer and adding messages to receive_buffer.

The model automaton at a given process sends messages to, and receives messages from, the model automata at other processes through the actions send$_A$, and receive$_A$, respectively. The messages exchanged among the model automata at different processes are of four types: msgReq, msgResponse, permitReq, and permitResponse. Message type msgReq is sent from process (say) i to process (say) j to solicit appli-

cation messages sent from i to j. These messages are stored in the send_buffer$[j, i]$ at j. In response to a msgReq message, j sends a msgResponse message that contains all the application messages in send_buffer$[j, i]$. Similar to application messages, the shared permits between two processes (say) i and j are managed through permitReq and permitResponse messages. Process i requests the permit shared between i and j by sending a permitReq message to j, and j sends the shared permit to i by sending a permitResponse message.

The program action of the model (application protocol) automaton sends messages by enqueuing them in the buffer called sBuffer. The output action send$_A$ simply dequeues a message from sBuffer and sends it to the recipient process as shown in Algorithm 15. Messages are received from another process (say, j) in action receive$_A$ at process (say) i, and as shown in Algorithm 15, action receive$_A$ first determines the type of the message, and depending on the type of the messages does one of the following:

- If the message type is msgReq, then the message is added to the queue msgReqQ[j].
- If the message type is msgResponse, then the message is added to the queue msgResponseQ[j].
- if the message type is permitReq, then the value of permitReq[j] is updated with the contents of the message.
- If the message type is permitResponse, the the value of permit[j] is updated with the contents of the message.

Thus, the program action at the model automaton (Algorithm 16) takes receipt of various messages by checking the values of the corresponding data structures in

which the messages are stored.

The model automaton also maintains a variable takeStepFlag, which is initially set to *disabled*, to permit the scheduled application to take a step. When the variable takeStepFlag is set to *enabled*, the automaton executes the output action takeStep and then sets takeStepFlag to *completed*.

In addition to the state variables described so far, the program action specified in Algorithm 16 utilizes additional state variables. In order to denote their 'internal' scope (that is, they are used exclusively by the program action and no other action in the system), these variables are prefixed with s_i. where i denotes the process whose model (application protocol) program action uses them. For each process i, s_i.state which determines if the process is *waiting* or *active*. The height of a process is stored in the variable s_i.ht which is initially 0. For each process j in the system, i maintains the arrays: (a) s_i.permit$[j]$ to determine if the permit shared with j is currently held by i, (b) s_i.req$[j]$ to determine if the request token to request a permit from j is currently at i, and (c) s_i.ht$[j]$ which stores the last received value of j's height (in permits and request messages).

We now describe the program action of the model automaton specified in Algorithm 16. All processes start in the *waiting* state with the permits at higher-id processes and request tokens at lower-id processes, and takeStepFlag is *disabled*. For a *waiting* process to become *active*, it must collect all its shared permits. A *waiting* process requests missing permits in Action 1. Upon receiving such a request in Action 2, the process determines if the request should be honored based on the following condition: if the process is *waiting*, holds the shared permit, and the requesting process has greater height (or equal height and higher process-id), then the process relinquishes the permit. Otherwise the process simply holds the token and defers sending the permit if the permit is present.

Upon receiving a permit in Action 3, the process again determines if the permit should be kept/deferred or sent based on the same condition mentioned previously.

Once a *waiting* process (say) i receives all shared permits from processes not suspected by the failure detector $\mathcal{D}$ (whose output is stored in the local variable suspectList, and the value of suspectList is refreshed infinitely often), process i becomes *active* in Action 4. Upon becoming *active*, i sends an application-message request (denoted $\langle$msgReq$\rangle$) with a new sequence number (s_i.seq$[j]$) to each process j in the system in Action 4. Upon receiving such a message in Action 5, process j sends the contents of its *local send buffer* in a message of type $\langle$msgResponse$\rangle$; the sequence number associated with the received $\langle$msgReq$\rangle$ message is appended to the $\langle$msgResponse$\rangle$ message. When process i receives a message of type $\langle$msgResponse$\rangle$ in Action 6, the process adds contents of the message to its *local receive buffer* and updates its local state to reflect the latest sequence number for which i has received a response from j (stored in s_i.macAck$[j]$). Eventually i receives responses from all trusted processes for the $\langle$msgReq$\rangle$ messages sent with the latest sequence number (that is, s_i.seq$[j] = s_i$.maxAck$[j]$ for all j trusted by i); let us denote this *condition A*. Also, eventually, all the messages in the local receive buffer are delivered to the application (through output action receive in Algorithm 15); let us denote this *condition B*. After *condition A* and *condition B* are satisfied (and takeStepFlag is still *disabled*), Action 7 is enabled. In Action 7, the scheduler sets takeStepFlag to *enabled*; this enables the output action takeStep (in Algorithm 15) which executes the program action of the scheduled application. This mechanism of receiving application messages before enabling the action takeStep in Algorithm 15 ensures that an active process i 'waits on' all the messages sent by a correct and trusted process j; this guarantees that a correct and trusted process is also a com-fair process.

The action takeStep in Algorithm15, upon execution, changes the value of the

variable takeStepFlag to *completed*. After executing takeStep, *conditions A and B* remain satisfied and takeStepFlag is *completed*. This enabled Action 8.

In Action 8, the process exits its *active* state by setting takeStepFlag to *disabled*, reducing its height below all processes (whose shared permits it holds), sending all the permits away, and transiting to *waiting*. Relinquishing the shared permits before *waiting* ensures that correct and trusted processes become proc-fair processes as well.

VII.4.3. Proof of correctness

In this section we prove that the scheduler (for the distributed system model) in Algorithm 16 satisfies the local progress and fairness properties specified in Section VII.4. For the purpose of the proof, let the application protocol specified by Algorithms 14, 15, and 16 be denoted r_s. Consider an arbitrary execution α of the system and the corresponding r_s-protocol restricted run $\alpha|_{r_s}$.

Lost request tokens or permits can compromise progress, while duplicated request tokens or permits can compromise fairness. First we prove that every pair of processes share a unique permit and a unique request token. We use the following notation to denote that a message of type y is in transit from process i to j: $M^{y}_{i\to j}$.

Lemma VII.4.1. *For all states in $\alpha|_{r_s}$, there exists exactly one request token between each pair of live processes; that is, for all pairs of processes (i, j): $s_i.\mathsf{req}[j] \oplus s_j.\mathsf{req}[j] \oplus M^{permitReq}_{i\to j} \oplus M^{permitReq}_{j\to i}$.*

Proof. For each pair of processes, the initialization code creates a unique request token at the lower-priority process. Since communication channels are reliable, this token is neither lost nor duplicated while in transit. Only Actions 1 and 2 can modify the token variables. No token is lost, because every token received is locally stored (Action 2), and no token is locally removed unless it is sent (Action 1). No token is

duplicated, because every token sent is locally removed, and no absent token is ever sent (Action 1). Thus, token uniqueness is preserved. □

Lemma VII.4.2. *For all states in $\alpha|_{r_s}$, there exists exactly one permit between each pair of live processes; that is, for all pairs of processes (i, j): $s_i.\mathsf{permit}[j] \oplus s_j.\mathsf{permit}[j] \oplus M_{i\to j}^{permitResponse} \oplus M_{j\to i}^{permitResponse}$.*

Proof. For each pair of processes, the initialization code creates a unique permit at the higher-priority process. Since communication channels are reliable, this permit is neither lost nor duplicated while in transit. Only Actions 2, 3, and 5 modify the permit variables. No permit is lost, because every permit received is locally stored (Action 3), and no permit is locally removed unless it is sent (Actions 2, 3, & 8). No permit is duplicated, because every permit sent is locally removed, and no absent permit is ever sent (Actions 2, 3, & 8). Thus, permit uniqueness is preserved. □

In order to prove local progress, we are required to show that every correct process is guaranteed to take application steps infinitely many times. This proof is established in two steps. In the first step (Lemma VII.4.4), we show that a correct process is *active* only for a finite duration, and in that duration, the process executes the output action takeStep exactly once. In the second step (Lemma VII.4.6 and Theorem VII.4.7), given that a correct process is *active* only for a finite time, we establish that every *waiting* process eventually becomes *active*. Since correct processes alternate between *waiting* and *active*, it follows that a correct process becomes *active* infinitely many times, and therefore takes application steps infinitely many times.

Lemma VII.4.3. *For all states in $\alpha|_{r_s}$, for all pairs of processes (i, j) where $i \neq j$, $s_i.\mathsf{maxAck}[j]$ never exceeds $s_i.\mathsf{seq}[j]$; that is, $\forall i, j \in \Pi : i \neq j : s_i.\mathsf{maxAck}[j] \leq s_i.\mathsf{seq}[j]$.*

Proof. Initially, $s_i.\mathsf{seq}[j] = s_i.\mathsf{maxAck}[j] = 0$, therefore the lemma is true initially. Note that the only action that changes the value of $s_i.\mathsf{seq}[j]$ is Action 6, and Action 6 increments the value by 1. Therefore, if the lemma was true before i executed Action 6, then the lemma is true upon executing Action 6 as well.

The only action that changes the value of $s_i.\mathsf{maxAck}[j]$ is Action 6. If Action 6 increases the value of $s_i.\mathsf{maxAck}[j]$, then the increased value num is received by i in a message $\langle \mathsf{msgResponse}, msgSet', num \rangle$ from j. But note that j sends $\langle \mathsf{msgResponse}, msgSet', num \rangle$ to i only upon receiving $\langle \mathsf{msgReq}, num \rangle$ from i (Action 5). But in the message $\langle \mathsf{msgReq}, num \rangle$ sent by i to j (at time t'), the value of num (in line 25, Action 6, Algorithm 16) is $s_i.\mathsf{seq}[j]$ at time t'. Inspection of the algorithm reveals that $s_i.\mathsf{seq}[j]$ is non-decreasing. Therefore, the new $s_i.\mathsf{maxAck}[j]$ is either the current or a previous value of $s_i.\mathsf{seq}[j]$. Therefore, if the lemma was true before i executed Action 6, then the lemma is true upon executing Action 6 as well.

Thus, the lemma is true initially, and the lemma is true after executing any action that changes the values of $s_i.\mathsf{seq}[j]$ and $s_i.\mathsf{MaxAck}[j]$; thus proved. □

Now we are ready to show that all correct processes are *active* only for finite durations.

Lemma VII.4.4. *Let C be a state in $\alpha|_{r_s}$ at time t in which a process i is active. Then in some state C' at time $t' > t$, either i is crashed or i is waiting.*

Proof. Let process i become *active* in state C'' at time t'' (in Action 4) and remain *active* through time $t \geq t''$ in state C. If i is faulty, then i crashes at some time $t' > t$, thus satisfying the lemma.

However, if i is correct, then the following argument holds: From Action 4, we know that i sends the message $\langle \mathsf{msgReq}, s_i.\mathsf{seq}[j] \rangle$ to all other processes in the step that i takes immediately following C''. For all correct processes j, j receives the message

$\langle \mathsf{msgReq}, s_i.\mathsf{seq}[j] \rangle$ from i, executes Action 5, and sends $\langle \mathsf{msgResponse}, msgSet, num \rangle$ where $num = s_i.\mathsf{seq}[j]$. The message $\langle \mathsf{msgResponse}, msgSet, num \rangle$ is eventually received by i (i is still live) in Action 6 where i sets the value of $s_i.\mathsf{maxAck}[j]$ to num (which equals $s_i.\mathsf{seq}[j]$). This follows from Lemma VII.4.3 which shows that $s_i.\mathsf{maxAck}[j] \leq s_i.\mathsf{seq}[j]$, and $num = s_i.\mathsf{seq}[j]$; therefore, in line 27 of Action 6, $s_i.\mathsf{maxAck}[j]$ is updated to $num = s_i.\mathsf{seq}[j]$.

For all faulty processes j, either (1) process i eventually receives a message $\langle \mathsf{msgResponse}, msgSet, num \rangle$ from j where $num = s_i.\mathsf{seq}[j]$, and hence, eventually $s_i.\mathsf{maxAck}[j] = s_i.\mathsf{seq}[j]$, or (2) j crashes and by strong completeness, j is eventually and permanently suspected by the failure detector $\mathcal{D}$ (that is, $j \in \mathsf{suspectlist}$).

Since only finitely many $\mathsf{msgResponse}$ messages are received by i, the size of $\mathsf{receive\_buffer}[i, j]$ is finite for all processes j. Since the output action $\mathsf{receive}$ in Algorithm 15 enabled while $\mathsf{receive\_buffer}[i, j]$ is not empty, and every execution of $\mathsf{receive}$ decreases the size of $\mathsf{receive\_buffer}[i, j]$. Eventually, $\mathsf{receive\_buffer}[i, j]$ is empty.

Therefore, eventually for all processes $j \in \Pi - \{i\}$, either $s_i.\mathsf{maxAck}[j] = s_i.\mathsf{seq}[j]$ or $j \in \mathsf{suspectList}$, and $\mathsf{receive\_buffer}[i, j]$ is empty. That is, eventually, Action 7 is enabled at i.

Upon executing Action 7, $\mathsf{takeStepFlag}$ is set to *enabled*, thus enabling the output action $\mathsf{takeStep}$ in Algorithms 14, 15. By executing the output action $\mathsf{takeStep}$, the scheduled application takes a step and $\mathsf{takeStepFlag}$ is set to *completed*. Thus, Action 8 is enabled at i. Upon executing Action 8, i starts *waiting*. Thus shown that if a process i is *active* in state C at time t, then at some future state C' at time $t' > t$ either i is crashed or i is *waiting*. □

Corollary VII.4.5. *Let (t, t') be an interval in $\alpha|_{r_s}$ where process i is active at all*

times in (t, t'), and let i be waiting at times t and t'. There exists a time t'' in the interval (t, t') at which i executes the action takeStep *in Algorithms 14, 15 (and the scheduled application takes a step).*

Given that live processes are active only for finite durations, in order to prove progress, we need to show that every *waiting* process eventually becomes *active*. For this purpose, we introduce some definitions to construct a metric function on states in α. First, we measure the priority distance between any two processes i and j in a state as:

$$dist(i,j) = \begin{cases} 0, & \text{if } (s_i.\mathsf{ht} < s_j.\mathsf{ht}) \\ s_i.\mathsf{ht} - s_j.\mathsf{ht}. & \text{if } ((s_i.\mathsf{ht} \geq s_j.\mathsf{ht}) \wedge (i < j)) \\ s_i.\mathsf{ht} - s_j.\mathsf{ht} + 1, & \text{if } ((s_i.\mathsf{ht} \geq s_j.\mathsf{ht}) \wedge (i > j)) \end{cases}$$

Suppose for any pair of processes i and j that $dist(i,j) = d$ in some state where j is *waiting*. While j remains *waiting*, $s_j.\mathsf{ht}$ remains unchanged. Also, recall from Action 7 that each process reduces its height (below all the processes whose shared permits it holds) when exiting the *active* state. Consequently, d is an upper bound on the maximum number of times that process i can overtake process j and become *active* before either j becomes *active* or $s_i.\mathsf{ht} < s_j.\mathsf{ht}$. Now we define a metric function $M : \Pi \to \mathbb{N}$ for each process $j \in \Pi$ as follows:

$$M(j) = \sum_{\forall i \in \Pi : i \neq j} dist(i,j)$$

Note that M is bounded below by 0, and that $M(j) = 0$ if and only if j currently has the highest priority value among all processes in Π. In general, the value of $M(j)$ depends only on processes that are currently higher-priority than j. This is because $dist(i,j) = 0$ for any process i with lower height than i or equal height as j but lower process id. If $M(j) = b$, then b is an upper bound on how many times any

higher-priority process can become *active* before either j becomes *active* or j is the process with highest priority.

Also note that the metric value of each process in a given state is unique: $(i \neq j) \Rightarrow M(i) \neq M(j)$. Moreover, $M(i) < M(j) \Leftrightarrow ((s_i.\mathsf{ht} < s_j.\mathsf{ht}) \vee ((s_i.\mathsf{ht} = s_j.\mathsf{ht}) \wedge (i < j))$. These properties follow from the fact that priorities are totally ordered.

Finally,the metric value $M(j)$ never increases while process j is *waiting*. $M(j)$ can only increase by reducing the height $s_j.\mathsf{ht}$ in Action 7 while exiting the *active* state. Importantly, this change in relative priority actually causes the metric values of all other processes to decrease.

We are now prepared to state and prove the following helper lemma for progress:

Lemma VII.4.6. *Let C be any state in $\alpha|_{r_s}$ with at least one live* waiting *process. Let j be the live* waiting *process in C with minimal metric. Then there is a later state C' in α such that: (1) j is active in C', or (2) j is crashed in C', or (3) some other process i is live and* waiting *and $M(i) < M(j)$ in C'.*

Proof. Assume in contradiction that in every state after C, j is live and *waiting* and has the minimal metric. We will show that eventually j is *active*, a contradiction.

Let C'' be a state after C in $\alpha|_{r_s}$ in where all faulty processes have crashed and by strong completeness of $\mathcal{D}$, all such crashed processes are permanently suspected. After C'', j only needs to collect permits from correct processes. We show that j succeeds in collecting and keeping all these permits, and thus, j will become *active*.

Let i be any correct process other than j. First we show that j will not lose the permit it holds with i. By hypodissertation, j is *waiting* and has higher priority than any correct process from state C onwards (recall that $M(j)$ never increases while j is *waiting*; hence, j will continue to be the highest priority process until it becomes *active*), so any request token received by j in Action 2 will be deferred. Note that it is

possible for j to receive an 'old' request token from i which has higher priority value, thereby causing j to give up its shared permit. However, j will send the request token to i in Action 1 right after sending the permit, and this time i will have to return the permit to j because j has higher priority. Thereafter, eventually, j defers the request token from i until j becomes *active*.

Now we show that j will eventually acquire the permit shared with i. By Lemma VII.4.2, j shares a unique request token with i. All permits that were in transit to j when j started *waiting* are delivered in finite time. For any missing permits, if j holds the request token, then j will eventually send the corresponding token.

However, if j has neither the request token or the shared permit upon transiting to *waiting*, then eventually the shared permit and the request token are either at i or at j. We now show that eventually j receives either the request token or the shared permit. For the purposes of contradiction, suppose eventually i holds both the permit and the request token permanently. If i is *active*, then i eventually starts *waiting* (by Lemma VII.4.4) and sends the permit to j in Action 5. If i is *waiting*, then depending on the order in which the request token and the permit arrived at i, process i executes either Action 2 or Action 3. Since priorities are non-increasing, the priority encoded in the token and the permit received by i must be at least as high as j's current priority. We have already established that j has the highest priority in the system. Therefore, in both Action 2 and Action 3, i sends the shared permit to j.

If j (eventually) receives the request token, then j sends this request token to i in Action 1. Recall that by hypodissertation, j has higher priority than i; consequently, this permit request must be honored unless i is currently *active*. In the latter case, we know from Lemma VII.4.4 that i eventually exits to be *waiting*; therefore, the

requested permit will be sent when i starts *waiting* in Action 5.

Thus, we conclude that if j remains *waiting* indefinitely, then j eventually suspects each faulty process and eventually holds the shared permit with each correct process. This enables the guard on Action 4, and eventually, j becomes *active*. □

Theorem VII.4.7. *Algorithm 16 satisfies local progress. That is, every correct process takes infinitely many application steps.*

Proof. Note that to prove the theorem, it is sufficient to prove the following claim: For every k, every state C of $\alpha|_{r_s}$, and every correct process j, if $M(j) = k$ in C, then there is a later state in which j is *active*. We prove this by a complete (strong) induction on metric values.

Base Case: $k = 0$. Suppose $M(j) = 0$ in state C. Since 0 is the smallest possible value that the metric can have and j is correct, Lemma VII.4.6 implies that in some subsequent state C', either j is *active* or there is another live *waiting* process i whose metric is smaller than j's metric in C'.

However, since a j's metric can never increase while j is *waiting*, and it is not possible for a process to have a metric less than 0, no such live *waiting* process i exists. So, j eventually becomes *active*.

Inductive Case: $k > 0$. Suppose for every $k' < k$, every state C of $\alpha|_{r_s}$, and every correct process j, if $M(j) = k'$ in C, then there is a later state in which j is *active*. We must show that for every state C and every correct process j, if $M(j) = k$ in C, then there is a later state in which j is *active*.

Let C be a state, and let j be a correct *waiting* process in C with $M(j) = k$. Suppose that k is the minimal metric value among all correct *waiting* processes in C. Then Lemma VII.4.6 applies to j, so we conclude that j eventually becomes *active*, or some correct process i with $M(i) < M(j)$ starts *waiting*. Alternatively, suppose

that k is not the minimal metric value among all correct *waiting* processes in C. Then some correct *waiting* process i with $M(i) < k$ already exists. Either way, we conclude that j eventually becomes *active* or the inductive hypodissertation applies to some correct *waiting* process i with $M(i) < k$. In the latter case, process i becomes *active*. By Lemma VII.4.4, i eventually exits the *active* state by executing Action 5, which thereby lowers the height $s_i.\mathsf{ht}$ and decreases $dist(i, j)$ by at least 1. Recall that while j remains *waiting*, $M(j)$ does not increase. Thus, any decrease in $dist(i, j)$ will cause the metric value of $M(j)$ to become less than k. Since j is now a correct *waiting* process with $M(j) < k$, the inductive hypodissertation applies directly to j. Thus, we conclude that j eventually becomes *active*.

By Lemma VII.4.4, Corollary VII.4.5, and Actions 7 and 8, we know that every time j becomes *active*, it executes an application action, and j eventually exits. Upon exiting j starts *waiting* again. Thus, we show that Algorithm 16 satisfies local progress by complete induction. □

To establish the proof for computational fairness, we make use of the notion of *distinguished* processes. Recall that a distinguished process is never suspected until it crashes and is suspected forever thereafter.

Theorem VII.4.8. *Every (eventually) distinguished process is (eventually)* 2*-proc-fair.*[3]

Proof. Consider an arbitrary execution α and let i be any process that is eventually distinguished in the run $\alpha|_{r_s}$ starting at some time t_i. We must show that for all j,

[3] *If all processes are distinguished, then a careful analysis shows that all correct distinguished processes are, in fact,* 1*-proc-fair. We omit this here because, for the purposes of our results, it is sufficient to show that a correct distinguished process is* 2*-proc-fair.*

in every interval starting after t_i in which j takes 3 application steps, either i takes at least 1 application step or i is crashed.

Consider a process $j \neq i$ that takes 3 steps after t_i, say at times t, t', and t'' and suppose that i is live through t''. We must show that i takes a step at least once between t and t''. Since i is distinguished after t_i, the failure detector $\mathcal{D}$ at j never suspects i between t_i and t''.

At time t, let the height of i be ht_i and the height of j be ht_j. Let $ht_j > ht_i$. From Action 8, we know that j sends the permit to i with height ht_j encoded in the permit message. We also know that j takes an application step at time t' and therefore is *active* at time t'. Therefore, j must hold the permit it shares with i at time t'. That is, i sends the permit to j in the interval (t, t'). Note that i sends the permit to j only in actions 2 and 8. If i executes Action 8 in (t, t'), it implies that i was *active* in the interval (t, t'), and hence took an application step in that interval; thus, the lemma is satisfied.

On the other hand, if i sends the permit to j in Action 2 (and noting that i is not *active* in the interval (t, t')), then i encodes its height ht_i in the permit. Therefore, when j is *active* at time t', the value of $s_j.\mathsf{ht}_i$ is ht_i. When j transits to *waiting*, it reduces its height to $ht_{j'} < ht_i$ and encodes this height in the permit sent to i.

We know that j takes an application step again at time t'', and so j is *active* again at time t''. Therefore, j must hold the permit it shares with i at time t''. That is, i sends the permit to j in the interval (t', t''). Note that in this interval i does not send the permit to j in Action 2. We conclude this based on the heights of i and j. When j sends a request token to i in (t', t''), it is encoded with height $ht_{j'} < ht_i$. Therefore, when i executes Action 2 upon receiving this request token, i notes that j's height is lower than its own height, and so does not send the permit to j. Therefore, j could have received the permit from i only through Action 8 (executed by i); that

is, i must have been *active* at some time in the interval (t', t''), and hence, i must have taken an application step in the interval (t', t''). That is, from time t_i onwards, i is 2-proc-fair.

In other words, i is eventually 2-proc-fair. However, if $t_i = 0$, then i is 2-proc-fair in the execution α; that is, if i is distinguished process, then i is 2-proc-fair.

Thus shown that every (eventually) distinguished process is (eventually) 2-proc-fair. □

Theorem VII.4.9. *Every (eventually) distinguished process is (eventually)* 1*-com-fair*

Proof. Consider an arbitrary execution α and let i be any process that is eventually distinguished in the run $\alpha|_{r_s}$ starting at some time t_i. We show that for all processes j and all application messages m sent from i and received by j after t_i, during the time m is in transit, either j takes at most one step or i is crashed.

Consider a process j to which i sends an application message m at some time t after t_i. This message is sent by the application when i is *active* and executes its output action send (m,i,j), which causes m to be added to send_buffer[i,j] (the message is actually sent by the model automaton later during the execution). By the assumption that m is received by j, we know that j takes at least one application step after t. Again, note that j executes its application step only when j is *active*. Let the earliest time after t that j is *active* be t'. In the *active* session that starts at time t', j sends $\langle \mathsf{msgReq}, s_j.\mathsf{seq}[j] \rangle$ to i in Action 4.

From Lemma VII.4.3 we know that $s_j.\mathsf{maxAck}[i] \leq s_j.\mathsf{seq}[i]$ before j executes Action 4. But Action 4 increments $s_j.\mathsf{seq}[i]$, therefore, after j executes Action 4, $s_j.\mathsf{maxAck}[i] < s_j.\mathsf{seq}[i]$

From Lemma VII.4.4, we know that j eventually stops being *active*. Since i is

a correct eventually distinguished process, we also know that j does not suspect i. Therefore, if j eventually exits the *active* state by executing Action 8, then eventually $s_j.\mathsf{maxAck}[i] = s_j.\mathsf{seq}[i]$.

The above two arguments imply that while j is , *active*, the value of $s_j.\mathsf{maxAck}[i]$ is updated to $s_j.\mathsf{seq}[i]$. However, the only action that updates $s_j.\mathsf{maxAck}[i]$ is Action 6, and Action 6 is executes only upon receiving $\langle \mathsf{msgResponse}, msgSet', num \rangle$.

While the message $\langle \mathsf{msgReq}, s_j.\mathsf{seq}[i] \rangle$ sent by j in action 4 is eventually received by i in Action 5 (or i is crashed, in which case the lemma is satisfied). Action 5 empties $\mathsf{send\_buffer}[i, j]$ and sends the messages in the buffer to j. But note that the message m was in $\mathsf{send\_buffer}[i, j]$ before Action 5 is executed. Therefore, message m is sent to j in the message $\langle \mathsf{msgResponse}, msgSet', s_j.\mathsf{seq}[i] \rangle$. This message is eventually received by j in Action 6, and Action 6 puts message m into the receive buffer $\mathsf{receive\_buffer}[j, i]$ and updates $s_j.\mathsf{maxAck}[i]$ to $s_i.\mathsf{seq}[j]$.

While $\mathsf{receive\_buffer}[j, i]$ has messages in it, the action $\mathsf{receive}$ in Algorithm 15 is enabled at j which delivers the messages from $\mathsf{receive\_buffer}[j, i]$ to the scheduled application.

Therefore, when j executes the output action $\mathsf{takeStep}$ in Algorithms 15 , m which is in $\mathsf{receive\_buffer}[j, i]$, is received by the scheduled application protocol at j. That is, j takes no more than 1 step after m is sent and before m is received. In other words, i is 1-com-fair after time t_i. However, if $t_i = 0$, then i is 1-com-fair in the execution α; that is, if i is an (eventually) distinguished process, then i is (eventually) 1-com-fair. □

Corollary VII.4.10. *If the failure detector $\mathcal{D}$ in Algorithm 16 is the perfect failure detector $\mathcal{P}$, then the action system described in Algorithm 16 implements the All Fair ($\mathcal{AF}$) system model.*

Proof. Given that $\mathcal{P}$ is a failure detector for which every process is distinguished, from Theorems VII.4.8 and VII.4.9 we know that all processes are 2-proc-fair and 1-com-fair. By definition, this is $\mathcal{AF}$. □

Corollary VII.4.11. *If the failure detector $\mathcal{D}$ in Algorithm 16 is the eventually perfect failure detector $\Diamond\mathcal{P}$, then the action system described in Algorithm 16 implements the Eventual All Fair ($\Diamond\mathcal{AF}$) system model.*

Proof. Recall that $\Diamond\mathcal{P}$ is a failure detector for which every process is eventually distinguished. Let the time after which every process is distinguished be t. From Theorems VII.4.8 and VII.4.9 we know that all distinguished processes are 2-proc-fair and 1-com-fair after time t. Since every process is distinguished after time t, it follows that, eventually, all processes are 2-proc-fair and 1-com-fair. By definition, this is $\Diamond\mathcal{AF}$. □

Corollary VII.4.12. *If the failure detector $\mathcal{D}$ in Algorithm 16 is the strong failure detector $\mathcal{S}$, then the action system described in Algorithm 16 implements the Some Fair ($\mathcal{SF}$) system model.*

Proof. Given that $\mathcal{S}$ is a failure detector for which some correct process is distinguished, from Theorems VII.4.8 and VII.4.9 we know that the correct distinguished process is 2-proc-fair and 1-com-fair. By definition, this is $\mathcal{SF}$. □

Corollary VII.4.13. *If the failure detector $\mathcal{D}$ in Algorithm 16 is the eventually strong failure detector $\Diamond\mathcal{S}$, then the action system described in Algorithm 16 implements the Eventually Some Fair ($\Diamond\mathcal{SF}$) system model.*

Proof. Note that $\Diamond\mathcal{S}$ is a failure detector for which, eventually, some correct process is distinguished. Let the time after which some correct process (say, i) is distinguished

be t. From Theorems VII.4.8 and VII.4.9 we know that i is 2-proc-fair and 1-com-fair. By definition, this is $\Diamond\mathcal{SF}$. □

Recall from Section III.2.1 that the Eventual All Fair ($\Diamond\mathcal{AF}$) system model is the same as the $\mathcal{M}_*$ system models (assuming that $\mathcal{M}_*$ specifications are fairness based, and not real-time based). Therefore, from Theorem VII.3.5 and Corollary VII.4.11, we know that $\Diamond\mathcal{P}$ is the weakest failure detector to construct the $\mathcal{M}_*$ system models.

CHAPTER VIII

IMPLEMENTING $\Diamond\mathcal{P}$ ON EMPIRICAL SYSTEMS

If a man will begin with certainties, he shall end in doubts; but if he will be content to begin with doubts he shall end in certainties.

The Advancement of Learning, Book 1, 1605
– *Francis Bacon*

Recall from Chapter IV that the construction of $\mathcal{M}_*$ from empirical systems follows three steps. The first step is to construct an appropriate failure detector on top of empirical systems; the second step is to implement reliable communication on top of empirical systems; and the third step is to construct $\mathcal{M}_*$ using the thus constructed failure detector and reliable channels. The second step has been accomplished in [1–3, 11, 37], and the third step was accomplished in Chapter VII. Also, Chapter VII determined that the appropriate failure detector for step one is $\Diamond\mathcal{P}$. Thus, the only remaining step is the construction of $\Diamond\mathcal{P}$ on empirical systems. We accomplish this task in this chapter.[1]

Implementing $\Diamond\mathcal{P}$ on empirical systems presents multiple challenges. While many such challenges have been overcome in previous implementations of $\Diamond\mathcal{P}$, two salient challenges remain unresolved: (1) tolerating infinite message loss, and (2) the celeration gap. We discuss these two challenges next.

VIII.1. Tolerating infinite message loss

Tolerating infinite message loss is critical to implementing $\Diamond\mathcal{P}$ on empirical systems simply because the ADD channels in empirical systems can lose an infinite subset

[1]The *celeration gap* and *bichronal time* described in this chapter have been published in [75].

of messages. The classic and many customized models of partial synchrony [33, 35, 40, 50, 73] make relatively strong assumptions about communication reliability. To our knowledge, among the partially synchronous system models sufficiently powerful to implement $\Diamond\mathcal{P}$, ADD channels provide weakest guarantees on communication reliability.

The models cited above assume that the communication channels are either always reliable, or eventually reliable (i.e., can lose at most finitely many messages over some prefix, followed by an infinite reliable suffix). Implementations of $\Diamond\mathcal{P}$ in models with such strong assumptions on communication reliability can be trivially adapted to withstand certain subsets of messages being arbitrarily delayed or dropped. For instance, consider a system E where only the odd-numbered messages may be delayed or dropped, but all the even-numbered messages are delivered reliably within some unknown bound on delay. Implementing $\Diamond\mathcal{P}$ in such a system is trivial, because the infinite pattern of potentially delayed and dropped messages in known, and hence can be used to advantage. Such applications can simply send dummy information in the odd-numbered messages and use the even-numbered messages to communicate. Effectively, the applications have access to a reliable sub-channel consisting of the even-numbered messages. However, ADD channels in the empirical system model may be viewed as links where, during every prefix of every computation, (say) at most a fraction f of the messages sent may be delayed or dropped, but all other messages are delivered within some (unknown) bound on delay. Implementing $\Diamond\mathcal{P}$ in such a system becomes non-trivial.

We resolve the issues with infinite message loss by imposing a 'bounded persistence' to the messages sent among processes. Informally, the sequence of messages sent from process (say) i to process (say) j is bounded persistent if there exists an upper bound on the real-time duration between times that two messages are sent

consecutively. The intuition is that if messages are sent with bounded persistence, then they are received with bounded persistence; that is, there exists an upper bound on the real-time duration between times that two messages are received consecutively. This roughly translates to an upper bound on the end-to-end communication delay of certain subset of messages (specifically the set of privileged messages) in the system.

While this approach works well in system models where there is either an upper bound on absolute process speeds or a lower bound on absolute process speeds, it fails in the empirical system model where absolute process speeds are unbounded (above and below). This is due to a phenomenon called the *celeration gap*. We discuss the celeration gap and a mechanism to bridge the gap in Section VIII.2 that follows.

VIII.2. Process celeration

There are several implementations of $\Diamond\mathcal{P}$ in various real-time based models of partial synchrony in the existing literature [4,14,20,39,58,59,62]. These implementations are based on the deduction that upper bounds on relative process speeds and message delay translate to an (unknown) upper bound on end-to-end communication delay. The end-to-end communication delay is the duration between the send of a message and its receipt. However, these implementations have overlooked an important subtlety with respect to measuring the passage of time in *celerating executions*, wherein absolute process speeds can continually increase and/or decrease while maintaining bounds on relative process speeds. Such traditional implementations of $\Diamond\mathcal{P}$ can precipitate an infinite number of failure detector mistakes in celerating executions.

In system models with bounded message delay and bounded relative process speeds, such infinite failure detector mistakes happen because end-to-end message delay, which is bounded either in real time, or in the number of steps executed by pro-

cesses, in executions where absolute process speeds are bounded, become unbounded in both real time and in the number of steps executed by processes when absolute process speeds are not bounded. We call this *the celeration gap*.

In order to understand the celeration gap, we first describe how end-to-end message delay is bounded in non-celerating executions (that is, execution where absolute process speed is bounded, both above and below) where there exists a real-time upper bound on message delay and an upper bound on relative process speeds. Then we show how the end-to-end message delay can be unbounded in celerating executions.

VIII.2.1. End-to-end message delay in non-celerating executions

In non-celerating environments, the process speeds, by definition, do not accelerate or decelerate continually. In other words, there exist some (potentially unknown) upper and lower bounds on absolute process speeds. Given such bounds on absolute process speeds, in conjunction with an upper bound on message delay, demonstrating an upper bound on end-to-end communication delay is straightforward.

End-to-end communication delay can be measured either as the number of steps executed by the recipient or as the passage of real time. Action clocks (or step timers) measure the former whereas real-time clocks measure the latter.

VIII.2.1.1. Action clocks

Action clocks increment their clock value by one every time the protocol executes its program action. Since we know that there exists an upper bound on absolute process speeds in non-celerating environments, there exists an upper bound on the number of program actions a control protocol can execute while a message is in transit (note that there exists an upper bound on message delay). Additionally, sending and receiving a message is assumed to be atomic and hence requires exactly one program action

each. Since there exists an upper bound on relative process speeds, there exists an upper bound on the number of program actions executed by control protocols at *each* process while a message is either being sent or received. In other words, there exists an upper bound on the number of action clock ticks at the control protocol while a message is being sent, is in transit, and is being received. That is, there exists an upper bound on end-to-end message delay as measured by the action clock.

VIII.2.1.2. Real-time clocks

A lower bound on absolute process speeds in non-celerating environments implies an upper bound on the real-time duration for a message to be generated and sent, as well as an upper bound on the real-time duration to complete the receipt of the message. Since the upper bound on message delay is assumed, there exists an upper bound on end-to-end communication delay, as measured by real-time clocks.

VIII.2.2. End-to-end message delay in celerating executions

In this section we describe the following:

- In accelerating executions, where absolute process speed is not bounded above, the end-to-end communication delay of messages is unbounded when denominated in ticks of an action clock.

- In decelerating executions, where absolute process speed is not bounded below by a positive finite value, the end-to-end communication delay of messages is unbounded when denominated in duration of a real-time clock.

VIII.2.2.1. Action clocks in accelerating executions

In accelerating executions, the end-to-end communication delay is unbounded when denominated in ticks of an action clock. The argument for this behavior is as follows.

Consider an accelerating execution. Let time be measured locally by action clocks. For the purpose of contradiction, assume that there exists an upper bound on message delay as measured by an action clock. Let such a bound be k. Let the bound on message delay be Δ real-time units. Since processes are continually accelerating, eventually (say, after real time t) process speeds exceed $\lceil \frac{k}{\Delta} \rceil$ program actions per unit real time. Let some message m sent after time t experience a delay of Δ real-time units. The message delay for m measured in action clock ticks exceeds k (because $\lceil \frac{k}{\Delta} \rceil \cdot \Delta \geq k$). However, this contradicts our assumption that the upper bound on message delay as measured by an action clock does not exceed k. This implies that message delay measured in action clock ticks is unbounded. Therefore, in accelerating environments, the end-to-end communication delay of messages is unbounded when denominated in ticks of an action clock.

VIII.2.2.2. Real-time clocks in decelerating executions

In decelerating executions, the end-to-end communication delay is unbounded when denominated in real-time duration. The argument for such behavior is as follows:

Consider a decelerating execution. Let time be measured locally by perfect real-time clocks[2]. For the purpose of contradiction, we assume that there exists an upper bound on real-time end-to-end communication delay. Let this bound be k real-time units. As processes decelerate, an increasingly greater duration of time elapses be-

[2]Although it is not necessary for the real-time clocks to be perfect, we strengthen the clock specification in order to strengthen the negative result

tween consecutive control-protocol program actions. Eventually (say, after time t), the time between consecutive control-protocol program actions exceeds k. Since it requires at least one program action to send or receive a message, after time t, generation of a message m takes longer than k real-time units. Consequently, the end-to-end communication delay for m exceeds k. This, however, contradicts our earlier assumption that end-to-end communication delay is bounded above by k. In other words, in decelerating environments, the end-to-end communication delay of a message is unbounded when denominated in real-time duration.

VIII.2.3. The celeration gap

Celerating executions denote the set of executions where processes accelerate and/or decelerate continually. Hence, absolute process speeds for (live) processes may be unbounded (above and below) in such executions. From Section VIII.2.2.1, we know that end-to-end communication delay when measured by action clocks is unbounded in executions where processes accelerate continually. However, in exclusively accelerating executions, the end-to-end communication delay remains bounded above when measured by a real-time clock (with bounded drift rate). Similarly, we know from Section VIII.2.2.2 that in executions where processes decelerate continually, end-to-end communication delay is unbounded when measured by a local real-time clock (with bounded drift rate). However, in exclusively decelerating executions, the end-to-end communication delay remains bounded when measured by action clocks.

On the other hand, in executions where processes may accelerate and decelerate continually, there exists neither an upper bound, nor a lower bound, on absolute process speeds. Consequently, the end-to-end communication delay is unbounded in terms of both real-time clocks and action clocks. Such unbounded end-to-end communication delay in terms of either real-time clocks, or action clocks, or both, in

celerating executions is referred to as the *celeration gap*.

VIII.2.4. Impact of the celeration gap

In this subsection, we discuss the impact of the celeration gap on designing a $\Diamond\mathcal{P}$ failure detector implementation in the empirical system model.

VIII.2.4.1. Existing $\Diamond\mathcal{P}$ implementations

Recall from Section VIII.2 that the existing implementations of failure detectors like $\Diamond\mathcal{P}$ in real-time based partially synchronous systems adaptively estimate the upper bound on end-to-end communication delay. However, since end-to-end communication delay is unbounded in terms of both action clock ticks and real-time duration, all such implementations fail in celerating executions.

VIII.2.4.2. Bridging the celeration gap

We introduce a new technique to bridge the celeration gap without assuming lower or upper bounds on absolute process speeds. Our solution is based on a composition of action clocks and real-time clocks and is motivated by the following observations: Message delay is bounded above in terms of real time. Therefore, there exists an upper bound on the number of ticks of a real-time clock while a message is in transit. Similarly, relative process speeds are bounded as well. Therefore, regardless of process celeration, there exists an upper bound on the number of program actions executed by the control protocol while messages that have been delivered at the recipient's receive buffer are being processed. In other words, there exists an upper bound on the number of action clock ticks in the duration it takes for a message to be processed. Therefore, running real-time clock timers when messages are in transit, and running action clock timers when messages are being processed, should make $\Diamond\mathcal{P}$ implementations immune

to the celeration gap. We explore this intuition by introducing a new clock called a *bichronal clock*.

VIII.3. Bichronal clocks

A bichronal clock is a composition of an action clock and a real-time clock. It has the following properties:

- **Composition:** A bichronal clock consists of an action clock a and a real-time clock r.
- **Two-dimensional Time:** The time on a bichronal clock is the vector $\langle a.time, r.time\rangle$ where $a.time$ is the value of the action clock, and $r.time$ the value of the real-time clock. These time components are independent and will not be ordered lexicographically.

Similarly, a bichronal timer consists of an action clock timer and a real-time clock timer. It counts down from a given value (a_t, r_t) where a_t is the starting value for the action clock timer, and r_t is the starting value for the real-time clock timer. A bichronal timer is said to have timed out iff both the action clock timer and the real-time clock timer have timed out.

VIII.4. Bichronal clocks as a part of the control protocol

Before proceeding to the $\Diamond\mathcal{P}$ implementation within the control protocols in empirical systems, we first need to include bichronal timers in the TIOA specification for the control protocol in each process. The TIOA framework for the control protocol in empirical systems is given in Algorithms 17–18. In addition to the specifications and descriptions of control protocol in Section V.3.2.1, we assume that the control protocol has access to a real-time clock (labeled clock) with a bound D on drift rate (as

specified in Section V.5.5 and as defined in the trajectory function in Algorithm 18). Bichronal timers, described next, are constructed from a real-time timer that is implemented using clock and an action timer constructed by augmenting action takeStep as described in the paragraphs that follow.

The automaton specified in Algorithms 17–18 uses four variables to implement a bichronal timer: actionTimer, realTimeTimer, realTimeTimerExpired, and bichronalTimerExpired. The bichronal timer is initialized by the program action by executing the macro startBichronalTimer(a,r) (defined in Algorithm 17) for a actions and r clock ticks. The macro initializes actionTimer to a, realTimeTimer to clock $+ r$, realTimeTimerExpired to $false$, and bichronalTimerExpired to $false$. The values of these variable change as follows.

The value of actionTimer is decremented by 1 every time the program action is executed in takeStep (in Algorithm 18) until actionTimer is 0. When actionTimer decrements to 0, the action timer is said to have expired. Thus, action timer expires after the control protocol executes its program action r times after the timer is (re)started.

In contrast, the value of realTimeTimer remains unchanged until startBichronalTimer(a,r) is executed again. However, the control protocol executes action realTimeTimerExpiry when clock, which evolves with a bound D on the drift with respect to real time, equals realTimeTimer; the action sets realTimeTimerExpired to $true$.

When both the real-time timer and the action timer expire, the bichronal timer also, by definition, expires. This is modeled by the internal action bichronalTimerExpiry which sets bichronalTimerExpired to $true$. To ensure that timer expiries happen at the instant that the deadlines are reached, the trajectory function in Algorithm 18 ensures that time evolution is paused when clock equals realTimeTimer and action realTimeTimerExpiry is not yet executed. Similarly, time is evolution is paused when both

Algorithm 17 Signature and states for the TIOA specification for the control protocol in empirical systems with a bichronal timer. The state transitions for the TIOA are specified in Algorithm 18.

automaton controlProtocol(i: pIndex) *//Control Protocol with a Bichronal Timer*
type pIndex = **enumeration of** p_1,p_2,...,p_n **where** $\Pi = \{p_1, p_2, \ldots, p_n\}$
type rIndex = **enumeration of** r_0, r_1, r_2,...,r_m **where** $\rho = \{r_0, r_1, r_2, \ldots, r_m\}$
and r_0 refers to the control protocol itself
signature (**in** addition to the actions specified **in** Algorithms 1 and 11)
input takeStep(i : pIndex) *//Enables process i to take a step. Redefined here.*
internal realTimeTimerExpiry() *//Signals expiry of the real-time timer*
internal bichronalTimerExpiry() *//Signals expiry of the bichronal timer*
states
now: Real ← 0 *//Current time. As defined in Algorithm 1*
rSendBuff: Array[rIndex, pIndex, Queue[Type]] ← constant(∅)
//Messages sent by protocols. As defined in Algorithm 1
rReceiveBuff: Array[rIndex, pIndex, Queue[Type]] ← constant(∅)
//Messages received for protocols. As defined in Algorithm 1
pause: Array[pIndex, Boolean] ← constant(false)
//Device to stop time evolution.As defined in Algorithm 1
queryInProgress: Array[rIndex, Boolean] ← constant(false)
//As defined in Algorithm 11
suspectList : Set[pIndex] ← ∅, *//Suspect List. As defined in Algorithm 11*
actionTimer: Integer *//Action timer value for the bichronal timer*
realTimeTimer: Real *//Real-time Timer value for the bichronal timer*
realTimeTimerExpired: Boolean ← true *//Denotes if the real-time Timer is expired*
bichronalTimerExpired: Boolean ← true *//Denotes if the bichronal Timer is expired*
clock: Real ← 0 *//Local clock value*
fd_i.aValue: Integer ← fixed constant a *//Action timer value; used by Algorithm 19*
fd_i.rValue: Integer ← fixed constant r *//Real-time timer value; used by Algorithm 19*
fd_i.hbBound: Array[pIndex, Integer] ← constant(1)
//Estimate on number of messages sent; used by Algorithm 19
fd_i.hbCount: Array[pIndex, Integer] ← constant(1)
//Counts down number of messages sent; used by Algorithm 19
aliases
sendBuffer[j] ≡ rSendBuff[r_0,j]
receiveBuffer [j] ≡ rReceiveBuff[r_0,j]
startBichronalTimer(a, r) ≡ {actionTimer ← a;
realTimeTimer ← clock + r;
realTimeTimerExpired ← false;
bichronalTimerExpired ← false}

Algorithm 18 Transitions for the TIOA Specification of the control protocol in empirical systems with a bichronal timer. The signature and states of the TIOA is specified in Algorithm 17.

input takeStep(i)
effect
execute an enabled program action **from** Algorithm 19
actionTimer $\leftarrow \min(\text{actionTimer} - 1, 0)$
$\forall$j$\in$ pIndex::pause[j] $\leftarrow$ true
internal realTimeTimerExpiry()
precondition
realTimeTimer = clock
effect
realTimeTimerExpired $\leftarrow$ true
internal bichronalTimerExpiry() *//Bichronal timer expires only when both*
precondition *//action timer and real-time timer expire*
(realTimeTimerExpired) $\wedge$ (actionTimer = 0) $\wedge$ ($\neg$bichronalTimerExpired)
effect
bichronalTimerExpired $\leftarrow$ true

trajectories
stop when
($\exists$j$\in$ pIndex::pause[j] = true) $\vee$ *//Pause time to send msgs*
(realTimeTimer = clock)$\wedge$($\neg$realTimeTimerExpired)) $\vee$
(realTimeTimerExpired $\wedge$ (actionTimer = 0) $\wedge$ ($\neg$bichronalTimerExpired))
evolve *//Pause time to flag timers' expiry*
d(now) = 1
D $\leq$ d(clock) $\leq$ 1/D *//Local clock has a bound D on drift rate*

the real-time timer and the action timer are expired until the action bichronalTimer-Expiry is executed. The program action verifies bichronal timer expiry by checking the value of bichronalTimerExpired.

Note that in the control-protocol framework, we have specified just one bichronal timer. However, it is permissible to have multiple timers. The only reason we have specified just one is that one bichronal timer is sufficient to implement $\Diamond\mathcal{P}$ in empirical systems.

VIII.5. Implementing $\Diamond\mathcal{P}$

In this section, we present an implementation of $\Diamond\mathcal{P}$ on the empirical system model (from Section V.5.5) using bichronal clocks. Let the set of processes in the system be Π. The action system in Algorithm 19 implements $\Diamond\mathcal{P}$ when executed as the program action of the control protocol at each process i in Π.

Algorithm 19 Program action of $\Diamond\mathcal{P}$ implementation at process i.

```
1 :  { true } ⟶                                                          Action 1
2 :   foreach j ∈ Π − {i}
3 :     dequeue m from receiveBuffer[j]                    //Receive at most one message
4 :     if (m = ⟨HB⟩)                                    //Upon receiving a heartbeat from j
5 :       if (j ∈ suspectList)                                  //Possible false suspicion
6 :         fd_i.hbBound[j] ← fd_i.hbBound[j] +1      //Increment bound on heartbeats sent
7 :       suspectList ← suspectList{j}                          //Take j off the suspect list
8 :       fd_i.hbCount[j] ← fd_i.hbBound[j]                     //Start counting down again
9 :  if (bichronalTimerExpired)                                        //Upon timer expiry
10 :   foreach j ∈ Π − {i}
11 :     enqueue ⟨HB⟩ in sendBuffer[j]                  //Send heartbeats to all processes
12 :     fd_i.hbCount[j] ← min(fd_i.hbCount[j] − 1, 0)
13 :     if (fd_i.hbCount[j] = 0) suspectList ← suspectList ∪ {j}                //Suspect j
14 :   startBichronalTimer(fd_i.aValue, fd_i.rValue)                  //Start bichronal timer
```

VIII.5.1. Algorithm description

The action system uses the additional state variables described next (and defined in Algorithm 17). In order to denote their 'internal' scope (that is, they are used exclusively by the program action and no other action in the system), these variables are prefixed with fd_i. where i denotes the process whose control-protocol program action uses them. The variables are: $\mathsf{fd}_i.\mathsf{aValue}$, $\mathsf{fd}_i.\mathsf{rValue}$, $\mathsf{fd}_i.\mathsf{hbBound}$, and $\mathsf{fd}_i.\mathsf{hbCount}$.

The variables $\mathsf{fd}_i.\mathsf{aValue}$ and $\mathsf{fd}_i.\mathsf{rValue}$ are set to some fixed values a and r, respectively. These variables are used to (re)start bichronal timers repeatedly so that each process i sends heartbeat messages to all other processes in the system every $(\mathsf{fd}_i.\mathsf{aValue}, \mathsf{fd}_i.\mathsf{rValue})$ bichronal time units. Each element $\mathsf{fd}_i.\mathsf{hbBound}[j]$ (in the array $\mathsf{fd}_i.\mathsf{hbBound}$) stores the estimate on the highest number of heartbeats that i sends to j without having received a heartbeat from j. The information about the actual number of heartbeats sent by i to j without having received a heartbeat from j is stored in the element $\mathsf{fd}_i.\mathsf{hbCount}[j]$ (in the array $\mathsf{fd}_i.\mathsf{hbCount}$). In fact, $\mathsf{fd}_i.\mathsf{hbCount}[j]$ counts down from $\mathsf{fd}_i.\mathsf{hbBound}[j]$, and if $\mathsf{fd}_i.\mathsf{hbCount}[j]$ equals 0, then that is the basis for i to suspect j.

The action system in Algorithm 19 implements $\Diamond\mathcal{P}$ as follows. It consists of a single guarded common *Action 1* that is always enabled (that is, the guard is *true*), and it consists of two phases. The first phase consists of lines 2–8 in Algorithm 19, and the second phase consists of lines 9–14 in Algorithm 19. In the first phase at (say) process i, process i receives at most one heartbeat from each process j in the system (in line 3, Algorithm 19). All processes from who messages have been received are taken off the suspect list (line 7, Algorithm 19). For each process y processes that was suspected prior to receiving a heartbeat from j, the action increments the value of $\mathsf{fd}_i.\mathsf{hbBound}[j]$. Then it resets $\mathsf{fd}_i.\mathsf{hbCount}[j]$, the countdown of number of heartbeats

to j sent prior to receiving a heartbeat from j, to $\mathsf{fd}_i.\mathsf{hbBound}[j]$.

In the second phase is executed only if the bichronal timer is expired. Note from Algorithm 17 that initially bichronalTimerExpired is *true*; that is, the bichronal timer is expired. If the bichronal timer is expired, then for every other process j in the system: i sends a heartbeat to j (line 11, Algorithm 19), counts down of number of heartbeats to j sent prior to receiving a heartbeat from j by decrementing $\mathsf{fd}_i.\mathsf{hbCount}$ by 1 (line 12, Algorithm 19), and adds j to the suspect list if $\mathsf{fd}_i.\mathsf{hbCount}$ is equals 0 (line 13, Algorithm 19). Finally, i restarts the bichronal timer with action timer value $\mathsf{fd}_i.\mathsf{aValue}$ and real-time timer value $\mathsf{fd}_i.\mathsf{rValue}$ (line 14, Algorithm 19).

VIII.5.2. Proof of correctness

To prove correctness, we need to show that the implementation in Algorithms 17–19 satisfies $\Diamond\mathcal{P}$ specifications *viz.*, *strong completeness* and *eventual strong accuracy*, when executed as the control protocol in the empirical system model. Recall that strong completeness specifies that every crashed process be eventually and permanently suspected by all correct processes, and eventual strong accuracy specifies that every correct process be eventually and permanently trusted by all correct processes.

First, we revisit the properties of the empirical system model to establish the intuition behind the correctness of the $\Diamond\mathcal{P}$ implementation. We then follow the intuition up with formal arguments to establish correctness.

VIII.5.2.1. Empirical system model: a refresher

Recall from Section V.5.5 that the ratio of process speeds is always bounded above; that is, there exists an unknown upper bound Φ on relative process speeds. That is, every process that is not crashed executes its control-protocol program action at least once in a duration where all other processes (that have not crashed) execute their

control-protocol program action $\Phi + 1$ times. See Section V.5.1 for details.

Also, recall from Section V.5.5 that the communication links between every pair of processes are ADD links. That is, there exist unknown $\Delta \in \mathbb{N}^+$ and unknown $R \in \mathbb{N}^+$ such that for every interval of time in which process i sends exactly $R + 1$ messages at least one message (called *privileged* message) is delivered reliably at j with delay not exceeding Δ time units. See Section V.5.5 for details.

The constants Φ, Δ, R, and D (upper bound on the clock drift rate) can be used to establish correctness as follows. For strong completeness, if a process j crashes, then the $\Diamond\mathcal{P}$-module action system at j stops sending heartbeats to each correct process i. Therefore, i eventually stops receiving heartbeats while the bichronal timer at i expires (and is restarted) infinitely often. Therefore, $\mathsf{fd}_i.\mathsf{hbCount}[j]$ eventually counts down to 0, j is suspected by i, and i never receives another message from j to take j off the suspect list. That is, every crashed process is eventually and permanently suspected.

Eventual strong accuracy is based on the assertion that bounds Φ and D impose an upper bound on the number of heartbeats that a correct process i sends to a correct process j in the duration that j sends two heartbeats to i. Similarly, the bounds Δ and D impose an upper bound on the number of heartbeats that i sends to j while a privileged message from j is in transit to i. Since at least one message in every $R+1$ consecutive messages is guaranteed to be privileged, the bounds Φ, Δ, R, and D impose an upper bound (say) MAX on the number of heartbeats i sends to j before receiving a heartbeat from j. Thus, after finitely many false suspicions, the value of $\mathsf{fd}_i.\mathsf{hbBount}[\mathrm{j}]$ exceeds MAX after which i never suspects j. That is, every correct process is eventually and permanently trusted by all correct processes.

VIII.5.2.2. Bichronal timer properties

Understanding the behavior of bichronal timers is critical to establishing correctness. Here we prove that, in Algorithms 17–19, when bichronal timers are started for bichronal value (a, r), they run for either exactly a control-protocol program actions, or exactly r units of local clock time.

Lemma VIII.5.1. *Let α be an arbitrary execution of Algorithms 17–19. Let a correct process i execute* startBichronalTimer(fd_i.aValue, fd_i.rValue) *at time t. Let the value of the* clock *at process i at time t be denoted* $\mathsf{clock}_{i,t}$. *Let $t' > t$ be the earliest time (after t) that* bichronalTimerExpired *is set to true (that is, the bichronal timer expires). In the closed interval $[t, t']$, either (1) i executes the guarded command in Algorithm 19 exactly* fd_i.aValue *times and* $\mathsf{clock}_{i,t'} - \mathsf{clock}_{i,t} \geq \mathsf{fd}_i$.rValue *(that is, the local real-time clock at i ticks for exactly* fd_i.rValue *units), or (2) i executes the guarded command in Algorithm 19 at least* fd_i.aValue *times and* $\mathsf{clock}_{i,t'} - \mathsf{clock}_{i,t} = \mathsf{fd}_i$.rValue *(that is, the local real-time clock at i ticks for exactly* fd_i.rValue *units).*

Proof. From the pesudo code in Algorithms 17–19, it can be verified that the state variables that maintain the bichronal timer are modified only by action takeStep, the macro startBichronalTimer in Algorithm 19, action realTimeTimerExpiry, and action bichronalTimerExpiry. It can be verified that: the action takeStep signals the expiry of the action clock component of the bichronal clock (when actionTimer $= 0$), the action realTimeTimerExpiry signals the expiry of the real-time clock component of the bichronal clock, and action bichronalTimerExpiry signals the expiry of the bichronal timer itself. Also, it can be verified that the guarded command in Algorithm 19 executes the macro startBichronalTimer only upon bichronal timer expiry.

Therefore, if i executes startBichronalTimer(fd_i.aValue, fd_i.rValue) at time t, then the bichronal timer is not restarted until the bichronal timer expires. That is, in

the closed interval $[t, t']$, process i does not execute startBichronalTimer(fd_i.aValue, fd_i.rValue).

Consequently, in the interval $[t, t']$, the value of actionTimer is decremented until actionTimer is zero, and the value of realTimeTimer is unchanged at fd_i.rValue.

Given that t' is the earliest time that bichronalTimerExpired is $true$ (and bichronalTimerExpired is set to false by the macro startBichronalTimer), we know that i executes action bichronalTimerExpiry at time t'. The precondition for executing bichronalTimerExpiry is (realTimeTimerExpired $\wedge$ (actionTimer $= 0$) $\wedge$ ($\neg$bichronalTimerExpired)) which is a disjunction in the predicate for stopping the time evolution as well. Therefore, at time t', actionTimer is zero and realTimeTimerExpired is $true$.

We now consider the change in the values of actionTimer and realTimeTimerExpired separately.

The state variable actionTimer is decremented by 1 in action takeStep, and takeStep also executes the guarded command in Algorithm 19. Let actionTimer decrement to zero at time (say) $t_a > t$. Since actionTimer was set to fd_i.aValue at time t, we know that in the interval $[t, t_a]$ i executes the guarded command in Algorithm 19 exactly fd_i.aValue times.

The only action that sets realTimeTimerExpired to $true$ is realTimeTimerExpiry which is executed when clock = realTimeTimer, and once realTimeTimerExpired is set to $true$, it is reset of $false$ only by startBichronalTimer(fd_i.aValue, fd_i.rValue). Also, note that time evolution stops when realTimeTimerExpired is $false$ and clock = realTimeTimer. Given that at time t the value of realTimeTimer is set to $\mathsf{clock}_{i,t}$ + fd_i.rValue, we know that there exists a time $t_r > t$ such that $\mathsf{clock}_{i,t_r} = \mathsf{clock}_{i,t}$ + fd_i.rValue. Thus, at time t_r, action realTimeTimerExpiry is enabled and the time evolution is stopped. Consequently, i executes realTimeTimerExpiry and sets realTimeTimerExpired to $true$ at time t_r.

If $t_a \geq t_r$, then $t' = t_a$; that is, in the interval $[t, t']$, i executes the guarded command in Algorithm 19 exactly $\mathsf{fd}_i.\mathsf{aValue}$ times, and $\mathsf{clock}_{i,t'} - \mathsf{clock}_{i,t} \geq \mathsf{fd}_i.\mathsf{rValue}$

On the other hand, if $t_r \geq t_a$, then $t' = t_r$. Consequently, $\mathsf{clock}_{i,t'} = \mathsf{clock}_{i,t} + \mathsf{fd}_i.\mathsf{rValue}$. That is, $\mathsf{clock}_{i,t'} - \mathsf{clock}_{i,t} = \mathsf{fd}_i.\mathsf{rValue}$. Also, i executes the guarded command in Algorithm 19 at least $\mathsf{fd}_i.\mathsf{aValue}$ times.

Thus proved. □

Note that from Lemma VIII.5.1 and the fact that processes send heartbeats every time their bichronal timers expire, and they immediately restart the bichronal timer, we know that there exists an upper bound on the bichronal duration between the times at which two consecutive heartbeats are sent. Alternatively, we say that heartbeats are sent with *bichronal bounded persistence.*

VIII.5.2.3. Strong completeness

Strong completeness states that every crashed process is eventually and permanently suspected by all correct processes. From line 13 in Algorithm 19, we know that a correct process i can suspect a faulty process j only upon expiry of the bichronal timer at i. So in order to prove strong completeness, we first show that the bichronal timer at i expires infinitely often.

Lemma VIII.5.2. *In an execution α of Algorithm 19 where i is a correct process, the macro* $\mathsf{startBichronalTimer}(\mathsf{fd}_i.\mathsf{aValue}, \mathsf{fd}_i.\mathsf{rValue})$ *is executed infinitely often.*

Proof. From Algorithm 17 we know that initially, $\mathsf{bichronalTimerExpired}$ is set to *true*. From lines 9 and 14 in Algorithm 19, we know that when a correct process i executes the guarded command *Action 1* when $\mathsf{bichronalTimerExpired} = \mathit{true}$. Therefore, i executes the macro $\mathsf{startBichronalTimer}(\mathsf{fd}_i.\mathsf{aValue}, \mathsf{fd}_i.\mathsf{rValue})$ when it executes the guarded command in Algorithm 19 the first time. As a result, the bichronal timer

starts counting down from ($\mathsf{fd}_i.\mathsf{aValue}, \mathsf{fd}_i.\mathsf{rValue}$) by resetting actionTimer to $\mathsf{fd}_i.\mathsf{aValue}$ and realTimeTimer to $\mathsf{clock} + \mathsf{fd}_i.\mathsf{rValue}$).

Observe that actionTimer is decremented in action takeStep and reset to $\mathsf{fd}_i.\mathsf{aValue}$ only after bichronalTimerExpired is set to $true$. Since i is correct, i executes action takeStep infinitely often, and therefore, while bichronalTimerExpired is $false$, then eventually actionTimer is set to 0.

Observe from line 9 and 14 in Algorithm 17 that the value of realTimeTimer is modified only after bichronalTimerExpired is set to $true$. Therefore, while bichronalTimerExpired is $false$, the value of realTimeTimer remains unmodified. From line 14 in Algorithm17 and the definition of the alias startBichronalTimer in Algorithm17, we know that every time the value of realTimeTimer is modified, it is reset to $\mathsf{clock} + \mathsf{fd}_i.\mathsf{rValue}$). Therefore, $\mathsf{fd}_i.\mathsf{rValue}$ clock time units after realTimeTimer was reset, realTimeTimer equals clock; and that forces the execution of the action realTimeTimerExpiry. As a result, realTimeTimerExpired is set to $true$.

Thus, after every instance of executing startBichronalTimer($\mathsf{fd}_i.\mathsf{aValue}, \mathsf{fd}_i.\mathsf{rValue}$), which sets bichronalTimerExpired to $false$, in α, there exists a future state s_{exp} in α where actionTimer equals zero and realTimeTimerExpired is $true$. This enables action bichronalTimerExpiry which sets bichronalTimerExpired to $true$.

In the execution of the control-protocol program action at i that follows state s_{exp} in α, we know from lines 9 and 14 in Algorithm 19 that bichronalTimerExpired is set to $true$, and i executes macro startBichronalTimer($\mathsf{fd}_i.\mathsf{aValue}$, $\mathsf{fd}_i.\mathsf{rValue}$), again.

Thus, in α for every state of the system where bichronalTimerExpired at i is set to $true$, there exists a future state after which i executes the macro startBichronalTimer ($\mathsf{fd}_i.\mathsf{aValue}$, $\mathsf{fd}_i.\mathsf{rValue}$) in α, and for each state that follows i's execution of startBichronalTimer ($\mathsf{fd}_i.\mathsf{aValue}$, $\mathsf{fd}_i.\mathsf{rValue}$), there exists a future state where bichronalTimerExpired at i is set to $true$ in α. Thus, we have shown that in every run α of

Algorithm 19 where i is a correct process, the macro startBichronalTimer(fd_i.aValue, fd_i.rValue) is executed infinitely often. □

Theorem VIII.5.3. *The implementation described in Algorithms 17–19 satisfies strong completeness.*

Proof. Recall that the strong completeness property states that every crashed process is eventually and permanently suspected by all correct processes.

Consider a run of the algorithm in Algorithms 17–19 and let process i be an arbitrary correct process and j be an arbitrary faulty process. Let j crash at time t_c. Let t_f be the time of the last receipt of heartbeats sent by j to i. After time t_f, process i does not receive any heartbeat from j, therefore the if condition in line 4 in Algorithm 19 at i evaluates to $false$ in the infinite suffix. Consequently, if j is suspected by i after time t_f, then i will suspect j permanently thereafter.

At time t_f, process i has some finite value of fd_i.hbCount[j]. We know from Lemma VIII.5.2 that the bichronal timer is restarted infinitely often, and every time the bichronal timer expires fd_i.hbCount[j] is decremented by one, so that eventually fd_i.hbCount[j] counts down to zero. Consider a time $t_{exp} > t_f$ at which fd_i.hbCount[j] is zero. Let t_s be the earliest time after t_{exp} that the program action in Algorithm 19 is executed. At time t_s the if condition in line 13 of Algorithm 19 evaluates to true. Therefore the correct process i adds j to its suspect list at t_s.

Thus, all correct processes eventually and permanently suspect all crashed processes. □

VIII.5.2.4. Eventual strong accuracy

In order establish eventual strong accuracy, we have to show that eventually fd_i.hbCount[j] remains non-zero for all pairs of correct processes i and j. In order to

show that, we have to prove that there exists an upper bound on the number of heartbeats i sends to j before i receives a heartbeat from j. The following lemmas are used to establish the aforementioned result.

In Lemma VIII.5.4 we show that in the duration spanning a start and the corresponding expiry of a bichronal timer at j, there exists an upper bound on the number of heartbeats sent by i. Note that j does not send a heartbeat immediately upon the expiry of the bichronal timer; j sends a heartbeat to i only in the program action after the bichronal time expiry. In Lemma VIII.5.5 we show that in the duration between bichronal timer expiry and the next execution of the program action at j, there exists an upper bound on the number of heartbeats sent by i. Subsequently, in Lemma VIII.5.6 we show that while a privileged heartbeat from j is transit to i, there exists an upper bound on the number of heartbeats sent by i.

In the lemmas that follow, let α be an arbitrary execution of Algorithms 17–19, and let i and j be an arbitrary pair of correct processes.

Lemma VIII.5.4. *Let j start its bichronal timer at time (say) t and let $t' > t$ be the earliest time (after t) that the bichronal timer expires. In the closed interval $[t, t']$, i sends at most* $\max(\Phi \times \frac{\mathsf{fd}_j.\mathsf{aValue}}{\mathsf{fd}_i.\mathsf{aValue}}, D^2 \times \frac{\mathsf{fd}_j.\mathsf{aValue}}{\mathsf{fd}_i.\mathsf{aValue}})$ *heartbeats to j, where Φ is the upper bound on relative process speeds and D is the upper bound on the clock drift rate.*

Proof. At time t, process j starts its bichronal clock by executing startBichronalTimer ($\mathsf{fd}_j.\mathsf{aValue}$, $\mathsf{fd}_j.\mathsf{rValue}$). Let $t' > t$ be the earliest time (after t) that the bichronal timer expires. From Lemma VIII.5.1 we know that in the interval $[t', t]$ either process j executes its control-protocol program action exactly $\mathsf{fd}_j.\mathsf{aValue}$ times or the local clock at j ticks for exactly $\mathsf{fd}_j.\mathsf{rValue}$ units.

Recall that Φ is the bound on the relative process speeds, and D is the bound on drift rate. In the interval $[t', t]$, if j executes its control-protocol program action

exactly $\mathsf{fd}_j.\mathsf{aValue}$ times, then i executes its control-protocol program action at most $\Phi \times \mathsf{fd}_j.\mathsf{aValue}$. On the other hand, in the interval $[t', t]$, if the local clock at j measures exactly $\mathsf{fd}_j.\mathsf{rValue}$ units, then the interval $[t', t]$ is at most $D \times \mathsf{fd}_j.\mathsf{rValue}$ time units; consequently, in the the interval $[t', t]$, the local clock at i measures at most $D \times D \times \mathsf{fd}_j.\mathsf{rValue}$) time units. To summarize, in the interval $[t', t]$, either i executes its control-protocol program action at most $\Phi \times \mathsf{fd}_j.\mathsf{aValue}$ times or the local clock at i measures at most $D^2 \times \mathsf{fd}_j.\mathsf{rValue}$ time units.

From Lemma VIII.5.1 we know that the bichronal timer at process i runs for at least $\mathsf{fd}_i.\mathsf{aValue}$ executions of the guarded command in Algorithm 19. Therefore, in the interval $[t, t']$, if i executes its control-protocol program action at most $\Phi \times \mathsf{fd}_j.\mathsf{aValue}$ times, then the bichronal timer at i is restarted at most $\Phi \times \frac{\mathsf{fd}_j.\mathsf{aValue}}{\mathsf{fd}_i.\mathsf{aValue}}$ times.

From Lemma VIII.5.1 we know that the bichronal timer at process i runs for at least $\mathsf{fd}_j.\mathsf{rValue}$ unit ticks of the local real-time clock at i. Therefore, in the interval $[t, t']$, if the local clock at i ticks at most $D^2 \times \mathsf{fd}_j.\mathsf{rValue}$ time units, then the bichronal timer at i is restarted at most $D^2 \times \frac{\mathsf{fd}_j.\mathsf{rValue}}{\mathsf{fd}_i.\mathsf{rValue}}$ times.

Recall from lines 9–14 in Algorithm 19 that every time the bichronal timer expires at i, process i sends a heartbeat to j. Therefore, in the interval $[t, t']$, during which the bichronal timer runs at most $\max(\Phi \times \frac{\mathsf{fd}_j.\mathsf{aValue}}{\mathsf{fd}_i.\mathsf{aValue}}, D^2 \times \frac{\mathsf{fd}_j.\mathsf{rValue}}{\mathsf{fd}_i.\mathsf{rValue}})$ times, i sends at most $\max(\Phi \times \frac{\mathsf{fd}_j.\mathsf{aValue}}{\mathsf{fd}_i.\mathsf{aValue}}, D^2 \times \frac{\mathsf{fd}_j.\mathsf{rValue}}{\mathsf{fd}_i.\mathsf{rValue}})$ heartbeats. □

Lemma VIII.5.5. *Let time t_{exp} be a time at which j executes* bichronalTimerExpiry *in α (that is, the bichronal timer at j expires at time t_{exp}). Let $t_{hb} > t_{exp}$ be the earliest time (after t_{exp}) in α at which j sends a heartbeat to i. In the interval $[t_{exp}, t_{hb}]$, process i sends at most $\frac{\Phi}{\mathsf{fd}_i.\mathsf{aValue}}$ heartbeats to j.*

Proof. Since, j executes action bichronalTimerExpiry at time t_{exp}, bichronalTimerExpired is $true$ at time t_{exp}. Note that bichronalTimerExpired is set to $false$ only in macro

startBichronalTimer in the guarded command in Algorithm 19. Let $t_{gc} > t_{exp}$ be the earliest time (after t_{exp}) in α at which j executes action takeStep and thus executes the guarded command in Algorithm 19. Note that at time t_{gc} bichronalTimerExpired is *true*. Therefore, when j executes the guarded command in Algorithm 19, the if condition in line 9 of Algorithm 19 evaluate to *true*, and hence j sends a heartbeat to i. Therefore, $t_{gc} = t_{hb}$, and in the interval $[t_{exp}, t_{hb}]$ process j executes takeStep exactly once.

Since relative process speeds is bounded above by Φ, in the interval $[t_{exp}, t_{hb}]$ process i executes takeStep (and hence, the guarded command at Algorithm 19) at most Φ times. Since, i executes its control-protocol program actions at least $\mathsf{fd}_i.\mathsf{aValue}$ times between the start of the bichronal timer and its expiry, in the interval $[t_{exp}, t_{hb}]$ the bichronal timer at process i expires at most $\frac{\Phi}{\mathsf{fd}_i.\mathsf{aValue}}$ times. Every time the bichronal timer at i expires, i sends a heartbeat to j. Therefore, in the interval $[t_{exp}, t_{hb}]$ process i sends at most $\frac{\Phi}{\mathsf{fd}_i.\mathsf{aValue}}$ heartbeats to j. □

Lemma VIII.5.6. *Let t_{priv} denote a time in α at which j sends a heartbeat m to i, and m is privileged. While m is in transit, i sends at most $\frac{D \times \Delta}{\mathsf{fd}_i.\mathsf{rValue}}$ heartbeats to j.*

Proof. Recall from Section VIII.5.2.1 that the communication link between i and j is an ADD channel, and therefore, the message delay of privileged messages does not exceed Δ time units. Therefore, while m, a privileged message is in transit from j to i, the local clock at i advances by at most $D \times \Delta$ time units. In the same duration, the bichronal timer at i expires at most $\frac{D \times \Delta}{\mathsf{fd}_i.\mathsf{rValue}}$ times, and every time the bichronal timer expires, i sends exactly one heartbeat to j. Therefore, m is in transit, i sends at most $\frac{D \times \Delta}{\mathsf{fd}_i.\mathsf{rValue}}$ heartbeats to j. □

We are now ready to prove that there exists an upper bound on the number of heartbeats i sends to j before i receives a heartbeat from j. Recall from the

specification of ADD channels that if j sends infinitely many messages to i, then for every interval of time where j sends exactly $R+1$ messages to i, at least one of these messages is guaranteed to be privileged. We use this property of ADD channels to establish our next lemma.

Lemma VIII.5.7. *Let i and j be an arbitrary pair of correct processes in the execution α. For every two instances of time t_1 and t_2 such that:*

- *i executes* takeStep *(and hence, executes the guarded command in Algorithm 19) at times t_1 and t_2 in α;*
- *the* if *condition in line 4 of the guarded command in Algorithm 19 at i evaluates to true (that is, i receives a message from j) at times t_1 and t_2; and*
- *at all times in the open interval (t_1, t_2), the* if *condition in line 4 of the guarded command in Algorithm 19 at i evaluates to false (that is, i does not receive a message from j between times t_1 and t_2);*

the following is true: i sends at most $\max(\Phi \times \frac{\mathsf{fd}_j.\mathsf{aValue}}{\mathsf{fd}_i.\mathsf{aValue}}, D^2 \times \frac{\mathsf{fd}_j.\mathsf{rValue}}{\mathsf{fd}_i.\mathsf{rValue}}) + \frac{\Phi}{\mathsf{fd}_i.\mathsf{aValue}} + \frac{D \times \Delta}{\mathsf{fd}_i.\mathsf{rValue}}$ *heartbeats to j in the interval $[t_1, t_2]$.*

Informally, the lemma states that there exists an upper bound on the number of heartbeats that i sends to j in the duration between consecutive receipts of heartbeats from j.

Proof. From Lemma VIII.5.2 we know that the bichronal timers at i and j are restarted infinitely often. Note that in the guarded command in Algorithm 19, every time a process restarts its bichronal timer, it also sends a heartbeat to each process. Therefore, in α, processes i and j send heartbeats to each other infinitely often.

From the properties of ADD channels (see Section VIII.5.2.1) we know that for all intervals of time in α where j sends $R+1$ heartbeats to i, at least one such heartbeat

is privileged which is delivered reliably with an upper bound Δ on delay. Since j sends heartbeats to j infinitely often, j sends privileged heartbeats to i infinitely often.

Let t_{p1} be a time that i receives a privileged heartbeat (denoted m_1) from j. Let $t_{p2} > t_{p1}$ be the earliest time (after t_{p1}) that i receives (another) privileged heartbeat (denoted m_2) from j. Next, we determine the maximum number of heartbeats that i sends to j in the interval $[t_{p1}, t_{p2}]$.

Note that if i receives m_1 at time t_{p1}, then j sends m_1 a time $t'_{p1} < t_{p1}$. Therefore, the maximum number of heartbeats that i sends to j in the interval $[t_{p1}, t_{p2}]$ does not exceed the maximum number of heartbeats that i sends to j in the interval $[t'_{p1}, t_{p2}]$.

The time interval $[t'_{p1}, t_{p2}]$ can be divided into two sub-intervals $[t'_{p1}, t'_{p2}]$ and $[t'_{p2}, t_{p2}]$ where t'_{p2} is the time at which j sends m_2.

From the properties of ADD channels, we know that in the interval $[t'_{p1}, t'_{p2}]$ process j sends at most $R+1$ heartbeats. From Lemmas VIII.5.4 and VIII.5.5, we know that in the duration between the times that j sends two heartbeats consecutively, i sends no more than $\max(\Phi \times \frac{\mathsf{fd}_j.\mathsf{aValue}}{\mathsf{fd}_i.\mathsf{aValue}}, D^2 \times \frac{\mathsf{fd}_j.\mathsf{aValue}}{\mathsf{fd}_i.\mathsf{aValue}}) + \frac{\Phi}{\mathsf{fd}_i.\mathsf{aValue}}$ heartbeats to j. Therefore, in the interval $[t'_{p1}, t'_{p2}]$, i sends at most $(R+1)(\max(\Phi \times \frac{\mathsf{fd}_j.\mathsf{aValue}}{\mathsf{fd}_i.\mathsf{aValue}}, D^2 \times \frac{\mathsf{fd}_j.\mathsf{aValue}}{\mathsf{fd}_i.\mathsf{aValue}}) + \frac{\Phi}{\mathsf{fd}_i.\mathsf{aValue}})$ heartbeats.

The interval $[t'_{p2}, t_{p2}]$ is the duration for which m_2 is in transit. Since m_2 is privileged, we can apply Lemma VIII.5.6 to show that in the interval $[t'_{p2}, t_{p2}]$, i sends at most $\frac{D \times \Delta}{\mathsf{fd}_i.\mathsf{rValue}}$ heartbeats to j.

Therefore, in the interval $[t_{p1}, t_{p2}]$, i sends at most $MAX \equiv (R+1)(\max(\Phi \times \frac{\mathsf{fd}_j.\mathsf{aValue}}{\mathsf{fd}_i.\mathsf{aValue}}, D^2 \times \frac{\mathsf{fd}_j.\mathsf{aValue}}{\mathsf{fd}_i.\mathsf{aValue}}) + \frac{\Phi}{\mathsf{fd}_i.\mathsf{aValue}}) + \frac{D \times \Delta}{\mathsf{fd}_i.\mathsf{rValue}}$ heartbeats.

Let the time at which i receives the x^{th} privileged message from j be denoted $t_{r(x)}$ (and let $t_{r(0)} = 0$). All the non-privileged messages, if any, are received by i in some interval of the form $[t_{r(x)}, t_{r(x+1)}]$. Therefore, every interval of time $[t_1, t_2]$ (where t_1 and t_2 are defined in the statement of the lemma) is some sub-interval $[t_{r(x)}, t_{r(x+1)}]$

(for some finite x). Given that in each interval $[0, t_{r(1)}]$, $[t_{r(1)}, t_{r(2)}]$, ..., $[t_{r(x)}, t_{r(x+1)}]$, ..., process i sends at most MAX heartbeats to j, therefore, in the interval of time $[t_1, t_2]$, process i sends at most MAX heartbeats to j. □

We are now ready to prove eventual strong accuracy. We prove eventual strong accuracy by showing that for all pairs of correct processes (i, j), process i may suspect j finitely many times, but after suspecting j at most MAX (defined in Lemma VIII.5.7) number of times, the value of $\mathsf{fd_i.hbBound[j]} = MAX + 1$, and i trusts j permanently thereafter.

Theorem VIII.5.8. *The implementation described in Algorithms17–19 satisfies eventual strong accuracy.*

Proof. Recall that eventual strong accuracy states that every correct process is eventually and permanently trusted by all correct processes.

Consider an execution α of Algorithms 17–19. Let i and j be two correct processes in α.

Let t_s be a time in α when i starts suspecting j by executing line 13 of Algorithm 19 (if no such time exists, then the theorem is satisfied vacuously). From Lemma VIII.5.2 we know that the bichronal timer at j expires infinitely often. Since j sends a heartbeat to i every time the bichronal timer at j expires, we know that j sends heartbeats to i infinitely often. From ADD channel properties we know that i receives an infinite subset of these heartbeats. Therefore, there exists a time $t_{ns} > t_s$ such that at t_{ns} process i receives a message from j. At time t_{ns}, the receipt of a heartbeat is processed in line 4 of Algorithm 19. Note that from lines 5–7 in Algorithm 19 we know at time t_{ns}, process i increments the value of $\mathsf{fd}_i\mathsf{.hbBound}$ and trusts j.

If only finitely many such times t_s exist, then consider the highest such t_s and the

corresponding highest such t_{ns} (denoted $t_{ns.high}$). By construction, i trusts j in the infinite suffix of α that follows time $t_{ns.high}$, thus satisfying eventual strong accuracy.

For the purpose of contradiction, we assume that infinitely many such times t_s exist. Consider a time $t_{s.MAX}$ such that t_s starts suspecting j by executing line 13 of Algorithm 19 and the value of $\mathsf{fd}_i.\mathsf{hbCount}[j]$ is MAX where $MAX \equiv (R + 1)(\max(\Phi \times \frac{\mathsf{fd}_j.\mathsf{aValue}}{\mathsf{fd}_i.\mathsf{aValue}}, D^2 \times \frac{\mathsf{fd}_j.\mathsf{aValue}}{\mathsf{fd}_i.\mathsf{aValue}}) + \frac{\Phi}{\mathsf{fd}_i.\mathsf{aValue}}) + \frac{D \times \Delta}{\mathsf{fd}_i.\mathsf{rValue}}$ from Lemma VIII.5.7. Let $t_{ns.MAX}$ denote the earliest time after $t_{s.MAX}$ when i receives a heartbeat from j. From lines 5–7 in Algorithm 19 we know at time t_{ns}, process i increments the value of $\mathsf{fd}_i.\mathsf{hbBound}$ to $MAX + 1$ and trusts j.

Note that i decrements $\mathsf{fd}_i.\mathsf{hbCount}[j]$ every time i sends a heartbeat, and i reset the value of $\mathsf{fd}_i.\mathsf{hbCount}[j]$ to $\mathsf{fd}_i.\mathsf{hbBound}$ (which is at least $MAX + 1$ after time $t_{ns.MAX}$). Applying Lemma VIII.5.7 for all time intervals $[t_1, t_2]$ where $t_1 \geq t_{ns.MAX}$, we know that i sends at most MAX heartbeats to j before receiving another heartbeat from j. Therefore, value of $\mathsf{fd}_i.\mathsf{hbCount}[j]$ is always positive when i receives a heartbeat from j after time $t_{ns.MAX}$. Consequently, after time $t_{ns.MAX}$, the if condition in line 13 of Algorithm 19 always evaluates to $false$, and hence i never suspects j after time $t_{ns.MAX}$.

Thus, we have shown that for every arbitrary pair of correct processes (i, j), process i eventually and permanently trusts j. In other words, the implementation in Algorithms 17–19 satisfies eventual strong accuracy. □

CHAPTER IX

CONCLUSION

The most exciting phrase to hear in science, the one that heralds new discoveries, is not "Eureka!" but "That's funny."

– *Issac Asimov*, as quoted in [25, p. 236]

IX.1. Summary of the results

Several existing techniques in crash tolerance have long been within the domain of the theory of distributed computing but their applicability in empirical systems was believed to be tenuous at best. This dissertation has developed new methodology and techniques to make these crash-tolerance techniques employable in empirical systems. We focused on $\mathcal{M}_*$ and its variant models of partial synchrony which have found significant favor with theoreticians but have largely been ignored by practitioners. This section provides a review of the new results in this dissertation and contrasts them with the current state of the art in distributed computing.

IX.1.1. Methodology

Typically, idealized partially synchronous models are assumed to exist (natively) for problem solving. In contrast, the methodology proposed here treats these models as problems that need solving. The algorithmic constructions presented in this dissertation implement a virtual execution environment that simulates the $\mathcal{M}_*$ (and other variant) partially-synchronous models on top of empirical systems. Thus, the crash-tolerance techniques developed for idealized partially-synchronous models like $\mathcal{M}_*$ can now be deployed on empirical systems through these algorithmic constructions.

To our knowledge, our results are the first to propose a generic methodology to implement idealized models of partial synchrony on top of common empirical systems in the presence of crash faults and infinite message loss. In related work, the methodologies proposed by [71, 72] to implement distributed schedulers using failure detectors consider shared-memory systems and already assume the existence of idealized shared memory primitives and use failure detectors without considering the underlying system within which the appropriate shared memory primitives and failure detectors are assumed to be implemented. Hence, these results present fragments of the proposed methodology, at best.

Failure detectors are central to our proposed methodology. Given the popularity of failure detectors, there have been several results exploring multiple facets of the failure detector abstractions in current literature. This dissertation provides a new implementation of the $\Diamond\mathcal{P}$ failure detector and clarifies our understanding of partial synchrony.

IX.1.2. Failure detectors

We employ failure detectors as an intermediary mechanism to encapsulate the underlying partial synchrony in empirical systems such that the encapsulated synchronism is used to implement the better-behaved $\mathcal{M}_*$ models on top of empirical systems. This dissertation makes the following contributions to our understanding of failure detectors.

IX.1.2.1. Process celeration

There have been several implementations of failure detectors like $\Diamond\mathcal{P}$ on real-time based system models without unbounded absolute process speeds. We showed that all these implementations are subtly broken if absolute process speeds are unbounded.

The failure of these existing implementations is inextricably linked to how time is measured in these systems. We showed that all mechanisms that employ a single time base (either real-time based or action based) to estimate end-to-end communication delay in these systems suffer from the *celeration gap* by which end-to-end communication delay is unbounded. Consequently, these $\Diamond\mathcal{P}$ implementations, which depend on bounded end-to-end communication delay, fail to behave correctly in executions where absolute process speeds are unbounded. We presented a novel mechanism called *bichronal time* to measure end-to-end communication delay such that even in runs where process speeds are unbounded, while real time or action time based measure of end-to-end communication delay may be unbounded, bichronal-time based measure of end-to-end communication delay remains bounded. This overcomes the *celeration gap* and permits provably correct implementations of $\Diamond\mathcal{P}$ on real-time based partially synchronous systems.

IX.1.2.2. Infinite message loss

Many existing implementations of $\Diamond\mathcal{P}$ assume (eventually) reliable communication or assume some trivial message loss pattern that enables processes to access a reliable communication sub-channel. We presented a $\Diamond\mathcal{P}$ implementation in systems with non-trivial message loss pattern such that processes could not access a reliable sub-channel and are forced to content with the uncertainties of infinite message loss. The proposed lossy channel, the Average Delayed/Dropped (ADD) channel is of independent interest because it provides the timeliness of $\mathcal{M}_*$ channels while retaining the uncertainties of message loss in a manner similar to the communication patterns observed in real-world distributed systems (cf. Section III.1.4.2 for a detailed justification).

IX.1.3. Partial synchrony

Since the introduction of the $\mathcal{M}_*$ models in [35], there have been two distinct, but unstated, interpretations of these models: *real-time* based interpretation, and *fairness* based interpretation. One set of results take a real-time based view of the $\mathcal{M}_*$ models and derive multiple system models, arguably customized to describe different distributed systems, that are real-time based variants of $\mathcal{M}_*$. Another set of results take a fairness-based view of $\mathcal{M}_*$ and derive other variants of $\mathcal{M}_*$. To our knowledge, we present the first detailed distinction of these two interpretations, describe the consequences of each interpretation, and argue for the irreducibility between the two interpretations.

Despite such irreducibility, we know that both interpretations of partial synchrony provide sufficient timeliness to solve problems in crash-prone systems. However, our results show that while real-time based timeliness is sufficient for crash tolerance, it is not necessary. On the other hand, fairness-based timeliness in partial synchrony is necessary and sufficient for crash tolerance.

IX.1.4. Weakest system models for failure detectors

Current work on the weakest system models for implementing failure detectors (see Section II.4.3) has met with limited success partly because the proposed system models assume real-time bounds on communication (and possibly computation too). Unfortunately, failure detectors do not preserve such real-time bounds. To find such weakest system models, we address a more fundamental question: what precisely about partial synchrony do failure detectors preserve?

We answered the foregoing question by demonstrating that failure detectors (at least when restricted to the Chandra-Toueg hierarchy [20]) encapsulate *fairness*: a

measure of the number of steps executed by a process relative to other events in the system. We argue that oracles are substitutable for the fairness properties (rather than real-time properties) of partially synchronous systems. We proposed fairness-based models of partial synchrony and demonstrate that they are, in fact, the 'weakest systems models' to implement the canonical failure detectors from the Chandra-Toueg hierarchy in the presence of arbitrary number of crash faults.

IX.1.5. Formal framework

The TIOA-based framework described in Chapter V may be of independent interest. To our knowledge, this is the first formal framework that provide a single consistent view of asynchrony, partial synchrony, failure detector properties, algorithms that implement failure detectors, and the algorithms that query failure detectors.

IX.2. Significance

The results in this dissertation provide new insights into future research direction, failure detector limitations, and failure detector performance. We discuss these insights next.

IX.2.1. Influence on future research

Failure detector oracles have gained tremendous popularity as the mechanism of choice for designing crash tolerant solutions. Our results further the shift in the direction of oracular research away from real-time notions of partial synchrony (which have traditionally been understood with respect to events that are essentially external to the system) and towards fairness-based partial synchrony (which can be understood solely with respect to other events that are internal to the system). In fact, our

results suggest that fairness is the currency for crash tolerance and research on weaker real-time bounds for crash tolerance should focus on enforcing appropriate fairness constraints on empirical systems relative to which known failure detector oracles can be implemented.

IX.2.2. Failure detector limitations

Even among fairness-based system models, questions on the ability of failure detectors to encapsulate partial synchrony remain. For instance, it was first noted in [22] that there exist time-free problems solvable in synchronous systems that are unsolvable with $\mathcal{P}$. This points to a 'gap in the synchronism' between $\mathcal{P}$ and the synchronous system. The following corollary of our results explains this gap.

$\mathcal{AF}$ — the weakest system model to implement $\mathcal{P}$ — is extremely similar to the synchronous system model with message delay being denominated in recipient's steps in the former and in real time in the latter. However, there is one significant difference. $\mathcal{AF}$ ensures full synchrony for each message as long as the sender is live. When a sender crashes, $\mathcal{AF}$ 'loses synchronism' for all the sender's messages that are still in transit. On the other hand, synchronous systems ensure the synchronism for these messages as well. This difference in the behavior between $\mathcal{AF}$ and synchronous systems is the 'gap in synchronism' between the perfect failure detector $\mathcal{P}$ and synchronous systems. To our knowledge, we are the first to characterize this gap.

In general, our results point to a limitation of a failure detector's ability to encapsulate fairness. Failure detectors encapsulate the fairness communication events for only as long as the sending and the receiving processes are live. If the sender or receiver of a message m crashes, while m is in transit, failure detectors fail to encapsulate any fairness guarantees on the delay of m. The intuition for this limitation is that upon the crash if a process, a failure detector is required to suspect that process

to satisfy completeness. However, in order to preserve the fairness constraints with respect to messages in transit from crashed processes, the failure detector is required to not suspect processes until all the messages that are in transit have been delivered. Depending on the 'strength' of the fairness constraints encapsulated by a failure detector, it may be possible to reliably determine if all such messages have been delivered. Therefore, permanent suspicion of crashed process will have to concede to possible fairness violations with respect to messages from a crashed process that are still in transit.

IX.2.3. Failure detectors and quality-of-service

Although our results argue that failure detectors are better understood as bounds on fairness and not real time, they do not discount the real-time bounds that empirical systems incidentally satisfy. The real-time bounds become useful when considering the performance or the Quality of Service (QoS) [24] provided by these oracles. In other words, our results provide a separation of concerns between the correctness and performance of oracles with respect to the temporal properties of the distributed systems. Specifically, our work shows that correctness of oracles can be determined and understood exclusively through the fairness constraints of the system, and once correctness has been established, the performance of the oracles can be analyzed exclusively through the real-time constraints that the system satisfies.

IX.3. Open problems

Several open problems and avenues for future research remain.

IX.3.1. On failure detectors and fairness

We have argued that several failure detector oracles encapsulate fairness in executions and provided evidence by demonstrating that the oracles in the Chandra-Toueg hierarchy encapsulate such fairness constraints. This opens a larger question: *do all oracles encapsulate fairness?* The answer is arguably *no*. Notable candidates for counterexamples include the failure detectors proposed in [53] whose output can be arbitrary and need not provide semantic information about process crashes alone. This presents another question: *what set of oracles do encapsulate fairness?* This question is open even when restricted to the extended Chandra-Toueg hierarchy (which include oracles like $\mathcal{T}$ [29], and other parametric oracles like the ones in [16, 72]). If it turns out that all oracles that output process ids do encapsulate fairness, then it provides us with a clean hierarchy of fairness-based system models that mirrors the extended Chandra-Toueg hierarchy. On the other hand, if we discover that there exist oracles within the extended Chandra-Toueg hierarchy that do not encapsulate fairness, then the implication is that these oracles encapsulate something other than fairness. Knowledge of this other encapsulated information could help in designing crash tolerant systems.

IX.3.2. Fault environments and fairness

Another consequence of failure detector oracles encapsulating fairness is that fault environments — defined as a set of fault patterns — could encapsulate fairness as well. Recall that the weakest oracles sufficient to solve problems in distributed systems vary depending on the number of processes that may crash. For instance, consider fault-tolerant consensus. Recall that $\Diamond\mathcal{S}$ is the weakest to solve the problem only in fault environments where a majority of the processes are correct [19]. In fault environments

where an arbitrary number of processes may crash, the weakest failure detector for the problem is a stronger oracle $(\Diamond\mathcal{S}, \Sigma)$ [28]. Given that $\Diamond\mathcal{S}$ encapsulates some fairness constraints, and Σ can be implemented in an asynchronous system with a majority of processes being correct, we conjecture that Σ and fault environments where a majority of the processes are correct encapsulate equivalent fairness constraints in the system. Furthermore, this implies that fairness is also encapsulated by constraints on the number of processes that may crash in the system. Based on the above observations and arguments, consider the following question: Is fairness a more general primitive to understand crash fault tolerance in distributed systems? That is, can fairness unify the different weakest failure detector results for the same problem in different fault environments?

IX.3.3. Fairness: a new currency for fault tolerance?

Much effort is spent pursuing the 'weakest' real-time-based models to implement certain oracles (like Ω, $\Diamond\mathcal{P}$, and such) for two reasons: (1) bounds in many empirical distributed systems are specified with respect to real time, and (2) these oracles are known to be the weakest to solve many problems in distributed computing. However, given the dependence of the weakest-oracle results on the fault environment, and the conjecture that fault environments themselves could encapsulate fairness, it is perhaps beneficial to investigate the 'weakest' real-time-based models to guarantee appropriate fairness constraints (rather than failure detectors) so that these constraints can then be encapsulated by various combinations of failure detectors and fault environments.

IX.3.4. Weakest system models to solve problems

Recall that $\Diamond\mathcal{S}$ is the weakest failure detector to solve consensus in asynchronous systems with a majority of correct processes [19], and we have shown that $\Diamond\mathcal{SF}$ is

the weakest fairness-based system model to implement $\Diamond\mathcal{S}$. Does that mean $\Diamond\mathcal{SF}$ is the weakest system model to solve consensus? The answer is *no*. While $\Diamond\mathcal{S}$ is the weakest to solve consensus only in majority-correct environments, $\Diamond\mathcal{SF}$ is the weakest to implement $\Diamond\mathcal{S}$ in all environments. This observation suggests that there is a weaker system model which can implement $\Diamond\mathcal{S}$ in majority-correct environments, but not in all environments.

The foregoing example illustrates the separation of failure detectors from classic problems in distributed computing. Informally, failure detectors are a means to an end, specifically, means to solving problems in crash prone environments. Therefore, specifying the weakest system models to implement failure detectors need not automatically translate to the weakest system model to solve the problems solvable by the failure detector. Given that our results provide strong evidence that fault tolerance depends on fairness in executions and not the real-time guarantees for computation and communication, future research on the weakest system models to solve various problems in crash prone systems should focus on fairness-based system models, and specifications of such system models remains unsolved.

REFERENCES

[1] Afek, Y., Attiya, H., Fekete, A., Fischer, M., Lynch, N.A., Mansour, Y., Wang, D.W., Zuck, L.: Reliable communication over unreliable channels. Journal of the ACM **41**(6), 1267–1297 (1994)

[2] Afek, Y., Gafni, E.: End-to-end communication in unreliable networks. In: Proceedings on the 7th Annual ACM Symposium on Principles of Distributed Computing, pp. 131–148 (1988)

[3] Aguilera, M.K., Chen, W., Toueg, S.: On quiescent reliable communication. SIAM Journal on Computing **29**(6), 2040–2073 (2000)

[4] Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Stable leader election. In: Proceedings of the 15th International Conference on Distributed Computing, pp. 108–122 (2001)

[5] Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing omega with weak reliability and synchrony assumptions. In: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing, pp. 306–314 (2003)

[6] Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Communication-efficient leader election and consensus with limited link synchrony. In: Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing, pp. 328–337 (2004)

[7] Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing omega in systems with weak reliability and synchrony assumptions. Distributed Computing **21**(4), 285–314 (2008)

[8] Anta, A.F., Raynal, M.: From an asynchronous intermittent rotating star to an eventual leader. IEEE Transactions on Parallel and Distributed Systems **21**(9), 1290–1303 (2010)

[9] Attiya, H., Dwork, C., Lynch, N.A., Stockmeyer, L.: Bounds on the time to reach agreement in the presence of timing uncertainty. Journal of the ACM **41**(1), 122–152 (1994)

[10] Attiya, H., Welch, J.: Distributed Computing : Fundamentals, Simulations, and Advanced Topics, second edn. John Wiley and Sons, Inc., Hoboken, NJ (2004)

[11] Awerbuch, B., Even, S.: Reliable broadcast protocols in unreliable networks. Networks **16**(4), 381–396 (1986)

[12] Basu, A., Charron-Bost, B., Toueg, S.: Simulating reliable links with unreliable links in the presence of process crashes. In: Proceedings of the 10th International Workshop on Distributed Algorithms, pp. 105–122 (1996)

[13] Bazzi, R.A., Neiger, G.: Simplifying fault-tolerance: providing the abstraction of crash failures. Journal of the ACM **48**(3), 499–554 (2001)

[14] Bertier, M., Marin, O., Sens, P.: Implementation and performance evaluation of an adaptable failure detector. In: Proceedings of the 32nd International Conference on Dependable Systems and Networks, pp. 354–363 (2002)

[15] Bhatt, V., Christman, N., Jayanti, P.: Extracting quorum failure detectors. In: Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, pp. 73–82 (2009)

[16] Biely, M., Hutle, M., Penso, L.D., Widder, J.: Relating stabilizing timing assumptions to stabilizing failure detectors regarding solvability and efficiency. In:

Proceedings of the 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems, pp. 4–20 (2007)

[17] Biely, M., Hutle, M., Penso, L.D., Widder, J.: Relating stabilizing timing assumptions to stabilizing failure detectors regarding solvability and efficiency. Tech. Rep. 54/2007, Technische Universität Wien, Institut für Technische Informatik (2007)

[18] Chandra, T.D., Griesemer, R., Redstone, J.: Paxos made live: an engineering perspective. In: Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing, pp. 398–407 (2007)

[19] Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM **43**(4), 685–722 (1996)

[20] Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43**(2), 225–267 (1996)

[21] Chandy, K.M., Misra, J.: The drinking philosophers problem. ACM Transactions on Programming Languages and Systems **6**(4), 632–646 (1984)

[22] Charron-Bost, B., Guerraoui, R., Schiper, A.: Synchronous system and perfect failure detector: solvability and efficiency issues. In: International Conference on Dependable Systems and Networks, pp. 523–532 (2000)

[23] Charron-Bost, B., Hutle, M., Widder, J.: In search of lost time. Information Processing Letters **110**(21), 928 – 933 (2010)

[24] Chen, W., Toueg, S., Aguilera, M.K.: On the quality of service of failure detectors. IEEE Transactions on Computers **51**(5), 561–580 (2002)

[25] Cline, R.B.: Becoming a Behavioral Science Researcher: A Guide to Producing Research That Matters. Guilford Press, New York, NY (2008)

[26] Cristian, F.: Understanding fault-tolerant distributed systems. Communications of the ACM **34**(2), 56–78 (1991)

[27] Cristian, F., Fetzer, C.: The timed asynchronous distributed system model. IEEE Transactions on Parallel and Distributed Systems **10**(6), 642–657 (1999)

[28] Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Hadzilacos, V., Kouznetsov, P., Toueg, S.: The weakest failure detectors to solve certain fundamental problems in distributed computing. In: Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing, pp. 338–346 (2004)

[29] Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Kouznetsov, P.: Mutual exclusion in asynchronous systems with failure detectors. Journal of Parallel and Distributed Computing **65**(4), 492–505 (2005)

[30] Dijkstra, E.W.: Solution of a problem in concurrent programming control. Communications of the ACM **8**(9), 569 (1965)

[31] Dijkstra, E.W.: Hierarchical ordering of sequential processes. Acta Informatica **1**(2), 115–138 (1971)

[32] Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM **18**(8), 453–457 (1975)

[33] Dolev, D., Dwork, C., Stockmeyer, L.: On the minimal synchronism needed for distributed consensus. Journal of the ACM **34**(1), 77–97 (1987)

[34] Dolev, S.: Self-Stabilization. MIT Press, Cambridge, MA (2000)

[35] Dwork, C., Lynch, N.A., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM **35**(2), 288–323 (1988)

[36] Eigen, M.: The origin of biological information. In: Symposium on the Development of the Physicist's Conception of Nature in the Twentieth Century, pp. 594–632 (1973)

[37] Fekete, A., Lynch, N.A., Mansour, Y., Spinelli, J.: The impossibility of implementing reliable communication in the face of crashes. Journal of the ACM **40**(5), 1087–1107 (1993)

[38] Fetzer, C.: Enforcing synchronous system properties on top of timed systems. In: Proceedings of the 7th Pacific Rim International Symposium on Dependable Computing, pp. 185–192 (2000)

[39] Fetzer, C., Raynal, M., Tronel, F.: An adaptive failure detection protocol. In: Proceedings of the 7th Pacific Rim International Symposium on Dependable Computing, pp. 146–153 (2001)

[40] Fetzer, C., Schmid, U., Süsskraut, M.: On the possibility of consensus in asynchronous systems with finite average response times. In: Proceedings of the 25th International Conference on Distributed Computing Systems, pp. 271–280 (2005)

[41] Fich, F., Ruppert, E.: Hundreds of impossibility results for distributed computing. Distributed Computing **16**(2-3), 121–163 (2003)

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

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

[44] Fischer, M.J., Merritt, M.: Appraising two decades of distributed computing theory research. Distributed Computing **16**(2-3), 239–247 (2003)

[45] Gärtner, F.C.: A gentle introduction to failure detectors and related subjects. Tech. Rep. TUD-BS-2001-01, Darmstadt University of Technology, Darmstadt, Germany (2001)

[46] Gopal, A.S.: Fault-tolerant broadcasts and multicasts: The problem of inconsistency and contamination. Ph.D. thesis, Cornell University (1992)

[47] Guerraoui, R., Kapalka, M., Kouznetsov, P.: The weakest failure detectors to boost obstruction-freedom. Distributed Computing **20**(6), 415–433 (2008)

[48] Guerraoui, R., Oliveira, R., Schiper, A.: Stubborn communication channels. Tech. Rep. LSR-REPORT-1998-009, Ecole Polytechnique Federale de Lausanne (1998)

[49] Helary, J.M., Hurfin, M.: Solving agreement problems with failure detectors: a survey. Annals of Telecommunications **52**(9–10), 447–464 (1997)

[50] Hermant, J.F., Widder, J.: Implementing reliable distributed real-time systems with the Θ-model. In: Proceedings of the 9th International Conference on the Principles of Distributed Systems, pp. 334–350 (2005)

[51] Hutle, M., Malkhi, D., Schmid, U., Zhou, L.: Chasing the weakest system model for implementing Ω and consensus. IEEE Transactions on Dependable and Secure Computing **6**(4), 269–281 (2009)

[52] Isard, M.: Autopilot: automatic data center management. ACM SIGOPS Operating Systems Review **41**(2), 60–67 (2007)

[53] Jayanti, P., Toueg, S.: Every problem has a weakest failure detector. In: Proceedings of the 27th Annual ACM Symposium on Principles of Distributed Computing, pp. 75–84 (2008)

[54] Kaynar, D.K., Lynch, N.A., Segala, R., Vaandrager, F.: The theory of Timed I/O Automata. Synthesis Lectures on Computer Science **1**(1), 1–114 (2006)

[55] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7), 558–565 (1978)

[56] Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems **16**(2), 133–169 (1998)

[57] Lampson, B.W.: Atomic transactions. In: Distributed Systems - Architecture and Implementation, An Advanced Course, pp. 246–265 (1981)

[58] Larrea, M., Arévalo, S., Fernández, A.: Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In: Proceedings of the 13th International Symposium on Distributed Computing, pp. 34–48 (1999)

[59] Larrea, M., Lafuente, A.: Communication-efficient implementation of failure detector classes $\diamond\mathcal{P}$ and $\diamond\mathcal{Q}$. In: Proceedings of the 19th International Symposium on Distributed Computing, pp. 495–496 (2005)

[60] Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco, CA (1996)

[61] Malkhi, D., Oprea, F., Zhou, L.: Ω meets Paxos: Leader election and stability without eventual timely links. In: Proceedings of the 19th International Symposium on Distributed Computing, pp. 199–213 (2005)

[62] Mostéfaoui, A., Mourgaya, E., Raynal, M.: Asynchronous implementation of failure detectors. In: Proceedings of the 33rd International Conference on Dependable Systems and Networks, pp. 351–360 (2003)

[63] Mostefaoui, A., Mourgaya, E., Raynal, M., Travers, C.: A time-free assumption to implement eventual leadership. Parallel Processing Letters **16**(2), 189–207 (2006)

[64] Mullender, S. (ed.): Distributed Systems, second edn. ACM Press, New York, NY (1993)

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

[66] Pike, S.M.: Distributed resource allocation with scalable crash containment. Ph.D. thesis, The Ohio State University, Department of Computer Science & Engineering (2004)

[67] Pike, S.M., Sastry, S., Welch, J.L.: Failure detectors encapsulate fairness. In: Proceedings of the 14th International Conference on Principles of Distributed Systems, pp. 173–188 (2010)

[68] Pike, S.M., Sivilotti, P.A.: Dining philosophers with crash locality 1. In: Proceedings of the 24th IEEE International Conference on Distributed Computing Systems, pp. 22–29 (2004)

[69] Pike, S.M., Song, Y., Sastry, S.: Wait-free dining under eventual weak exclusion. In: Proceedings of the 9th International Conference on Distributed Computing and Networking, pp. 135–146 (2008)

[70] Ponzio, S.: Consensus in the presence of timing uncertainty: omission and byzantine failures (extended abstract). In: Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing, pp. 125–138 (1991)

[71] Rajsbaum, S., Raynal, M., Travers, C.: Failure detectors as schedulers (an algorithmically-reasoned characterization). Tech. Rep. 1838, IRISA, Université de Rennes, France (2007)

[72] Rajsbaum, S., Raynal, M., Travers, C.: The iterated restricted immediate snapshot model. In: Proceedings of 14th Annual International Conference on Computing and Combinatorics, pp. 487–497 (2008)

[73] Robinson, P., Schmid, U.: The Asynchronous Bounded-Cycle Model. In: Proceedings of the 10th International Symposium on Stabilization, Safety, and Security of Distributed Systems, pp. 246–262 (2008)

[74] Sastry, S., Pike, S.M.: Eventually perfect failure detection using ADD channels. In: Proceedings of the 5th international Symposium on Parallel and Distributed Processing and Applications, pp. 483–496 (2007)

[75] Sastry, S., Pike, S.M., Welch, J.L.: Crash fault detection in celerating environments. In: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium, pp. 1–12 (2009)

[76] Sastry, S., Pike, S.M., Welch, J.L.: The weakest failure detector for wait-free dining under eventual weak exclusion. In: Proceedings of the 21st ACM Symposium

on Parallelism in Algorithms and Architectures, pp. 111–120 (2009)

[77] Schlichting, R.D., Schneider, F.B.: Fail-stop processors: an approach to designing fault-tolerant computing systems. ACM Transactions on Computer Systems **1**(3), 222–238 (1983)

[78] Song, Y., Pike, S.M.: Eventually k-bounded wait-free distributed daemons. Tech. Rep. TAMU-CS-TR-2007-2-1, Texas A&M University (2007)

[79] Song, Y., Pike, S.M., Sastry, S.: The weakest failure detector for wait-free, eventually fair mutual exclusion. Tech. Rep. TAMU-CS-TR-2007-2-2, Texas A&M University (2007)

[80] Thekkath, C.A., Mann, T., Lee, E.K.: Frangipani: a scalable distributed file system. ACM SIGOPS Operating Systems Review **31**(5), 224–237 (1997)

[81] Trenseni, M., Gazso, A., Reinhardt, H.: PaxosLease: Diskless paxos for leases. White Paper, Scalien. URL `http://scalien.com/pdf/PaxosLease.pdf`. Accessed on February 1^{st}, 2011

[82] Wattenhofer, R., Attiya, H., Malkhi, D., Marzullo, K., Mavronicolas, M., Pelc, A.: Edsger W. Dijkstra prize in distributed computing: 2007. URL `http://www.podc.org/dijkstra/2007.html`. Accessed on February 1^{st}, 2011

[83] Widder, J., Lann, G.L., Schmid, U.: Failure detection with booting in partially synchronous systems. In: Proceedings of the 5th European Dependable Computing Conference, pp. 20–37 (2005)

[84] Widder, J., Schmid, U.: Booting clock synchronization in partially synchronous systems with hybrid process and link failures. Distributed Computing **20**(2), 115–140 (2007)

APPENDIX A

CLASSIC PROBLEMS IN DISTRIBUTED COMPUTING

This appendix lists the many problems in distributed computing that have been referenced in this dissertation.

1. **The Consensus Problem.** In the Consensus problem [65], all processes propose a value and must reach a unanimous and irrevocable decision on one of these values. The Consensus problem is defined in terms of two primitives, *propose*(v) and *decide*(u). When a process executes *propose*(v), we say that it proposes v; similarly, when a process executes *decide*(u), we say that it decides u. The Consensus problem is specified as follows:

   - *Termination.* Every correct process eventually decides some value u.
   - *Integrity.* Every process decides at most once.
   - *Agreement.* No two correct processes decide differently.
   - *validity.* If a process decides v, then v was proposed by some process.

   The above definition is called *non-uniform consensus* since it permits faulty (but long-lived) to terminate with a decision value that is different from the correct processes'. Another variant called *uniform* consensus modifies the agreement condition to state that "No two process that decide, decide differently," and this ensure that even if a live, but faulty, process decides, it has to decide on the consensus value.

2. **Non-Blocking Atomic Commit** The non-blocking atomic commit problem resembles the consensus problem in that they both require processes to reach

a common decision. In non-blocking atomic commit, the processes propose to either *commit* a transaction or *abort* it. Unlike consensus, processes can decide of *commit* all processes propose *commit*.

3. **Leader Election** The leader election problem is equivalent to the consensus problem. In leader election, all the processes are required to agree on a single process on the *leader*. This is a special form on consensus in which each process proposes its own process id as input value to a consensus protocol and the process with the id that is output by the consensus protocol is considered to be the *leader*

4. **Stable Leader Election** Stable leader election [4] is a 'long-lived' variant of leader election in which processes are assumed to execute leader election repeatedly. Apart from the requirement that all processes elect a common *leader*, stable leader election requires that once a process i is elected leader, any successive leader election should elect i as the leader until i crashes.

5. **Mutual Exclusion.** In the mutual exclusion problem [30], a set of processes are assumed to have a specific set of actions that are "critical", and no more than one process can be in its respective "critical section" at any given time. A process that is executing its critical section is said to be *active*, a process vying for exclusive access to its critical section is said to be *trying*, and a process that is neither active nor trying is said to be *idle*. A correct solution to mutual exclusion requires that no more than one processes is *active* at any given time, and if processes are active only for a finite duration, then every *trying* process eventually becomes *active*.

6. **Dining Philosophers Problem.** The dining philosophers problem [31] is

a generalization of mutual exclusion in which each process (called *diners*) is in potential conflict with some only some subset of processes (diners) in the system. A dining instance is modeled by an undirected conflict graph $DP = (\Pi, E)$, where each vertex $p \in \Pi$ represents a diner, and each edge $(p, q) \in E$ represents the potential conflict between neighbors p and q. Each diner is either *thinking*, *hungry*, or *eating*. The above three states correspond to the four basic phases of a participating process: executing independently (*idle*), requesting access to critical section (*trying*), and executing its critical section (*active*, respectively.

Initially, every process (diner) is thinking. Although processes may think forever, they are permitted to become hungry at any time. Upon being scheduled to eat, the process enters its critical section. Eating is always finite (but not necessarily bounded) for correct processes; such processes must transit from eating to *thinking* in finite time. There are multiple variants of dining determined by the constraints of the transition of diners from being hungry to eating.

No-Deadlock dining solutions guarantee that if at some process is hungry, then eventually some process eats. *No-Lockout* dining solutions guarantee that if some process is hungry, then *that* process eventually eats. Similarly, *Wait-Free* dining solutions guarantee that every hungry diner eventually eats regardless of process crashes in the system. *Crash-Locality* n dining solutions guarantee that hungry processes that are n hops away from a crashed process in the conflict graph are guaranteed to eat, but processes within n hops may starve.

Strong Exclusion dining solutions guarantee that no two neighbors (live or crashed) eat simultaneously. *Weak Exclusion* dining solutions guarantee that no two live neighbors eat simultaneously. *Eventual Weak Exclusion* dining solutions guarantee that initially live neighbors may eat simultaneously, but eventually

no two live neighbors eat simultaneously.

7. **Clock Synchronization.** In clock synchronization [55], every process has access to a local physical clock and is allowed to modify the value of the clock such that the clock values at all correct processes are always approximately the same.

8. **Quiescent Reliable Communication.** The problem of quiescent reliable communication [3] assumes that all the processes in the system are connected to each other by lossy links. Therefore, for a process to reliably communicate a message to another process, multiple copies of the message have to be sent until an acknowledgment for that message is received. However, if a sender keeps sending copies of a messages without receiving an acknowledgment, then to could be because (1) the copies of the message or the acknowledgments are being dropped by the links, or (2) the recipient is crashed. In the problem of quiescent reliable communication: in the former case, the sender has to continue sending copies of the message until an ack is received, whereas in the latter case the process has to cease sending any messages to the recipient.

VITA

Srikanth Sastry received his Bachelor of Technology from Calicut University in India in 2001. He entered the doctoral program in the department of Computer Science and Engineering at Texas A&M University in August 2004 and received his Ph.D. in May 2011. His research interests include theory of distributed systems, wireless/mobile networks and systems, and algorithmic fault tolerance.

Srikanth Sastry can be reached by snail-mail at Department of Computer Science and Engineering, MS 3112 TAMU, College Station TX 77840. His email address is sastry@cse.tamu.edu.