Reliable and Real-Time Distributed Abstractions

THÈSE N^{O} 7289 (2016)

PRÉSENTÉE LE 16 DÉCEMBRE 2016

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE PROGRAMMATION DISTRIBUÉE
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

David KOZHAYA

acceptée sur proposition du jury:

Prof. V. Kuncak, président du jury
Prof. R. Guerraoui, directeur de thèse
Prof. F. Pedone, rapporteur
Dr Y.-A. Pignolet, rapporteuse
Prof. J.-D. Decotignie, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2016

Start by doing what's necessary; then do what's possible;
and suddenly you are doing the impossible.
— Saint Francis of Assisi

To Elias and Viviane…

Acknowledgements

Despite all the challenges faced in its pursuit, this PhD gave me a lot in return, not only academically, but more importantly personally and socially. I have had the chance to work, discuss and be-friend amazing people who made this experience a truly fantastic and fruitful one.

First, I would like to thank my adviser, Prof. Rachid Guerraoui, for his constant support, his trust in me and his excellent advice throughout my doctoral studies, which made this thesis possible. I would like to thank as well Dr. Yvonne-Anne Pignolet, who co-supervised the projects I worked on during my PhD, for her patience, advice, insightful feedback, and her kindness and concern especially for ensuring that I have an amazing time while working at ABB offices in Baden. I would also like to thank Dr. Dacfey Dzung and Dr. Manuel Oriol whom I enjoyed working with and whom I learned from greatly.

I want to thank Prof. Jean-Dominique Decotigni and Prof. Fernando Pedone for their time and valuable comments regarding my thesis and Prof. Viktor Kuncak for presiding my jury committee.

Many thanks to all my colleagues at the Distributed Programming Lab, Dr. Peva Blanchard, Dr. Victor Bushkov, Georgios Damaskinos, Dr. Alexandre Maurer, Mahdi El Mhamdi, Rhicheek Patra, Dr. Julien Stainer, Mahsa Taziki and Jingjing Wang who provided a sincerely great atmosphere both on- and off-work. But mainly, a huge thanks to Karolos Antoniadis, Dr. Vasileios Trigonakis, Tudor David, Georgios Chatzopoulos, Dragos-Adrian Seredinschi and Matej Pavlovic, who became more like brothers rather than colleagues; thank you for all the good times during this PhD and for making me feel among family. A special thanks to France Faille for her great care and patience in dealing with organizational details of the work events I was involved in.

A big thanks to Dr. Grace Zgheib, a lifetime friend who accompanied me with my academic journey since the bachelors till the end of my PhD, for her kind willingness to endure my nagging and complaints during the PhD years, but most importantly for being there for me. I would also like to thank my friends Maya Mansour, Hiba Fayyad and Rana Youssef-Rhayem for their constant thoughtfulness, advice and encouragement.

Last but surely not least, thanks for the people whom I could never thank enough, my parents, Elias and Viviane, and my sisters, Mireille, Carla and Mandy for their endless love and support without which I would have never been who I am now.

Lausanne, October 18 2016 D. K.

Preface

The work presented in this dissertation is part of the research conducted at the *Distributed Programming Laboratory* (LPD) at the School of Computer and Communication Sciences at EPFL. This work was performed under the supervision of Prof. Rachid Guerraoui and in collaboration with ABB Corporate Research, supervised by Dr. Yvonne-Anne Pignolet. The research in this dissertation was centered on investigating ways to build crucial distributed abstractions in *cyber-physical systems*, namely distributed control systems, smart grids and sensor networks, where new constraints have to be taken into account, such as, message losses, real-time responses and energy efficiency.

The material presented in this thesis is published in the following papers:

1. Rachid Guerraoui, David Kozhaya, and Yvonne-Anne Pignolet, "Right On Time Distributed Shared Memory". In *the Proceedings of the 37th IEEE Real-Time Systems Symposium* (RTSS 2016), 2016.

2. Rachid Guerraoui, David Kozhaya, Manuel Oriol, and Yvonne-Anne Pignolet, "Who's On Board? Probabilistic Membership for Real-Time Distributed Control Systems". In *the Proceedings of the 35th IEEE Symposium on Reliable Distributed Systems* (SRDS 2016), 2016.

3. Dacfey Dzung, Rachid Guerraoui, David Kozhaya, and Yvonne-Anne Pignolet, "Never Say Never - Probabilistic and Temporal Failure Detectors". In *the Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium* (IPDPS 2016), pages 679-688, 2016.

4. Dacfey Dzung, Rachid Guerraoui, David Kozhaya, and Yvonne-Anne Pignolet, "To Transmit Now Or Not To Transmit Now". In *the Proceedings of the 34th IEEE Symposium on Reliable Distributed Systems* (SRDS 2015), pages 246-255, 2015.

I was also involved in other research projects during my doctoral studies that resulted in the following publication:

Dacfey Dzung, Rachid Guerraoui, David Kozhaya, and Yvonne-Anne Pignolet, "Source Routing in Time-Varing Lossy Networks". In *the Proceedings of the International Conference on Networked Systems* (NETYS 2015), pages 200-215, 2015.

Lausanne, 18 October 2016 D. K.

Abstract

The celebrated distributed computing approach for building systems and services using multiple machines continues to expand to new domains. Computation devices nowadays have additional sensing and communication capabilities, while becoming, at the same time, cheaper, faster and more pervasive. Consequently, areas like industrial control, smart grids and sensor networks are increasingly using such devices to control and coordinate system operations. However, compared to classic distributed systems, such *real-world physical systems* have different needs, e.g., real-time and energy efficiency requirements. Moreover, constraints that govern communication are also different. Networks become susceptible to inevitable random losses, especially when utilizing wireless and power line communication.

This thesis investigates how to build various fundamental distributed computing abstractions (services) given the limitations, the performance and the application requirements and constraints of real-world control, smart grid and sensor systems. In quest of completeness, we discuss four distributed abstractions starting from the level of network links all the way up to the application level.

At the link level, we show how to build an energy-efficient reliable communication service. This is especially important for devices with battery-powered wireless adapters where recharging might be unfeasible. We establish transmission policies that can be used by processes to decide when to transmit over the network in order to avoid losses and minimize re-transmissions. These policies allow messages to be reliably transmitted with minimum transmission energy.

One level higher than links is failure detection, a software abstraction that relies on communication for identifying process crashes. We prove impossibility results concerning implementing classic eventual failure detectors in networks with probabilistic losses. We define a new implementable type of failure detectors, which preserves modularity. This means that existing deterministic algorithms using eventual failure detectors can still be used to solve certain distributed problems in lossy networks: we simply replace the existing failure detector with the one we define.

Using failure detectors, processes might get information about failures at different times. However, to ensure dependability, environments such as *distributed control systems* (DCSs), require a membership service where processes agree about failures in real time. We prove that the necessary properties of this membership cannot be implemented deterministically, given probabilistic losses. We propose an algorithm that satisfies these properties, with high probability. We show analytically, as well as experimentally (within an industrial DCS), that our technique significantly enhances the DCS dependability, compared to classic membership services, at low additional cost.

Finally, we investigate a real-time shared memory abstraction, which vastly simplifies programming control applications. We study the feasibility of implementing such an abstraction within DCSs, showing the impossibility of this task using traditional algorithms that are built on top of existing software blocks like failure detectors. We propose an approach that circumvents this impossibility by attaching information to the failure detection messages, analyze the performance of our technique and showcase ways of adapting it to various application needs and workloads.

Keywords: Distributed abstractions, probabilistic message loss, transmission policy, partially observable Markov decision problem, reliable links, failure detection, consensus, distributed control system, membership, distributed shared memory, real-time systems

Résumé

L'approche du calcul réparti, c'est-à-dire la coordination de multiples machines afin de construire des systèmes ou d'offrir différents services, voit son domaine d'application s'élargir de plus en plus. Aujourd'hui, les unités de calcul sont dotées de nouvelles capacités de communication et de mesures, tout en devenant moins chères, plus rapides et plus répandues. Par conséquent, des domaines comme le contrôle industriel, les réseaux intelligents, ou les réseaux de capteurs emploient de plus en plus de tels outils afin de contrôler et coordonner les opérations de leurs systèmes. Cependant, par rapport aux systèmes répartis classiques, ces systèmes physiques "réels" imposent des contraintes différentes, par exemple, de temps réel ou de rendement énergétique. De plus, les contraintes sur les communications sont également différentes. Les réseaux sont sujet à d'inévitables et imprévisibles pertes, notamment dans le cas de communication sans-fil ou par courants porteurs en ligne.

Cette thèse étudie comment diverses abstractions fondamentales (services) du calcul réparti peuvent être implémentées étant donné les contraintes imposées par les cas concrets de contrôle, de réseaux intelligents et de systèmes de capteurs. Par souci de complétude, nous présentons quatre abstractions, depuis la couche de liaison d'un réseau, jusqu'à la couche applicative.

Au niveau de la couche de liaison, nous montrons comment construire un service de communication fiable et énergétiquement efficace. Un tel service est particulièrement important dans le cas d'unités de calcul alimentées par batteries et communiquant sans fil, lorsque recharger ces batteries n'est pas toujours faisable. Nous définissons quatre politiques de transmission que les processus peuvent utiliser pour décider quand transmettre sur le réseau afin d'éviter les pertes et minimiser les retransmissions. Ces politiques permettent de transmettre des messages de manière fiable avec un minimum d'énergie.

Un niveau au-dessus de la couche de liaison se trouve la détection de défaillances. Il s'agit d'une abstraction logicielle fondée sur la communication qui permet d'identifier les processus morts. Nous prouvons l'impossibilité d'implémenter certains détecteurs classiques dans le cadre de réseaux avec défaillances probabilistes. Nous définissons un nouveau type de détecteur de défaillance qui est implémentable, et qui préserve la modularité. Cela implique que les algorithmes existant employant les détecteurs de défaillance classiques peuvent être déployés dans ces nouveaux réseaux directement : il suffit de substituer au détecteur classique un détecteur du type que nous avons défini.

Grâce aux détecteurs de défaillances, les processus peuvent obtenir des informations à propos des défaillances à différents instants. Cependant, pour garantir un niveau de fiabilité, certains environnements comme les systèmes de contrôle répartis (distributed control systems, DCS) requièrent un service d'adhésion où les processus se mettent d'accord à propos des défaillances en temps réel. Nous prouvons que, étant données des défaillances probabilistes, les propriétés d'un tel service d'adhésion ne peuvent pas être implémentées de manière déterministe. Nous présentons un algorithme qui satisfait ces propriétés, avec haute probabilité. Nous montrons, analytiquement, et expérimentalement (dans le cadre d'un DCS industriel), que notre approche améliore la fiabilité d'un DCS de manière significative, comparé aux services d'adhésion classiques, pour un coût additionel bas.

Enfin, nous examinons une abstraction de mémoire partagée temps-réel, qui simplifie largement la programmation d'applications de contrôle. Nous étudions la possibilité d'implémenter une telle abstraction au sein des DCSs. Nous prouvons qu'il n'est pas possible de résoudre ce problème en utilisant des algorithmes classiques qui emploieraient des modules logiciels existants comme des détecteurs de défaillance. Nous présentons une approche qui évite cette impossibilité en ajoutant des informations aux messages du détecteur de défaillances. Nous analysons les performances de notre technique, et illustrons les manières de l'adapter aux besoins et charges de travail de diverses applications.

Mots-clés : Abstraction répartie, perte probabiliste de messages, politique de transmission, problème de décision Markovien partiellement observable, liaisons fiables, détection de défaillances, consensus, système de contrôle réparti, adhésion, mémoire partagée répartie, système temps-réel.

Contents

List of Figures

List of Tables

1 Introduction

The world is moving so fast these days that the man who says it can't be done is generally interrupted by someone doing it.
— Elbert Hubbard

Speed has always intrigued the human mind. While the reason behind this fascination remains ambiguous, it may be explained by the simple ability of doing more and thus exploring new unattainable horizons when going faster. For example, speed has allowed us to travel across continents; speed permitted us to visit and explore other planets in a single lifetime duration; enough speed can even allow us, according to Einstein [2], to "slow down time" [3], a phenomenon known as *time dilation*.

The longing for more speed, at various levels, has in fact manifested itself in many human inventions. The desire for a faster computational ability than that of humans, for example, led to inventing computers. Decades of research later confirm this desire, as scientists and engineers have been constantly striving to increase the available computing power. In fact, this increase has sustained an exponential pace for the past four decades, a growth rate foreseen by Intel's co-founder Gordon E. Moore [4].

Over the course of many years, different computing trends have emerged with one goal in mind, that being, defying physical and technological limitations to obtain faster systems and hence a larger computing power. One such trend that has been prevailing over the past two decades is increased distribution, at the system and service level. That is, systems and services are built using multiple machines or computers, since the computing capabilities of a single machine are not always sufficient, for example, to provide the desired latencies. In fact, many important systems that we rely on nowadays, such as banking and weather prediction systems, planes, cars, power grid controls and the Internet, comprise tens if not hundreds, thousands or even millions of computers. Such a group of computers seeking to accomplish some form of cooperation or common goal, by sending and receiving messages over the network (interconnecting them), is known as a *distributed system*.

Roughly speaking, a distributed system can be viewed as a collection of *processes* and *links*. Processes represent the elements of the distributed system that are capable of performing computations. A process in this case might represent a physical computer, a processor within a computer or even a thread executing within a processor. In order to achieve a common task, these processes need to communicate. Links, in this sense, symbolize the communication ability between processes, and represent physical or logical networks supporting inter-process communication. Processes and links, thus, capture all underlying physical components that constitute a distributed system.

In order for processes to coordinate and deliver some coherent service, a *distributed algorithm* is needed. Such an algorithm manages the behavior of individual processes by assigning to every process a specific program to execute. A distributed algorithm is hence the collection of programs executed by all processes and is said to solve a *distributed problem*: the problem of making processes implement a specific service or a set of guarantees.

1.1 Distributed Computing Abstractions

It is a common occurrence that slight variations of the same distributed problem appear in different contexts [1]. One example of such problems is *reliable information dissemination* between processes, be it in *unicast* (one-to-one communication) or *broadcast* (one-to-all communication). Another example is the *consensus* problem [5], which requires agreement between processes to perform a certain task or to take a common decision.

To facilitate reasoning and to avoid repeatedly reinventing the same solutions, abstract versions of these problems are defined. A *distributed abstraction* is a representation of a distributed problem capturing the set of essential characteristics of that problem. In this sense, distributed abstractions help decipher the fundamental from the accessory and present the problem in a form suitable for mathematical study [1, 6].

Not only do distributed abstractions highlight the necessary aspects of problems, but they also allow solutions (distributed algorithms) to be developed in a modular fashion. In other words, existing abstractions can be used as building blocks when devising solutions to new higher-level problems.

1.2 System Models

In fact, processes and links, themselves, are abstractions of the computing and communication components of the distributed system. Defining properties of processes and links, for example, by specifying their mode of operation, the timing assumptions they adhere to (how computation and communication delays behave) and their failure patterns, leads to defining a *system model*. Different assumptions about processes and links define different system models and hence capture various instances of distributed systems. Typically, models are defined with properties that correspond to practical, real-life distributed system deployments.

Once a specific system model is considered, the next step is to understand how to build distributed abstractions in that model. Building the same abstraction in a different model might not be trivial, e.g., solving consensus [5] in models having different timing assumptions requires different algorithms and (solution) methods [7]. In fact, the newly considered model may pose challenges that could even render implementing that abstraction impossible. Consensus, for example, can be easily solved in models where processing and communication delays are bounded and known, even if processes can crash. However, the consensus problem becomes impossible to solve in a model with no assumptions about processing and communication delays, even if only one process can crash [8].

In short, systems may exhibit various events, e.g., some processes might stop operating, as a result of a crash, while others might continue normal operation, or some processes might become disconnected for some time as a result of network failures. The occurrence of such events, if not anticipated at design time, might cripple the abstraction by violating its specifications, rendering the solution, which was developed in the absence of such assumptions, unusable or incorrect.

1.3 Reliable and Real-Time Distributed Abstractions

With computation devices nowadays becoming cheaper and faster, while incorporating at the same time additional communication and sensing capabilities, distributed computing continues to expand reaching new realms and domains. In particular, the "distributed trend" in computing has managed to pave its way into application areas of transportation (auto-mobiles, avionics, railways, etc.), smart cities, grids and buildings, health-care (medical devices, remote surgery and even implantable devices) and industrial control (manufacturing plants, power plants, etc.), to name a few. Specifically, designers and engineers are attempting to realize new system capabilities by embedding more and more computing devices in the networks of these *real-world physical systems*, e.g., to control, coordinate and monitor various operations. This results in a new type of distributed systems[1] that have a different set of restrictions and needs. We further elaborate on this in what follows.

The set of constraints and specifications governing the operation of systems in these automation domains are not fully accounted for in classic distributed computing models and abstractions. For example, as opposed to the eventual guarantees of classic distributed systems, many applications of real-world physical systems typically have real-time requirements [10, 11], such as those needed, for instance, for detecting and recovering from failures [12–21]. Besides, networks of such systems also have energy efficiency concerns, e.g., regarding minimizing the energy consumption of battery-powered devices that might be operating part(s) of these automation systems and which might be tedious to recharge [22–29].

[1]Typically, such distributed systems are referred to as *cyber-physical systems* [9].

Moreover, as more and more devices (actuators, computation elements, etc.) are connected to such systems, communication networks become prone to inevitable message losses [30, 31]. These network losses could stem from unpredictable system loads or from physical properties of the different, possibly heterogeneous, communication media connecting such devices [32–34]. For example, networks in control systems, despite their high claimed reliability, typically experience link failure rates in the range of $10^{-5}/hr$ for permanent failures and $10^{-3}/hr$ for transient failures [35, 36]. These losses are believed to be even more severe when incorporating wireless and power line communication technologies, whose quality is influenced by path loss, fading, interference, switching of the power grid, activation of electrical equipment, and more [37–39]. In this sense, the quality of communication is affected by randomly occurring phenomena, which makes message losses over the network non-deterministic. In fact, link characteristics in general in such contexts can be, at best, seen as probabilistic and temporary rather than deterministic and perpetual [30–32, 37–40].

This thesis aims at developing a better understanding of the power and the limitations of distributed computing methods and abstractions given the communication behavior, the performance and the application needs and requirements of smart grid, sensor and distributed control systems. To this end, we investigate how to build *dependable* distributed abstractions that can indulge probabilistic message losses while providing non-trivial guarantees, such as reliability, energy efficiency and real-time responses, which are fundamental for the real-world physical systems under consideration. Specifically, we explore four substantial abstractions starting from the level of network links all the way up to the application level: (i) energy-efficient reliable communication, (ii) reliable failure detection, (iii) real-time group membership and (iv) real-time distributed shared memory. In what follows, we briefly introduce and discuss each of these abstractions.

1.3.1 Communication

At the lowest primitive level, the link level, we show how to build a peer-to-peer energy-efficient reliable communication service that is synchronous with high probability. In environments such as sensor networks [41–49], communication is lossy and some devices have battery-powered wireless adapters where recharging may be tedious or even impossible. Energy-efficient algorithms are crucial in these contexts [22–29, 50] in order to prolong the lifetime of the network. Distributed algorithms can benefit by transparently utilizing such a communication abstraction to deliver energy-efficient services to the whole network.

We investigate a setting where a link connecting two processes can alternate with time between being reliable and lossy. A sender process, without knowing the current state of the link, should "cleverly" decide, based on feedback relative to previous transmissions, when to send over the link in order to ensure a reliable transmission with minimum transmission energy consumption (precisely minimizing the number of re-transmissions, which are typically needed when messages are sent at times when the link is in a lossy state).

We study building the desired communication abstraction under several variations of acknowledgment-based (Ack/Nack) feedback mechanisms [51–54], which, to the best of our knowledge, sum-up all possible reliable and unreliable Ack/Nack feedback schemes. We compute transmission policies, that can be used by processes to specify when to transmit over the network such that messages are transmitted in a reliable energy-optimal manner, i.e., minimizing message re-transmissions. We show that the implementation complexity of these policies varies and depends on the feedback mechanism itself.

1.3.2 Failure Detection

Failure detection is a software abstraction that relies on inter-process communication for determining process crashes. Failure detectors thus exist one level higher than links and are often used to encapsulate synchrony assumptions regarding communication within distributed systems. This approach is known for its modularity, which allows the construction of algorithms using abstract failure detection mechanisms, defined by axiomatic properties, as building blocks. Determining how to implement the failure detection mechanism becomes a separate lower-level task.

Typically, classic failure detectors [7] require eventual synchrony guarantees on communication in order to be implemented. This means that certain links should stop delaying messages, after some point in time, forever and deterministically. In practice, such a link behavior is hard to guarantee. Synchrony assumptions may hold only probabilistically and temporarily, as shown by networking studies, especially in wireless and power line communication [30, 31].

To this end, we study failure detectors in a network with asynchrony inflicted by probabilistic synchronous communication. We show that the classic failure detectors, e.g., $\diamond\mathcal{S}$ for solving consensus [7], are impossible to implement in networks with probabilistic losses. To circumvent this impossibility, we refine the notion of classic failure detectors by defining new probabilistic failure detector abstractions, that can be implemented in lossy networks while preserving the failure detectors' modularity. Our solution thus allows existing deterministic algorithms based on classic failure detectors to be re-used in lossy networks for solving certain distributed problems. This is achieved by simply replacing the existing classic failure detector with our corresponding probabilistic one. Moreover, we derive lower bounds on the complexity of algorithms implementing our failure detectors. We also define communication optimality of failure detectors in systems with probabilistic losses and propose communication-optimal algorithms implementing our failure detectors.

1.3.3 Group Membership

Failure detectors provide processes with information about crashes in the system. However, when using failure detectors in their classic form [7], processes might get information about crashes at different times or in different orders. Having such "non-synchronized" failure information might be undesirable in certain situations, such as in *distributed control systems* (DCSs). For example, uncoordinated failure notifications in DCSs might cause the local components (within processes) that are responsible to jointly schedule application tasks to be inconsistent, resulting in applications executing incorrectly.

A higher abstraction level than failure detectors is hence needed, namely a *membership service*. A membership service in DCSs allows processes to agree at all times about what processes have crashed in the system, while detecting crashes in real time.

We investigate how to build such a real-time membership service in distributed control systems. We prove that the necessary membership properties in DCSs cannot be implemented deterministically if, besides process crashes, communication links can also fail. To this end, we define implementable probabilistic variants of membership properties, which constitute what we call a *synchronous membership service* (SYMS). We propose *ViewSnoop*, our algorithm for implementing SYMS with high probability.

We then evaluate our algorithm analytically as well as experimentally by implementing and deploying *ViewSnoop* in an industrial DCS framework. We show that *ViewSnoop* significantly improves the dependability of DCSs compared to membership schemes based on classic heartbeats, at low additional cost. Moreover, *ViewSnoop* distinguishes, with high probability, host crashes from message losses, enabling DCSs to counteract losses better than existing algorithms.

1.3.4 Distributed Shared Memory

Last but not least, we investigate a real-time distributed shared memory abstraction. Such an abstraction typically builds on top of existing software blocks like failure detection and is considered to greatly simplify programming of control applications [1].

We investigate the feasibility of implementing this abstraction within DCSs using algorithms that either do not have access to failure detector information or that rely on failure detectors as software blocks. Such algorithms thus cannot use failure detectors' internals. We prove that, in both cases, building a real-time DCS shared memory is impossible in the presence of message losses.

We propose a novel way to circumvent this impossibility by devising a white-box approach that uses failure detector messages to communicate between processes. More precisely, our algorithm, which we call *TapeWorm*, attaches information to the crash monitoring messages (heartbeats) of the failure detector element of DCSs, using these heartbeats as the sole means to send information between processes. We prove that *TapeWorm* indeed implements the required distributed shared memory abstraction. We then provide an analytic evaluation showing that our algorithm can be adapted to application needs and can be optimized for specific workloads.

1.4 Thesis Contributions

This thesis studies how to build distributed computing abstractions for distributed control, smart grid and sensor systems. In these environments, constraints and requirements such as message losses, real-time behavior and energy efficiency, dominate the design process. To this end, we investigate four abstractions starting from the basic communication link level and reaching abstractions that directly influence the application level.

The main contributions of this thesis are the following:

1. **Communication:**
   (a) A first full analytic study of energy-optimal transmission policies for links with varying quality under several reliable and lossy transmission feedback mechanisms.
   (b) A derivation of explicit and closed form solutions to implement energy-optimal transmission policies.
   (c) An establishment of the necessary conditions for reliable communication using energy-optimal transmission policies.

2. **Failure Detection:**
   (a) Impossibility result regarding the implementation of classic eventual failure detectors, e.g. $\diamond\mathcal{S}$ for solving consensus, in networks with probabilistic message losses.
   (b) A new probabilistic failure detector notion, which preserves failure detection modularity and can be implemented in lossy networks.
   (c) Lower bounds on the number of processes and links required for implementing our new failure detectors.

3. **Membership:**
   (a) Impossibility results for deterministically implementing membership requirements in distributed control systems (DCSs) with both host crashes and message losses.
   (b) SYMS, a probabilistic abstraction for the membership service of DCSs.
   (c) *ViewSnoop*, an algorithm implementing SYMS. *ViewSnoop* distinguishes host crashes from message losses without affecting accuracy.
   (d) An extensive experimental and analytic evaluation of *ViewSnoop*'s performance showing that *ViewSnoop* provides a significantly more dependable service for DCSs, compared to existing methods that rely on classic heartbeats.

4. **Distributed Shared Memory:**
   (a) A first precise derivation of the necessary guarantees that a shared memory abstraction must provide in DCSs.
   (b) Theoretical proofs showing that such guarantees are impossible to implement using traditional approaches [55–59], e.g., using algorithms that build on top of software blocks such as failure detectors.
   (c) *TapeWorm*, an algorithm that circumvents the above impossibility by following a white-box approach directly utilizing failure detector algorithms of DCSs. *TapeWorm* implements the required shared memory guarantees for applications running in DCSs.
   (d) An analytic evaluation quantifying the performance of *TapeWorm* and showcasing ways of adapting and optimizing *TapeWorm* to application needs and workloads.

1.5 Roadmap

This thesis is divided into four parts. The first part presents some background on distributed system models and networking communication models in Chapter 2 and Chapter 3 respectively.

The second part discusses the four abstractions we study, each presented in an individual chapter. Chapter 4 studies building an energy-optimal reliable communication link, which is synchronous with high probability. Chapter 5 investigates building failure detectors in lossy networks while preserving their modularity. Chapter 6 examines coordinating failure notifications in real time, providing a membership abstraction for distributed control systems. Chapter 7 inspects the feasibility and researches solution methods for implementing real-time distributed shared memory in the context of distributed control systems.

The third part sums up the main results of this thesis and contemplates about prospective research areas that remain an interesting open territory to explore. The fourth and last part is a dedicated appendix, which comprises some tedious computation details related to deriving closed form expressions and some discussions which are outside the scope of prior chapters.

A Glimpse On System Models Part I

History is for human self-knowledge ... the only clue to what man can do is what man has done. The value of history, then, is that it teaches us what man has done and thus what man is.
— R. G. Collingwood

This part of the thesis briefly surveys the literature of distributed computing models. It presents a selection of well-known process and link abstractions in addition to well-studied timing assumptions within the distributed computing community.

Besides distributed computing models, we describe the networking view of communication links, precisely regarding message losses in physical communication media, used in the context of smart grid, sensor and distributed control systems. We also compare this networking view of communication to the link abstractions of distributed computing.

2 Distributed Computing Models

This chapter briefly discusses some of the widely adopted system models in distributed computing, where processes communicate by message passing. We first distinguish some of the well known failure properties of processes and links. We describe afterwards properties related to the timing assumptions, i.e., the behavior of processes and links with respect to the passage of time.

A system model can be defined by combining different timing and failures properties of processes and links.

2.1 Process Abstractions

In a distributed system, every process executes the program assigned to it by the distributed algorithm. As long as the process adheres to the program assigned to it, that process is said to be correct. A process, however, is said to *crash* or *fail* whenever that process no longer abides by the algorithm. A process can deviate in many ways from the program (algorithm) assigned to it. As such, different process abstractions are defined to capture these different failure patterns. We discuss next, four process abstractions, depicted in Figure 2.1, representing the most common four types of process failures used in the distributed computing literature.

2.1.1 Crash-Stop

The *crash-stop* abstraction depicts a process failure pattern where a crashed process stops executing any further steps. In other words, a process in this abstraction executes the program assigned to it correctly. However, this process may reach a certain point in time, say t, after which it permanently stops performing computations and sending messages. This behavior implies that in the *crash-stop* abstraction, once a process crashes, it never recovers.

Processes in the system can thus be either *faulty* or *correct*. A process that never crashes during the algorithm's execution time is said to be correct. Otherwise, the process is said to be faulty.

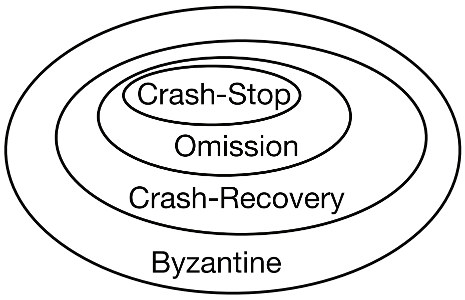

Figure 2.1 – Types of process abstractions [1]

It is important to note that considering that crashed processes never recover in the crash-stop abstraction does not eliminate the possibility of processes recovering in practice. It rather means that algorithms devised assuming crash-stop processes do not rely on processes to recover in order to continue execution [1].

As such, distributed algorithms considering this abstraction are typically devised to work when no more than f processes in the system can be faulty, where f could be zero or all processes except one.

2.1.2 Crash-Recovery

Guaranteeing that f processes will not fail during the entire execution of an algorithm, for any value of f, might be a difficult (even an implausible) requirement in certain systems, such as, distributed operating systems and grid computing [60]. Hence, in such environments, algorithms based on the assumption that some processes do not fail, will not work. This situation necessitates a new process abstraction: the *crash-recovery* abstraction.

This abstraction basically allows processes to recover and thus resume the execution of the algorithm. A process is termed faulty in this case when the process crashes and never recovers or crashes and recovers an infinite number of times. A correct process in this abstraction, hence, is a process that never crashes or that crashes and recovers a finite number of times.

2.1.3 Omission

As part of the algorithm assigned to it, a process might be required to send and/or receive messages at certain points in time or relative to certain events. *Omission* faults characterize the following behavior about processes: an omission fault occurs when a process does not send or receive a message, which that process was supposed to send or receive according to the distributed algorithm [1]. In this sense, a process in the omission fault abstraction diverges from the algorithm assigned to it, and hence crashes, by dropping messages that should have been exchanged with other processes. Omission faults in practice can result from stack/buffer overflows, synchronization errors, congestion, etc.

2.1.4 Byzantine

The *Byzantine* process abstraction, also known as *arbitrary-fault* or *malicious*, is considered as the most general type of process failure. Processes in the Byzantine model can deviate from the distributed algorithm in any plausible way. For example, processes might not perform some or all local operations, processes also might drop (at sending or receiving) some messages or even modify the contents of messages [1, 6].

Obviously, Byzantine faults are the most expensive faults to tolerate when designing algorithms. However such arbitrary process faults could be the only option in environments where unpredictable faults can occur. Such is the case, for example, in systems vulnerable to many attacks where some processes become controlled by users who seek to deliberately interrupt the system operation. Byzantine faults do not always occur as a result of intentional and malicious behaviors. Bugs in implementation, the programming language or even the compiler could cause such arbitrary faults.

2.2 Communication Abstractions

In distributed computing, communication between a pair of processes is typically abstracted by a logical link. In practice, these two processes might or might not have a direct physical communication medium between each other, depending on the underlying network topology. Still, communication between these two processes might be possible, for example, using routing algorithms which might relay messages over multiple hops. As such, the job of a link boils down to relaying information between processes. A link takes the message from the sender process and hands this message to the receiver process. In this sense, links can alter messages, e.g., by dropping them, regardless if processes are behaving correctly or not.

We elaborate next on two widely-used point-to-point link abstractions in the distributed computing community (broadcast or multicast abstractions are typically built on top of point-to-point links). These point-to-point abstractions define what happens to messages sent by processes. We adopt the classic distributed computing style for recalling the specifications of such communication abstractions [1, 6].

2.2.1 Fair-Loss Links

Messages transiting through the network can be lost due to various reasons. The *fair-loss* link abstraction captures this property, assuming that a message, when sent, has a chance of successfully reaching the destination.

Formally, the fair-loss link is defined through the following three properties [1]:

1. *Fair-loss*: A correct process q delivers a message m an infinite number of times, if some correct process p sends m infinitely often to q.

2. *Finite duplication*: A message m that is sent a finite number of times cannot be delivered an infinite number of times.
3. *No creation:* If some process q delivers a message m such that process p is the sender, then p has previously sent m to q.

The first property of the fair-loss abstraction guarantees that if a process keeps on re-transmitting a message, then that message is guaranteed to reach the destination, assuming that neither the sender nor the receiver processes crash.

The second property ensures that the network cannot perform infinite re-transmissions on its own, if that is not initiated by the sender process. Finally the last property ensures that the network does not create any new messages, besides the ones sent by processes.

2.2.2 Reliable Links

Compared to the fair-loss links, the *reliable* links abstraction provides higher level guarantees. Specifically, reliable links abstract the re-transmission mechanism that guarantees successful message delivery over fair-loss links and eliminate messages from being duplicated when delivered by processes.

More formally, reliable links ensure the following three properties [1]:

1. *Reliable delivery*: If a correct process p sends a message m to a correct process q, then q eventually delivers m, i.e., q delivers m at some point in time after being sent.
2. *No duplication*: No message is delivered more than once by a process.
3. *No creation:* If some process q delivers a message m with sender p, then p has previously sent m to q.

The first property guarantees that if a correct process sends a message to another correct process then this message will be delivered (by the destination) at some point in time.

The second property eliminates duplicate message deliveries and the third property, as in fair-loss links, ensures that no messages are created out of thin air.

2.3 Process and Link Timing Models

Besides defining the failure properties of processes and links, i.e., the multiple ways that processes and links could "misbehave" when running a distributed algorithm, it also important to characterize how processes and links behave with respect to time. For instance, this means specifying if and how bounds on processing and communication delays hold. We next discuss three main distributed computing models for different process and link timing assumptions.

2.3.1 Asynchronous System

An *asynchronous* system makes no timing assumptions about processes and links. That is, in an asynchronous system, processes do not have access to physical clocks and no bounds are assumed concerning processing and communication delays.

Despite the absence of physical time, some notion of *logical time* [61] can be established in an asynchronous system based on the sent and received messages. This logical notion of time allows us to provide some form of *partial ordering* [61] of certain events that happened in the system.

2.3.2 Synchronous System

A *synchronous* system can be viewed as the other extreme, compared to the asynchronous system. A synchronous system considers both processing and communication delays to be bounded; this is achieved by assuming *synchronous processes* and *synchronous communication*, described in more details below.

Synchronous process. A synchronous process always executes any step within a duration less than some known bound. A step consists of delivering a message received on some link (possibly nil), performing a local computation and sending a message to some other process (if required).

Synchronous communication. Synchronous communication means that there is a bound on message transmission delays. In other words, the time interval between the moment a message is sent and the moment that message is delivered to the destination process is always smaller than some known bound.

2.3.3 Partially Synchronous System

Systems in practice define and rely on some timing assumptions that are believed to be respected "most of time". However, there are some periods in which the system stops respecting these bounds and starts behaving asynchronously. These periods could occur as a result of network congestion or slow memory for instance.

The *partially synchronous* system captures this notion and can thus be viewed as a more realistic model than either synchronous or asynchronous systems. In partial synchrony either communication or processes or both, communication and processes, are partially synchronous [62]. We elaborate below on the meaning of partial synchrony both on the communication and the processes level.

Partially synchronous communication. Partial synchrony at the communication level means that one of the following two conditions holds [62]:

> *Condition 1.* There exists an upper bound δ on message delay respected at all times. However, δ is not known to the distributed algorithm and thus acceptable solutions here cannot assume any fixed value of δ at design.
>
> *Condition 2.* There exists an upper bound δ on message delay which might not be respected all the time; the message system is sometimes unreliable and thus may deliver messages late or not even at all. For each execution there is a time t, not known to any of the processes, after which the message delays respect δ from t onward. The communication system is said to respect the bound *eventually*. Note that in this case, violations of δ are not considered as process faults.

Partially synchronous processes. Similar to communication, processes are partially synchronous if the bound on processing speed, say Δ, either (i) exists but is unknown or (ii) only holds after some time t onward.

3 Networking Communication Models

In distributed computing, communication is abstracted by links. In this sense, the behavior exhibited by communication is fully encapsulated by link abstractions, such as the fair-loss links and reliable links (introduced in Section 2.2).

Modeling physical networking links and communication media is highly non-trivial due to its dependence on the underlying communication technology and the factors that affect it. Sometimes processes might even be connected by multiple heterogeneous (hybrid) communication media [32–34, 63, 64], for example a combination of wireless, Ethernet and power line communication. This makes modeling the communication behavior even more challenging.

However, what is of concern to this thesis, is specifically the behavior of traffic (messages) on physical networks that connect processes. Precisely, we are interested in the loss patterns that messages experience when sent over the network (which in turn affect message delays). Research in the networking community has observed, experimented and modeled the behavior of network traffic, in various settings. These studies are conducted under different communication media that are believed to become an important part of the future network infrastructure of smart grid, sensor and control systems [30–32, 37–40].

We review these message (packet) loss models focusing on three main models widely adapted in the literature of networking and communication [52, 65–69]: (i) Bernoulli loss model (ii) Gilbert-Elliot loss model, and (iii) higher order (4-state) Markov chain loss models.

Later, towards the end of this chapter, we provide a discussion comparing the networking communication models with the link abstractions of distributed computing, highlighting the differences and similarities between the two approaches for modeling communication.

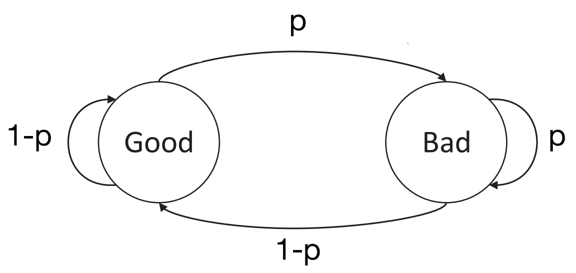

Figure 3.1 – The Bernoulli model for message (packet) loss.

3.1 Message Loss Models

3.1.1 Bernoulli Loss Model

The Bernoulli loss model is a simple discrete loss model where messages can be independently lost with an identical probability “p”.

Let us denote by $X_i \in \{0, 1\}$ the random variable stating if a message is lost or not at time i, i.e., $X_i = 1$ represents that fact that at time i the packet is lost while $X_i = 0$ the message is not lost. The probability of X_i being 0 or 1 is independent of all other values of the time series and is the same for all i. This model can be thus characterized by the single parameter p, which is the probability that $X_i = 1$.

In practice, the value of p can be estimated for a sample trace of loss events by:

$$p = \frac{n_1}{n},$$

where n_1 is the number of times the value $X_i = 1$ occurs in the time series and n is the total number of samples in the time series.

The Bernoulli model is also known as the *memory-less* model since message loss is independent of previous losses. In a high level view, as shown in Figure 3.1, the Bernoulli loss model can be seen as a two-state process in which one state (Good) represents a packet reaching the destination, and the other state (Bad) represents a packet loss.

3.1.2 Gilbert-Elliot Loss Model

The Gilbert-Elliot loss model is a discrete 2-state Markov chain.

Similar to the Bernoulli model, let us denote by $X_i \in \{0, 1\}$ the random variable stating if a message is lost or not at time i, i.e., $X_i = 1$ represents that fact that at time i the message is lost while $X_i = 0$ the packet is not lost.

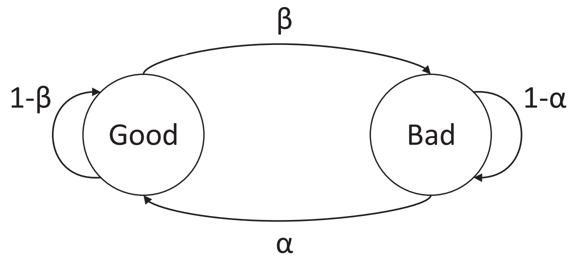

Figure 3.2 – The Gilbert-Elliot (GE) model for message (packet) loss.

The stochastic process X_i can exist in two states such that its current state depends only on its previous value, X_{i-1}. In contrast with the Bernoulli model, Section 3.1.1, the Gilbert-Elliot model (GE) is capable of capturing the dependence between consecutive losses and is characterized by two parameters α and β, as shown in Figure 3.2.

These parameters are known as the transition probabilities and are expressed as:

$$\alpha = P[X_i = 0|X_{i-1} = 1], \quad \beta = P[X_i = 1|X_{i-1} = 0].$$

In other words, if $X_{i-1} = 1$, with probability α, $X_i = 0$, and with probability $1 - \alpha$, $X_i = 1$. Similarly if $X_{i-1} = 0$ then, with probability β, $X_i = 1$ and with probability $1 - \beta$, $X_i = 0$.

In a sample trace, α and β can be approximated using the maximum likelihood estimators:

$$\hat{\alpha} = \frac{n_{01}}{n_0},$$

$$\hat{\beta} = \frac{n_{10}}{n_1},$$

where n_{01} is the number of times in the observed trace that 1 follows a 0 and n_{10} is the number of times 0 follows 1. n_0 is the number of 0s and n_1 is the number of 1s in the trace.

A Gilbert-Elliot model is said to be positively correlated (implying a bursty packet loss behavior) when $1 - \beta > \alpha$, while it said to be negatively correlated if $\alpha > 1 - \beta$.

3.1.3 4-State Markov loss Model

Finite-state Markov processes can capture a large variety of temporal dependencies. The Bernoulli model and the Gilbert-Elliot model are special cases of this class of models composed of only two states.

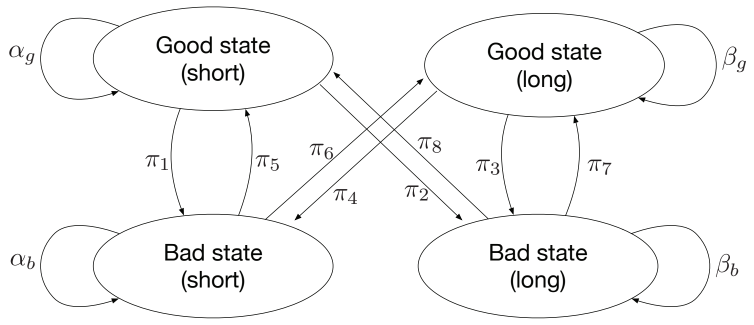

Figure 3.3 – The 4-state Markov model for message (packet) loss.

McDougall et. al [70] proposed a 4-state Markov model with two good and bad states. The goal was to be able to further specify a distribution of the duration of the good and the bad phases approximating an IEEE 802.11 channel [71]. The 2-state Markov models (Bernoulli and Gilbert-Elliot) can capture relation between consecutive messages losses and transmissions. Extending the number of states, as done by McDougall et. al [70], can allow us to model both consecutive losses and longer link failure events (that could last for few seconds [72]). In this sense the 4-state model can be seen as an extension of the Gilbert-Elliot model where loss bursts with term correlation, typically observed on error prone wireless channels, are well captured.

Figure 3.3 shows the 4-state model along with the transition probabilities between the different states [73]. The values of π_i are detailed below in terms of the system parameters $\alpha_g, \alpha_b, \beta_g, \beta_b, p_g, p_b$:

$$\begin{aligned}
\pi_1 &= (1-\alpha_g)p_b & \pi_5 &= (1-\alpha_b)p_g \\
\pi_2 &= (1-\alpha_g)(1-p_b) & \pi_6 &= (1-\alpha_b)(1-p_g) \\
\pi_3 &= (1-\beta_g)(1-p_b) & \pi_7 &= (1-\beta_b)(1-p_g) \\
\pi_4 &= (1-\beta_g)p_b & \pi_8 &= (1-\beta_b)p_g
\end{aligned}$$

3.2 Networking Versus Distributed Computing

In this section, we try to compare and contrast the networking view of communication with that of distributed computing. At a very high level, communication in all networking models, depicted in the previous section, can be seen as *probabilistic*. That is, a message sent at some point in time t can be lost, for example, with probability $p(t)$. Roughly speaking, the difference between the multiple networking models of communication is the way that $p(t)$ evolves (i) with time, (ii) relative to previous transmissions, and (iii) across links.

3.2.1 Similarity with Distributed Computing Communication Models

In the Bernoulli, the Gilbert-Elliot and the 4-state Markov chain loss models, a message sent at any point in time can be probabilistically lost. As such, a single message which is constantly re-transmitted may experience an unbounded number of losses. Hence, the delay for successfully (reliably) transmitting any message is unbounded (similar to the asynchronous communication assumption of distributed computing).

Also theoretically speaking, given this probabilistic view of losses, re-transmitting a message infinitely often would guarantee that this message is delivered successfully to the destination an infinite number of times, similar to the fair-loss links of distributed computing.

3.2.2 Difference with Distributed Computing Communication Models

The main difference, however, between the networking and the distributed computing view on communication, is that messages, from a networking perspective, can respect some delays probabilistically. In other words, as opposed to the asynchronous computing model where no bounds are assumed on communication delays, networking models imply that communication delays can respect some known bounds, however when and how long these bounds are respected is non-deterministic. This non-determinism also means that there is no point in time after which some unique bound on communication delay always holds, e.g., as considered in the partially synchronous model. This raises many questions concerning the practicality of solutions developed based on partial synchrony for example, when link characteristics in practice only hold temporarily (see Chapter 5).

As opposed to the distributed computing link abstractions, this networking view of communication can also permit the prediction and anticipation of message losses and thus of respective message delays. Such additional information about the behavior of messages, which cannot be achieved using classic distributed computing link models, is of substantial importance for contexts requiring real-time guarantees [10, 11, 14–18, 74] such as the ones of interest to this thesis.

This thesis exploits these factors that are not captured by the classic distributed computing links. We investigate the limitations, explore new abilities and rethink effective ways for applying existing distributed computing techniques to implement abstractions given the networking probabilistic view of communication.

Distributed Abstractions Part II

Being abstract is something profoundly different from being vague...
The purpose of abstraction is not to be vague, but to create a new semantic level
in which one can be absolutely precise.
— E. Dijkstra

In this part of the thesis, we introduce and discuss four fundamental distributed abstractions: (i) reliable energy-efficient communication (ii) probabilistic failure detection (iii) real-time group membership, and (iv) real-time distributed shared memory, respectively in Chapters 4, 5, 6 and 7.

We investigate how to implement these abstractions taking probabilistic losses into account. These losses reflect the networking view of how messages behave on physical communication media within systems, such as smart grids, sensor networks and distributed control systems. Besides providing robustness against losses, we also study how to guarantee crucial properties to the systems under consideration, such as energy efficiency and real-time behavior.

All chapters in this part follow a common structure. They start by motivating the abstraction under consideration presenting its benefits as well as the challenges posed by implementing such an abstraction in the context of specific real-world physical systems. After that, we present the system model in details followed by the chapter's central part, which demonstrates the major results. Finally, the chapters end with a discussion of existing work relative to the abstraction under study and with a summary that highlights the main results and contributions.

4 Energy-Optimal Synchronous Reliable Links

Communication is the very basic abstraction, besides processes, upon which all other distributed abstractions rely. This chapter discusses how to build a communication abstraction that provides important guarantees for upper layer abstractions like energy efficiency, reliability and synchrony, while considering unreliability in the underlying communication media. Precisely, in this chapter we investigate how to obtain in an energy-efficient manner, a reliable communication service that is synchronous with high probability.

We consider a *Partially Observable Markov Decision Process* (POMDP) setting in which a communication link's transmission quality: (i) changes according to a classic Markovian model and (ii) can be only partially observed, through feedback relative to previous transmissions. We perform a thorough analysis under several variations of acknowledgment-based (*Ack/Nack*) feedback mechanisms.

Despite the general intractability of POMDPs, this chapter proves that a communication service with the desired guarantees, under reliable feedback, can be inexpensively implemented. We obtain closed form solutions specifying when to transmit over the link, which allows to derive an energy-optimal implementation, precisely by minimizing the number of re-transmissions. We also analyze in this chapter the impact of lossy feedback on implementing the required communication service. Considering multiple lossy feedback mechanisms, we namely illustrate that an easily implementable structure for the communication service can also be obtained, depending on the feedback mechanism itself.

4.1 Motivation

Unreliable asynchronous communication renders the design and analysis of distributed algorithms a challenging task [75–77]. Consequently, fault-tolerant distributed algorithms typically assume reliable or even synchronous (or partially synchronous) links [78–82].

However, in practice, message losses are unavoidable. In fact, all communication media are lossy to some extent, due to uncertainties stemming from various phenomena such as unpredictable system loads and physical properties of the media. For example, wireless and power line communication quality are influenced by path loss, fading, interference, switching of the power grid, activation of electrical equipment, etc [37–40]. Although message losses are common in wireless and power line networks, they actually exist everywhere [30, 31]. Due to the random occurrences of such inevitable phenomena, messages losses typically come and go over time. The communication between a pair of processes (abstracted by a communication link) thus experiences time-varying unreliability, i.e., changes in the quality of the communication link with time between lossy and reliable. In addition to being a problem by itself, time-varying unreliability induces asynchrony[1], as successful message transmission delays become hard to anticipate.

This chapter discusses how to mask such message losses through a communication service, that can be used by high level applications of the network. More precisely, we want to provide a communication service which guarantees that message transfer, over a time-varying unreliable communication link, is: (i) always reliable and (ii) synchronous with high probability. Moreover, we want to design our communication service in an energy-efficient manner. In environments such as sensor networks [41–49], some devices have battery-powered wireless and recharging may be tedious or even impossible. In order to prolong the lifetime of a network, energy-efficient algorithms are crucial [22–29, 50]. Distributed algorithms can thus transparently utilize this communication service to deliver energy-efficient services at the network level, e.g., to build an energy-efficient reliable broadcast or higher level abstracts.

As the link changes state between reliable and lossy (relative to the occurrence of various phenomena) the current state of the link might not be known to the sending process [51–54, 83, 84]. The sender can however benefit from the feedback regarding previous transmissions to guess the current state. It can thus make better decisions of when to transmit; for example to avoid transmitting when the link state is bad. Such adaptive transmission decisions employ link prediction to appropriately adjust the transmission rate to the varying link conditions. In short, *transmission policies* which tell the sending process when to transmit and when to withhold from transmitting can be devised.

A reliable communication can be achieved by re-transmitting a message until it has been received [85]. However, the rate at which the protocol attempts to re-transmit yields a trade-off between (i) low energy consumption, (ii) high throughput and (iii) low latency. At one extreme, while merely optimizing for throughput and latency favors transmitting at every possible opportunity, this scheme results in maximum waste of energy. Especially at times when the link might be in a “bad condition”, i.e., constantly losing messages, for a long duration. At the other extreme, optimizing solely for energy might lead to a throughput bottleneck and an overwhelming message latency. Given this trade-off, this chapter addresses the question of how to build the desired communication service optimally by first studying the fundamental question of *when to transmit*.

[1]Asynchrony means here that there is no upper bound on message transmission delays, while synchrony means that the bound always exists and is known.

In particular, we seek to solve the *optimal transmission policy* which defines when should a sender transmit messages in order to optimize a defined energy-throughput balance while favoring lower latency. We also address two corollary questions: (i) can optimal transmission policies guarantee reliability under all desired energy-throughput trade-offs? If not, under what energy-throughput trade-offs can reliability be ensured? (ii) how to provide synchronous guarantees with high probability, given unbounded time-varying message losses between a pair of processes?

Determining the optimal transmission policy in such a setting is an instance of partially observable Markov decision processes (POMDPs), known to be notoriously intractable [86]. Despite the challenge, this chapter investigates the optimal transmission policy under various *Ack/Nack* feedback schemes, which to our knowledge, covers all possible *Ack/Nack* feedback mechanisms.

We start first by describing in details the setting of the system under consideration.

4.2 System Overview

We study the communication between any two processes. One process, noted by S, needs to send messages to the other process noted by R (see Figure 4.1). We assume discrete time events denoted by $T_{sys} = \{t_1, t_2, t_3, ...\}$. A subset, $\mathcal{T}$, of these time events occur at S, where $t_i \in \mathcal{T}\ \forall i$ *odd* (i.e., $\mathcal{T} = \{t_1, t_3, t_5, ...\}$). The time interval between consecutive events in T_{sys} is an upper bound on the propagation delay over the link in a single direction. As a result, the time interval between consecutive time events in $\mathcal{T}$ is an upper bound on the round trip propagation delay over the link. We designate by the time events in $\mathcal{T}$ the instances at which S is allowed to use the link, if it desires.

4.2.1 Communication Link Behavior

As noted earlier in Section 4.1 various inevitable phenomena such as unpredictable system loads and physical properties of the media result in message losses that typically come and go over time. This loss behavior induces changes in the quality of the communication link, which, as a consequence, switches with time between lossy and reliable. To capture the time-varying message losses of a communication link, we consider a widely-used approach for such cases, the Gilbert-Elliot (GE) model [67, 87], (also introduced earlier in Section 3.1.2). The GE model, consisting of two states (see Fig. 4.1), is a simple non-trivial finite state Markov chain (FSMC) [88], established to capture well message loss behavior resulting from randomly occurring phenomena [65, 89, 90]. In fact, the GE model has been empirically verified, by a large body of work [52, 65–69], as a good approximation of message losses in real-life communication scenarios. The GE model, for instance, has been used to model losses in wireless media IEEE 802.11 [69], wired power line networks [32] and other hybrid networks [30, 31]. The two states of the GE model (Fig. 4.1), noted by *good* and *bad*, can for example abstract the following: the communication link between

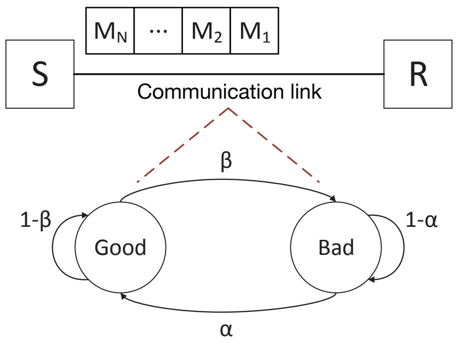

Figure 4.1 – A time-varying communication link under the 2-state GE model

a pair of processes occupies the bad state when the packet success-rate drops below a certain "unacceptable" threshold and the good state otherwise. The cause for these state transitions can, itself, lie in the random phenomena leading to message losses [65].

At any point in time, the link can be in one of the two states: the good state or the bad state. The link ***transitions*** with time, i.e., the link moves to its new state, which can be the same state it existed in or the other state. The time instants at which the link transitions are known as the ***transition times*** of the link. For example, given the link is in the good state at some point in time, it will remain in the same state at the next transition time with probability $1-\beta$ and will move to the other state with probability β. Similarly if the link state is bad at some point in time, it will remain bad at the next transition time with probability $(1-\alpha)$ and will shift to good with probability α. The link state remains fixed in the interval separating the transition times. We assume that S knows the parameters of the link, i.e., α and β (in practice S can estimate these parameters, see for example [91]).

If S sends a message m at some time $t_i \in \mathcal{T}$ and the link is in the good state in the interval $[t_i, t_{i+1}[$, then m is received by R at time $t\ :\ \ t_i < t < t_{i+1}$ (i.e., before t_{i+1}). We say R ***receives m by time*** t_{i+1}. If however the link state is bad, the transmission of message m fails, in which case S will retry to send this same message m in the following time unit in $\mathcal{T}$. Consequently, transmitting a message might span several time units (depending on the state of the link). Meanwhile, new messages (whether from outside or generated by S itself) that may arrive to S will be enqueued. We assume a FIFO queue where a message is dequeued only when it is successfully acknowledged. If a message transmission fails, the respective message remains at the top of the queue. Practically, this queue should be of a finite size. We thus assume that the rate at which new messages arrive to S, relative to the rate at which messages are dequeued, amounts to having at all times a non-empty queue[2] of size at most N. In practice and with the help of queuing theory, N can be chosen such that the probability of the queue overflowing is extremely small ($\approx$negligible). Guaranteeing a non-empty queue can be achieved by having S generate dummy messages whenever needed, which then serve to probe the link state.

[2]This is otherwise known as the infinite backlog assumption resulting in infinite messages to send.

4.2.2 Transmission Feedback

We investigate the optimal transmission scheme which determines the time instances in $\mathcal{T}$ when S should transmit a message over the link such that some defined reward function is maximized (see Section 4.2.3 for defining that function). We study optimal transmission schemes considering different types of feedback from the receiver side, precisely four $Ack/Nack$ feedback mechanisms. These four feedback mechanisms constitute, to the best of our knowledge, all possible variations of feedback mechanisms that are based on sending receipt acknowledgments relative to message transmissions.

Perfect Feedback

This feedback mechanism allows the sender to know what was the link state in the last carried transmission. Practically, such mechanism corresponds to one of two assumptions: (i) the transmission of the message and the acknowledgment happen in the same time slot under the same link condition (if the message is received successfully so is the ack) or (ii) both error-free sensing of the link and message transmission complete within one time slot. An example for (ii) is sensing applied in cognitive radio contexts (roughly implies “probing” the link). To achieve multi-channel opportunistic access, secondary users, typically, always apply sensing before attempting to transmit. Previous work such as [54, 92, 93] hinge on error-free sensing to solve problems (among many others). To implement this feedback scheme, we assume that the link transitions at every time instant in $\mathcal{T}$ only. When S sends a message m at some time $t_i \in \mathcal{T}$ and the link is in the good state, m will be successfully propagated to R. R then directly replies with an ack, which will be received by S before the next time instant in $\mathcal{T}$, i.e., t_{i+2}. However, if the link state is bad, transmission fails and R receives nothing. S is thus informed about the success of the last message transmission, and simultaneously about the last link state, by the presence or absence of an acknowledgment from R.

Next we describe various mechanisms in which feedback, sent by R to S, can be lost.

Constant Feedback

In this mechanism, the sender expects to receive periodical feedback from the receiver about the link state. However, this feedback can be lost. This mechanism is analogous to sensing the link on a periodical basis. We achieve this feedback by assuming that the link transitions at every time instant in T_{sys}. We also assume that R has access to the time instants $\mathcal{T}_{rcv} = \{t_2, t_4, t_6, ...\}$. R thus sends a message to S at every time instant in $\mathcal{T}_{rcv}$ regardless if S has sent something or not. The message sent by R is an ack if some message from S is received and is a $nack$ otherwise. As a result, messages sent by R can be lost independently of those sent by S.

Smart Feedback

In this mechanism, the sender expects feedback, despite potential losses, every time it transmits over the link. The sender process S, hence, can always know the state of the link after every message transmission, but might not be sure about the fate of the message sent, i.e., if that message was lost or not. *Smart feedback* can be achieved similar to *constant feedback*, except that we consider now that R knows the times at which S sends a message. As such R will not send anything when it's not expecting to receive a message from S. The assumption that R knows the sending times of S can be easily satisfied under deterministic sending schemes, e.g., constantly transmitting over the link. Later in Section 4.5 we show that indeed such a sending scheme (constantly transmit) could be the optimal scheme adopted by S.

Unreliable Feedback

The sender in this feedback mechanism is not sure if it can obtain feedback even when it transmits over the link. This mechanism is analogous to a sensing service which is not available all the time. To implement this mechanism, we also assume that the link transitions at every time instant in T_{sys}. As noted in *constant feedback*, having the transition times in T_{sys} means that messages sent by R can be lost independently of those sent by S. The receiver R sends an ack message to S only when it receives a message from S; otherwise R sends nothing.

4.2.3 Cost Assignment

At each time instant in $\mathcal{T}$, S can either (i) use the link to transmit a message or (ii) idle transmission. Both transmission and idling incur energy costs. We assume that a message transmission incurs a cost (negative reward) of $c_p(\leq 0)$, while idling incurs a cost of $c_d(\leq 0)$. It is obvious that in practice $c_p < c_d$ (since the idling energy cost is at least one order of magnitude less than sending in wireless sensor networks, see e.g., [94]), otherwise the optimal policy would be to always transmit. If a message is both transmitted and acknowledged successfully, S obtains an additional reward $r_s >\mid c_p \mid$. In this case, the total reward relative to a successful transmission is $r = r_s + c_p\ (> 0)$, while an unsuccessful or unacknowledged transmission gives no additional reward. This assignment of rewards and costs constitutes a generic function which allows to define any desired weighted balance between energy and throughput.

Illustration of Cost Assignment

Let us set the idling cost c_d to 0 and try to establish a throughput-energy balance based on the reward r_s and cost c_p. Assuming that messages sent by S are of maximum size, i.e., equivalent to the link capacity, then a reward of value $r_s = 1$ can be assigned. This reflects a maximum throughput, achieved by utilizing 100% of the link capacity in every time unit where transmission is successful. The amount of energy needed to transmit these fixed size messages is possible to obtain by measuring how much power is dissipated in the required transmission period. Assuming that S is a battery operated process[3], we can calculate the average percentage energy consumption relative to a single transmission (guaranteed to be < 1), which can in turn be assigned as the value of the energy cost c_p.

4.3 Optimal Transmission Formulation

In this section, we formally define our first question concerning how our communication service can be made energy optimal. In other words, we define mathematically the problem of determining when to transmit over the link so that a defined cost function is optimized. For simplicity we consider throughout the rest of the paper $t = \{t_i : t_i \in \mathcal{T}\}$ and $t + 1 = t_{i+2}$, i.e., the next time instant in $\mathcal{T}$. Let a_t be the action taken a time t. $a_t = 1$ ($a_t = 0$) corresponds to transmitting (idling) at time t respectively. The transmission policy, π, will then be the set of all decisions to be taken, i.e., $a_t\ \forall t$. We denote by o_t the feedback received by S (precisely from R) by time t. If nothing is received by time t then $o_t =\perp$. Let $\mathcal{R}(a_t, o_{t+1})$ be the reward obtained at time t relative to action a_t and the corresponding feedback relative to action a_t (which is obtained by time $t + 1$).

$$\mathcal{R}(a_t, o_{t+1}) = \begin{cases} r & a_t = 1, o_{t+1} = ack, \\ c_p & a_t = 1, o_{t+1} = nack \vee \perp, \\ c_d & a_t = 0 \end{cases}$$

Under all feedback mechanisms, S can make probabilistic guesses about the link state. Consequently, a conditional probability that the link state is good given the last received feedback from R, can be maintained by S at all times in $t \in \mathcal{T}$. This conditional probability is called the link belief w_t. We compute the link belief under each feedback mechanism.

Perfect Feedback. Under *perfect feedback*, if S sends a message at time t then at time $t + 1$, S will know what was link state in $[t, t + 1[$. Accordingly the link belief is updated at the end of every time $t \in \mathcal{T}$ as follows:

$$w_{t+1} = \begin{cases} 1 - \beta & a_t = 1, o_{t+1} = ack, \\ \alpha & a_t = 1, o_{t+1} =\perp, \\ \tau(w_t) = (1 - \beta)w_t + \alpha(1 - w_t) & a_t = 0. \end{cases}$$

[3]Sensor networks in a variety of contexts such as building automation and smart grids have finite energy sources or intermittent energy harvesting scenarios.

Constant Feedback. At all times $t \in \mathcal{T}$, S can know the exact last state of the link before t. Consequently, the link belief w is updated at every time t as follows:

$$w_{t+1} = \begin{cases} 1-\beta & o_{t+1} = ack \ \vee nack, \\ \alpha & o_{t+1} = \perp, \end{cases}$$

Smart Feedback. S can know the exact previous state of the link at all times $t : a_t = 1$, i.e., the times at which S sends a message over the link. Consequently, the link belief w is updated at time instants t as follows:

$$w_{t+1} = \begin{cases} 1-\beta & a_t = 1, o_{t+1} = ack \ \vee nack, \\ \alpha & a_t = 1, o_{t+1} = \perp, \\ \tau^2(w_t) = \tau(\tau(w_t)) & a_t = 0. \end{cases}$$

Unreliable Feedback. S can know the exact previous state of the link at all times t such that $a_t = 1 \wedge o_{t+1} = ack$, i.e., the times at which S sends a message and receives an acknowledgment for that message. At all other times S can not be sure about the previous link state. Consequently, the link belief w is updated at time instants t as follows:

$$w_{t+1} = \begin{cases} 1-\beta & a_t = 1, o_{t+1} = ack, \\ \mathsf{T}(w_t) = \frac{\tau(\alpha) - \alpha w_t(2-2\beta-\alpha)}{1-w_t(1-\beta)} & a_t = 1, o_{t+1} = \perp, \\ \tau^2(w_t) = \tau(\tau(w_t)) & a_t = 0. \end{cases}$$

We now show how $\mathsf{T}(w_t)$ is derived. By Bayes' Theorem,

$$\mathsf{T}(w_t) = Pr(s_{t+1} = G | w_t, a_t, o_{t+1} = \perp) = \frac{Pr(s_{t+1} = G, w_t, a_t, o_{t+1} = \perp)}{Pr(o_{t+1} = \perp) | w_t, a_t)} \tag{4.1}$$

where s_t is the link state at time t and

$$Pr(s_{t+1} = G, w_t, a_t, o_{t+1} = \perp) = \sum_{s_t} Pr(s_{t+1} = G | s_t, a_t, o_{t+1} = \perp) Pr(o_{t+1} = \perp | s_t, a_t) Pr(s_t, w_t). \tag{4.2}$$

This results in:

$$\mathsf{T}(w_t) = \frac{\tau(\alpha) - \alpha w_t(2-2\beta-\alpha)}{1-w_t(1-\beta)}.$$

We want to favor lower message latency while maximizing the defined energy-throughput cost function (Section 4.1), i.e., we consider a delay sensitive communication. Accordingly, the performance measure we seek to maximize is the expected total *discounted* reward. The *discounting factor* is a constant denoted by γ, such that $0 < \gamma < 1$. This γ can be roughly thought of as a penalty for delay. For a practical choice of γ, one should note that γ weights future rewards. Thus, a smaller γ should be chosen for more delay-sensitive applications, as it puts more emphasis on early transmissions. The expected total discounted reward can be formally written as:

$$E_{\pi}\left[\sum_{j=0}^{\infty}\gamma^{j}\mathcal{R}(a_{t_{j+1}},o_{t_{j+3}})|w_0\right], \tag{4.3}$$

where w_0 is the initial belief. The objective is to obtain the maximum expected total discounted reward that can be incurred from transmitting over a single link, also known as the value function $V_\gamma(w)$. Let $V_\gamma(w;a=1)$ (and analogously $V_\gamma(w;a=0)$) designate the expected total discounted reward from transmitting (not transmitting) on the link in the first decision followed by the optimal decisions in future times. Due to POMDP theory, the value function satisfies the Bellman equation and thus $V_\gamma(w)$ can be written as [52]:

$$V_\gamma(w)=\max\{V_\gamma(w;a=1),V_\gamma(w;a=0)\}.$$

4.4 Optimal Transmission Strategies

For presentation simplicity we first derive the value function and study the optimal transmission policy under the *perfect feedback* mechanism. We study the optimal transmission policy under the other feedback mechanisms later in Section 4.6.

Transmitting over the link yields an immediate expected reward of:

$$w\mathcal{R}(a_t=1,o_{t+1}=ack)+(1-w)\mathcal{R}(a_t=1,o_{t+1}=\perp)=wr+(1-w)c_p=w(r-c_p)+c_p.$$

The future maximum expected total discounted reward, relative to transmitting over the link, will be either: (i) $\gamma V_\gamma(1-\beta)$ (if the current state is good) or (ii) $\gamma V_\gamma(\alpha)$ (if the current state is bad). The former occurs with probability w while the latter occurs with probability $1-w$ resulting in:

$$V_\gamma(w;a=1)=w(r-c_p)+c_p+\gamma[wV_\gamma(1-\beta)+(1-w)V_\gamma(\alpha)].$$

Idling however yields an immediate expected cost of c_d (since no other cost/reward exists relative to idling the link). By the update function of the link belief (Section 4.3), w deterministically evolves to $\tau(w)$ as a result of not using the link. The consequent future maximum expected total discounted reward is $V_\gamma(\tau(w))$ occurring with probability 1. Hence,

$$V_\gamma(w;a=0)=c_d+\gamma V_\gamma(\tau(w)).$$

The value function, $V_\gamma(w)$, can be recursively written as:

$$\begin{aligned} V_\gamma(w) &= \max\{V_\gamma(w; a = 1), V_\gamma(w; a = 0)\} \\ &= \max\{w(r - c_p) + c_p + \gamma[wV_\gamma(1 - \beta) + (1 - w)V_\gamma(\alpha)], c_d + \gamma V_\gamma(\tau(w))\}, \end{aligned} \tag{4.4}$$

We distinguish between the different link types and determine the optimal transmission strategy in each case. The link can be categorized, based on its transition probabilities, as either being *memoryless* or *not*. The latter itself is subdivided into two categories: *positively correlated* ($1 - \beta > \alpha$) and *negatively correlated* ($1 - \beta < \alpha$).

4.4.1 Memoryless link

The link is memoryless when the probability of being in either state at the next time step is independent from current state, i.e., $1 - \beta - \alpha = 0$. As a consequence, $1 - \beta = \alpha = \tau(w) = p$. The value function in (4.4), thus reduces to

$$V_\gamma(w) = \max\{w(r - c_p) + c_p + \gamma V_\gamma(p), c_d + \gamma V_\gamma(p)\} = \max\{w(r - c_p) + c_p, c_d\} + \gamma V_\gamma(p).$$

The optimal transmission policy hence depends merely on the values of $w(r - c_p) + c_p$ and c_d:

$$\text{optimal policy} = \begin{cases} \text{transmit} & \text{if } w > \frac{c_d - c_p}{r - c_p}, \\ \text{idle} & \text{otherwise.} \end{cases}$$

Since $1 - \beta = \alpha = \tau(w) = p$, w will have a constant value for a given link. The optimal policy thus is either: (i) transmit on the link at every t or (ii) never transmit on the link.

4.4.2 Link with Memory

It is well established that the value function can be obtained by value iteration as a uniform limit of cost functions for finite horizon problems, which are continuous, piecewise linear and convex [51, 93]. The uniform convergence follows from the discounted dynamic operator being a contraction mapping [93]. As a consequence of uniform convergence, $V_\gamma(w)$ is a convex function in w continuous on [0,1].

Lemma 1. *If $c_d < w(r - c_p) + c_p$ (in particular if $c_d < c_p$), then the optimal decision is to use the link for transmission at every t.*

Proof. By convexity of $V_\gamma(w)$ in w we have:

$$V_\gamma(\tau(w)) = V_\gamma(w(1 - \beta) + \alpha(1 - w)) \leq wV_\gamma(1 - \beta) + (1 - w)V_\gamma(\alpha).$$

$V_\gamma(w; a = 1)$ is greater than $V_\gamma(w; a = 0)$, if $c_d < w(r - c_p) + c_p$, i.e., if $c_d - c_p < w(r - c_p)$.

Given that $w(r - c_p) \geq 0$, $V_\gamma(w; a = 1) > V_\gamma(w; a = 0)$ if $c_d - c_p < 0$. □

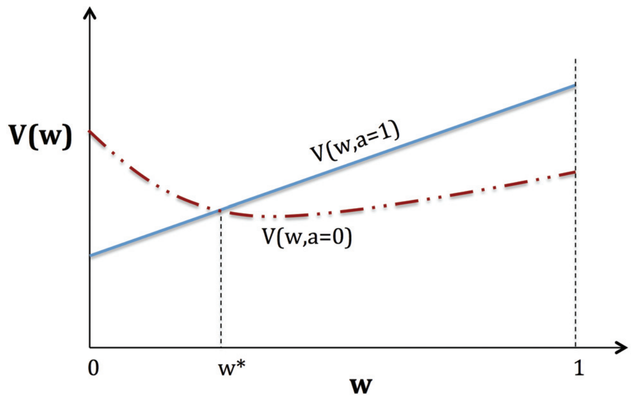

Figure 4.2 – The behavior of the value functions for transmitting and idling.

Lemma 2. *For a link with a defined cost function, there exists a unique value w^* such that $V_\gamma(w^*; a=1) = V_\gamma(w^*; a=0)$, $V_\gamma(w; a=1) < V_\gamma(w; a=0)$ $\forall w < w^*$ and $V_\gamma(w; a=1) > V_\gamma(w; a=0)$ $\forall w > w^*$.*

Proof. For $w = 0$:

$V_\gamma(0; a=1) = c_p + \gamma V_\gamma(\alpha)$ and $V_\gamma(0; a=0) = c_d + \gamma V_\gamma(\alpha)$. From Lemma 1, $c_d < c_p$ trivializes the optimal policy to that which constantly transmits over the link. We thus consider $c_d > c_p$ which yields $V_\gamma(0; a=1) < V_\gamma(0; a=0)$.

For $w = 1$:

$V_\gamma(1; a=1) = r + \gamma V_\gamma(1-\beta)$ and $V_\gamma(1; a=0) = c_d + \gamma V_\gamma(1-\beta)$. Since $r > 0$ and $c_d \leq 0$, we get $V_\gamma(1; a=1) > V_\gamma(1; a=0)$.

It can be seen that $V_\gamma(w; a=1)$ is linear in w. Following from the convexity of $V_\gamma(w)$, we can conclude that $V_\gamma(w; a=0)$ is convex in w. As a result, there exists a single intersection point between $V_\gamma(w; a=1)$ and $V_\gamma(w; a=0)$, where the implication w^* is unique comes from. This leads to the graph shown in Figure 4.2 concluding the proof. □

As a direct consequence of Lemma 2 the optimal policy has the following structure:

$$\text{optimal policy} = \begin{cases} \text{transmit} & \text{if } w > w^*, \\ \text{idle} & \text{if } w < w^*. \end{cases}$$

We compute the value of w^* by distinguishing between positively and negatively correlated links. Depending on the possible position of w^* with respect to α, $1-\beta$ and π_g (the stationary probability of being in the good state, $\pi_g = \frac{\alpha}{\alpha+\beta}$), w^* takes different values and admits different closed form expressions (refer to Appendix A). However, this requires first to guess the position of w^* with respect to α, $1-\beta$ and π_g for a given link and given cost assignment. Not knowing the position of w^* in the general case, we define the optimal policy in terms of costs. The computed closed form expressions of w^*, for all cases, depend on the cost c_d. More precisely, these closed forms show that w^* is strictly increasing in c_d (refer to Appendix A). We substitute the given fixed cost c_d by an unknown cost $C(w)$. We let $C(w)$ be the cost such that $w^* = w$. In other words, $C(w)$ is the idling cost under which $V_\gamma(w; a = 1) = V_\gamma(w; a = 0)$.

Lemma 3. *There exists a unique cost $C(w)$ such that $V_\gamma(w; a = 1) = V_\gamma(w; a = 0)$, where*

$$V_\gamma(w; a = 1) = w(r - c_p) + c_p + \gamma[wV_\gamma(1 - \beta) + (1 - w)V_\gamma(\alpha)],$$
$$V_\gamma(w; a = 0) = C(w) + \gamma V_\gamma(\tau(w)).$$

Proof. By Lemma 2 there is a unique intersection point (w^*) between $V_\gamma(w^*; a = 1)$ and $V_\gamma(w^*; a = 0)$. Since w^* is strictly increasing in the idling cost c_d (refer to Appendix A), then no two or more distinct idling costs can lead to the same w^*, which concludes the proof. □

The closed form expressions of $C(w)$ can then be obtained by simply inverting the closed form expressions of w^* and setting $w^* = w$ and are shown below.

Positively Correlated Channels.

1. If($w \geq 1 - \beta$ or $w \leq \alpha$)
$$C(w) = w(r - c_p) + c_p.$$

2. If($\pi_g \leq w < 1 - \beta$)
$$C(w) = \frac{w(r - c_p(1 - \gamma)) + c_p(1 - \gamma(1 - \beta))}{1 - \gamma(1 - \beta - w)}.$$

3. If($\alpha < w < \pi_g$)
$$C(w) = \frac{rA(k) - B(k)(1 - w)}{A(k) - D(k)(1 - w)}.$$

where $k = \lceil \frac{\ln(1 - \frac{w}{\pi_g})}{\ln(1 - \beta - \alpha)} \rceil - 2$,
$A(k) = \frac{(1 - \gamma^{k+2})(1 - \gamma(1 - \beta)) + \gamma^{k+2}(1 - \gamma)\tau^{k+1}(\alpha)}{1 - \gamma(1 - \beta - \alpha)}$,
$B(k) = r(1 - \gamma^{k+2}) - c_p(1 - \gamma)$,
$D(k) = \gamma(1 - \gamma^{k+1})$.

Negatively Correlated Channels.
Case: $1 - \beta = 0; \alpha = 1$

1. If($w > 0.5$)
$$C(w) = \frac{[r(1 - \gamma) - c_p(1 - \gamma^2)]w + \gamma r + c_p(1 - \gamma^2)}{1 + \gamma - \gamma^2 - \gamma(1 - \gamma)w}.$$

2. If$(w < 0.5)$
$$C(w) = \frac{(1-\gamma^2)[r - c_p(1+\gamma)]w - \gamma^3 r + c_p}{1 - \gamma(1-\gamma^2)w}.$$

Case: $-1 < 1 - \beta - \alpha < 0$

1. If$(w \geq \alpha$ or $w \leq 1 - \beta)$
$$C(w) = w(r - c_p) + c_p.$$

2. If$(\tau(1-\beta) \leq w < \alpha)$
$$C(w) = \frac{[(1-\gamma)r - c_p]w + \gamma\alpha r + c_p}{(1+\gamma\alpha) - \gamma w}.$$

3. If$(\pi_g \leq w < \tau(1-\beta))$
$$C(w) = \frac{\gamma\alpha r + c_p(1-\alpha\gamma^2+\gamma^2(1-\beta)(\alpha-(1-\beta))) + (1-\gamma)(r-c_p(1+\gamma))w}{1+\alpha\gamma(1-\gamma)+\gamma^2(1-\beta)(\alpha-(1-\beta))-\gamma(1-\gamma)w}.$$

4. If$(1 - \beta < w < \pi_g)$
$$C(w) = \frac{w(r-(1+\gamma)c_p) + c_p(1+\gamma(1-\beta))}{1+\gamma(1-\beta-w)}.$$

Lemma 4. *$C(w)$ is strictly increasing in w.*

Proof. From Appendix A, w^* is strictly increasing in c_d, which means that c_d is also strictly increasing in w^* (inverting the relation preserves the monotonicity). But the expressions for $C(w)$ are obtained by replacing c_d by $C(w)$ and setting $w^* = w$, which concludes the proof. $\square$

Theorem 1. *The optimal policy for a link with memory under a given cost assignment is:*

$$\textit{optimal policy} = \begin{cases} \textit{transmit} & \textit{if } c_d < C(w), \\ \textit{idle} & \textit{if } c_d \geq C(w). \end{cases}$$

Proof. For a link with a given idling cost c_d, there exists by Lemma 2 a unique value w^* which makes the action of transmitting on the link as equally attractive as that of idling transmission. More precisely $C(w^*) = c_d$. By Lemma 4, $C(w) > c_d$ for $w > w^*$. The optimal policy definition says to transmit if $w > w^*$ and idle otherwise. Thus $C(w) > c_d$ amounts to having the action of transmitting over the link as optimal. Similarly by Lemma 4, $C(w) < c_d$ for $w < w^*$, which means that idling transmission is optimal. $\square$

4.5 Optimal Reliable Transmission

Still considering *perfect feedback*, we investigate in this section how to guarantee reliability using optimal transmission schemes. Clearly reliability is guaranteed if and only if optimal decisions do not suspend transmission endlessly.

Lemma 5. *Optimal transmission policies do not always guarantee reliability across a link be it memoryless or not.*

Proof. We prove the lemma for each case of link memory by illustrating a counter example showing endless suspension of transmission under optimal decisions.

Memoryless. In the memoryless case the optimal decision at every time instant t is transmit only if $w > \frac{c_d - c_p}{r - c_p}$; however $w = p = 1 - \beta = \alpha, \forall t > 0$. An assignment of $c_d = 0$ and $r = -c_p$ leads to $\frac{c_d - c_p}{r - c_p} = 0.5$. Hence any link satisfying $1 - \beta = \alpha < 0.5$ will have the optimal decision of always idling the link.

Positively correlated. One possible cost assignment could lead to the following relation being satisfied: $c_d \geq (1 - \beta)(r - c_p) + c_p$. Such an assignment can happen in cases where the link rarely resides in the good state and the energy costs are relatively high (e.g. $c_d = 0$, $r = -c_p$ and $1 - \beta = 0.3$). The expected reward relative to transmitting on the link is $w(r - c_p) + c_p$. Note that $1 - \beta \geq w \geq \alpha$ (a direct consequence of the belief update function in Section 4.3) and that $w(r - c_p) + c_p$ increases monotonically as w increases. The expected reward (c_d) relative to idling the link is thus always greater than the maximum expected reward relative to using the link. Hence attempting to transmit at any time will only make the value of the total expected discounted reward less.

Negatively correlated. As in the positively correlated case, a link which rarely resides in the good state and whose energy costs are relatively high (e.g. $c_d = 0$, $r = -c_p$ and $\alpha = 0.3$) may lead to having $c_d \geq \alpha(r - c_p) + c_p$ satisfied. Due to negative correlation, we have $1 - \beta \leq w \leq \alpha$ (following from the belief update function in Section 4.3). Attempting to transmit on the link at any time will yield an expected reward less than that obtained by idling the link. This makes the decision of idling the link at all times lead to the maximum value of the total expected discounted reward. □

Theorem 2. *In a memoryless link, i.e., a link with a constant probability, $1 - p$, of losing messages, reliability is guaranteed under an optimal policy only if $p > \frac{c_d - c_p}{r - c_p}$. The policy in this case is to constantly transmit on the link.*

Proof. The proof follows directly from the optimal policy of memoryless links in Section 4.4.1. □

Theorem 3. *Given that a positively correlated link is used at some time t, there are exactly 2 forms of optimal transmission policies at $[t+1, ..., \infty[$, capable of guaranteeing reliability.*

1. ***Constantly Transmit**: In case of high idling cost, keep sending, irrespective of the predicted link state ($a_{t+i} = 1\ \forall i$). This policy is optimal if $c_d < \alpha(r - c_p) + c_p$.*

2. ***Back-off on Bad**: Transmit as long as the observed link state is good. When the observed state is bad, transmission is withheld for T time instants (somehow wait until link is expected to transition to the good state) after which transmission resumes again.*

$$\mathsf{T} = \left\lfloor \frac{\ln \frac{(r-c_d)A(k)+(1-w_0)(c_d D(k)-B(k))}{w_0(B(k)-D(k))}}{\ln(1-\beta-\alpha)} \right\rfloor.$$

   This policy is optimal if $c_d < (1-\beta)(r-c_p) + c_p \wedge c_d \geq (\alpha)(r-c_p) + c_p$.

Proof. We split the possible search space of w^*, i.e. $[0; 1]$, into different regions. If $w^* \in [\pi_g, 1]$, and the link is observed in the bad state, i.e., $w = \alpha$, then the optimal policy will be to suspend transmission forever. This follows from the fact that (A.1) and (A.2) lead to $w \geq w^*$ never being satisfied. The only possible range for w^*, such that transmission is never suspended forever, is to exist in the region $[0, \pi_g]$. This range can be split in two: $[0, \alpha] \cup [\alpha, \pi_g]$.

Following from the update function of Section 4.3 the link belief eventually abides by:

$$1 - \beta \geq w \geq \alpha. \tag{4.5}$$

If $w^* \in [0, \alpha]$, then by (4.5), $w > w^*$ will always be satisfied and the optimal policy would be to always use the link. If $w^* \in [\alpha, \pi_g]$ then as long as the link is observed to be good ($w = 1 - \beta > w^*$) it would be optimal to use the link again.

However if the link is observed to be bad ($w = \alpha \leq w^*$) then it is optimal not to use the link until $w > w^*$ is satisfied, which by (A.1) happens in finite time T. By Theorem 1 $w > w^*$ for $w^* \in [\alpha, \pi_g]$ is equivalent to $c_d < \frac{rA(k)-B(k)(1-w)}{A(k)-D(k)(1-w)}$, where $w = \tau^{\mathsf{T}}(\alpha)$ for $\mathsf{T} \in [0, \infty]$. □

Theorem 4. *Given a negatively correlated link is used at some time t, there are exactly 2 forms for the optimal transmission policies at times $[t+1, ..., \infty]$, which are capable of ensuring reliability:*

1. ***Constantly Transmit**: This policy is optimal if $c_d < (1-\beta)(r-c_p) + c_p$.*

2. ***Skip if Good**: Transmits as long as the observed link state is bad. If the observed state is good transmission is withheld for the following time instant after which it resumes again. This policy is optimal if $c_d < \alpha(r-c_p) + c_p \wedge c_d \geq (1-\beta)(r-c_p) + c_p$.*

Proof. For the negatively correlated links [54]:

$$\tau^{2k}(w) \text{ and } \tau^{2k+1}(w) \to \pi_g, \text{ from opposite directions , as } k \to \infty. \tag{4.6}$$

If $w^* \in [\tau(1-\beta), 1]$ then transmission can be suspended forever. When the link is observed in the good state ($w = 1-\beta < w^*$) then it is optimal to idle the link. This results in the link never being used since from (4.6), $\tau^k(1-\beta) > w^*$, can never be satisfied.

Therefore, to guarantee reliability, we should have $w^* \in [0, \tau(1-\beta)]$, which can be split into $[0, 1-\beta] \cup [1-\beta, \tau(1-\beta)[$.

If $w^* \in [0, 1-\beta]$, by $1-\beta \leq w \leq \alpha$, $w > w^*$ is always satisfied and the optimal policy is to always use the link.

Now if $w^* \in [1-\beta, \tau(1-\beta)]$, then when $w = \alpha > w^*$ (i.e., the link is observed to be bad) it is optimal to use it again. If the link is good though, then $w = 1-\beta$ which is not greater than w^*, meaning it is optimal to idle the link. Consequently, in the following time instant, w will be updated to $w = \tau(1-\beta) > w^*$ and the optimal action is to transmit over the link. □

Implications of Theorems 3 and 4

The established theorems indicate that an optimal reliable transmission protocol continues to transmit after a successful (failed) transmission along a positively (negatively) correlated link. This same protocol, however, will wait for a fixed time, say T_{wait}, before attempting to send again after a failed (successful) transmission. Hence an optimal reliable protocol can be defined solely by the waiting time T_{wait} after a successful (failed) sending attempt.

4.6 Impact of Lossy Feedback

Recall that so far, and for presentation simplicity, we have considered *perfect feedback*. We now investigate the impact of lossy feedback on energy-optimal transmission policies by studying the system under the *constant feedback*, *smart feedback* and *unreliable feedback* mechanisms.

Transmitting over the link yields an immediate expected reward of:

$$\begin{aligned} &w(1-\beta)\mathcal{R}(a_{t_i}=1, o_{t_{i+2}}=ack) + \alpha(1-w)\mathcal{R}(a_{t_i}=1, o_{t_{i+2}}=nack) \\ &\qquad + (w\beta + (1-\alpha)(1-w))\mathcal{R}(a_{t_i}=1, o_{t_{i+2}}=\perp) \\ &\qquad = w(1-\beta)(r-c_p) + c_p. \end{aligned}$$

Idling however yields an immediate expected cost of c_d (since no other cost/reward exists relative to idling the link). Next we derive the value function relative for each of the other feedback mechanisms.

4.6.1 Constant Feedback

The future maximum expected total discounted reward, regardless if S transmits or idles the link, will be either: (i) $\gamma V_\gamma(1-\beta)$ (if an ack or a $nack$ is obtained) or (ii) $\gamma V_\gamma(\alpha)$ (if a $\perp$ is obtained). The former occurs with probability $\tau(w)$ while the latter occurs with probability $1-\tau(w)$, resulting in:

$$\begin{aligned}V_\gamma(w;a=1) &= w(1-\beta)(r-c_p)+c_p+\gamma[\tau(w)V_\gamma(1-\beta)+(1-\tau(w))V_\gamma(\alpha)]\\ V_\gamma(w;a=0) &= c_d+\gamma[\tau(w)V_\gamma(1-\beta)+(1-\tau(w))V_\gamma(\alpha)].\end{aligned}$$

The value function $V_\gamma(w)$ can be thus recursively written as:

$$\begin{aligned}V_\gamma(w) &= \max\{V_\gamma(w;a=1),V_\gamma(w;a=0)\}\\ &= \max\{w(1-\beta)(r-c_p)+c_p+\gamma[\tau(w)V_\gamma(1-\beta)+(1-\tau(w))V_\gamma(\alpha)],\\ &\quad c_d+\gamma[\tau(w)V_\gamma(1-\beta)+(1-\tau(w))V_\gamma(\alpha)]\}.\end{aligned} \tag{4.7}$$

4.6.2 Smart Feedback

Under this feedback mechanism, the future maximum expected total discounted reward, relative to S transmitting over the the link, will be either: (i) $\gamma V_\gamma(1-\beta)$ (if an ack or a $nack$ is obtained) or (ii) $\gamma V_\gamma(\alpha)$ (if a $\perp$ is obtained). The former occurs with probability $\tau(w)$ while the latter occurs with probability $1-\tau(w)$.

$$V_\gamma(w;a=1) = w(1-\beta)(r-c_p)+c_p+\gamma[\tau(w)V_\gamma(1-\beta)+(1-\tau(w))V_\gamma(\alpha)]$$

However, when S does not use the link to send a message, w will deterministically shift to $\tau^2(w)$ resulting in:

$$V_\gamma(w;a=0)=c_d+\gamma V_\gamma(\tau^2(w)).$$

So, $V_\gamma(w)$ can be written as:

$$\begin{aligned}V_\gamma(w) &= \max\{V_\gamma(w;a=1),V_\gamma(w;a=0)\}\\ &= \max\{w(1-\beta)(r-c_p)+c_p+\gamma[\tau(w)V_\gamma(1-\beta)+(1-\tau(w))V_\gamma(\alpha)],c_d+\gamma V_\gamma(\tau^2(w_t))\}.\end{aligned} \tag{4.8}$$

4.6.3 Unreliable Feedback

Under this feedback mechanism, the future maximum expected total discounted reward, relative to S transmitting over the link, will be either:

- $\gamma V_\gamma(1-\beta)$ (if an ack is obtained) or

- $\gamma V_\gamma(\frac{\tau(\alpha)-\alpha w(2-2\beta-\alpha)}{1-w(1-\beta)})$ (if $\perp$ is obtained).

The former occurs with probability $w(1-\beta)$ while the latter occurs with probability $1-w(1-\beta)$.

$$\begin{aligned}V_\gamma(w; a=1) &= w(1-\beta)(r-c_p)+c_p \\ &\quad + \gamma\left[w(1-\beta)V_\gamma(1-\beta) + (1-w(1-\beta))V_\gamma(\frac{\tau(\alpha)-\alpha w(2-2\beta-\alpha)}{1-w(1-\beta)})\right].\end{aligned}$$

As in the *smart feedback* mechanism when S idles the link, $V_\gamma(w; a=1) = c_d + \gamma V_\gamma(\tau^2(w))$. So, $V_\gamma(w)$ can be written as:

$$\begin{aligned}V_\gamma(w) &= \max\{V_\gamma(w; a=1), V_\gamma(w; a=0)\} \\ &= \max\{w(1-\beta)(r-c_p)+c_p \\ &+ \gamma\left[w(1-\beta)V_\gamma(1-\beta) + (1-w(1-\beta))V_\gamma(\frac{\tau(\alpha)-\alpha w(2-2\beta-\alpha)}{1-w(1-\beta)})\right], c_d + \gamma V_\gamma(\tau^2(w))\}.\end{aligned} \tag{4.9}$$

4.6.4 Optimal Transmission Policies Under Lost Feedback

Having obtained the value function $V_\gamma(w)$, we investigate the structure of the reliable energy-optimal transmission policy under the different feedback mechanisms.

Constant Feedback

By observing (4.7), we can note that the maximum future total expected discounted reward for using the link and idling it is the same (denote it by F). Thus:

$$\begin{aligned}V_\gamma(w) &= \max\{V_\gamma(w; a=1), V_\gamma(w; a=0)\} \\ &= \max\{w(1-\beta)(r-c_p)+c_p+F, c_d+F\}.\end{aligned} \tag{4.10}$$

Theorem 5. *A myopic (greedy) threshold policy is the energy-optimal transmission policy that guarantees reliable transmission under a constant feedback mechanism when* $(1-\beta)^2 > \frac{c_d-c_p}{r-c_p}$.

Proof. From (4.10), it is clear that the value function depends only on the immediate expected reward. The optimal policy can be stated as:

$$\text{optimal policy} = \begin{cases} \text{transmit} & \text{if } w > \frac{c_d-c_p}{(1-\beta)(r-c_p)}, \\ \text{idle} & \text{otherwise.}\end{cases}$$

By the update belief function under *constant feedback* (Section 4.2), the optimal policy will never suspend transmission forever if $(1-\beta)^2 > \frac{c_d-c_p}{r-c_p}$. □

Smart Feedback

As in *perfect feedback* (Section 4.4), $V_\gamma(w)$ for *smart feedback* can be shown to be a convex function in w continuous on $[0, 1]$.

Lemma 6. *If $c_d < w(1-\beta)(r-c_p) + c_p$ (in particular if $c_d < c_p$), then the optimal decision is to use the link for transmission at every t.*

Proof. By convexity of $V_\gamma(w)$ in w we have

$$V_\gamma(\tau^2(w)) = V_\gamma(\tau(w)(1-\beta) + \alpha(1-\tau(w))) \leq \tau(w)V_\gamma(1-\beta) + (1-\tau(w))V_\gamma(\alpha).$$

$V_\gamma(w; a=1)$ is greater than $V_\gamma(w; a=0)$, if $c_d < w(1-\beta)(r-c_p) + c_p$, i.e., if $c_d - c_p < w(1-\beta)(r-c_p)$. Given that $w(1-\beta)(r-c_p) \geq 0$, $V_\gamma(w; a=1) > V_\gamma(w; a=0)$ if $c_d - c_p < 0$.

$\square$

Theorem 6. *The energy optimal transmission policy under a smart feedback mechanism is a threshold policy, i.e., $V_\gamma(w; a=1) < V_\gamma(w; a=0)$ $\forall w < w^*$ and $V_\gamma(w; a=1) > V_\gamma(w; a=0)$ $\forall w > w^*$ for a unique w^*, only if $c_d - c_p \geq \gamma[\tau(\alpha)V_\gamma(1-\beta) + (1-\tau(\alpha))V_\gamma(\alpha) - V_\gamma(\tau(\alpha))] \wedge r > \frac{c_d - \beta c_p}{1-\beta}$.*

Proof. At $w = 0$:
$V_\gamma(0; a=1) = c_p + \gamma[\tau(\alpha)V_\gamma(1-\beta) + (1-\tau(\alpha))V_\gamma(\alpha)]$ and $V_\gamma(0; a=0) = c_d + \gamma V_\gamma(\tau(\alpha))$. From Section 4.2 and Lemma 1 $c_d > c_p$ (since $c_d < c_p$ trivializes the optimal policy to that which constantly transmits over the link). By convexity of $V_\gamma(w)$ in w, $V_\gamma(\tau(\alpha)) \leq \tau(\alpha)V_\gamma(1-\beta) + (1-\tau(\alpha))V_\gamma(\alpha)$. Thus, if $c_d - c_p < \gamma[\tau(\alpha)V_\gamma(1-\beta) + (1-\tau(\alpha))V_\gamma(\alpha) - V_\gamma(\tau(\alpha))]$, then $V_\gamma(0; a=1) < V_\gamma(0; a=0)$; otherwise $V_\gamma(0; a=1) > V_\gamma(0; a=0)$.

At $w = 1$:
$V_\gamma(1; a=1) = r(1-\beta) + \beta c_p + \gamma V_\gamma(1-\beta)$ and $V_\gamma(1; a=0) = c_d + \gamma V_\gamma(1-\beta)$. Consequently, if $r > \frac{c_d - \beta c_p}{1-\beta}$, then $V_\gamma(1; a=1) > V_\gamma(1; a=0)$; otherwise $V_\gamma(1; a=1) < V_\gamma(1; a=0)$. From (4.8), it can be seen that $V_\gamma(w; a=1)$ is linear in w. Following from the convexity of $V_\gamma(w)$, we can conclude that $V_\gamma(w; a=0)$ is convex in w. This leads to a graph as that shown in Figure 4.2 concluding the proof. $\square$

Closed form solutions and conditions relative to attaining reliability can be obtained similar to those of *perfect feedback* Section 4.5. We do not elaborate further on these conditions since an analogous analysis has been already conducted in details (Section 4.5).

Unreliable Feedback

We analyze *unreliable feedback* for positively correlated links only.

Lemma 7. *Under the unreliable feedback mechanism the link belief is* $w \in [0, 1]$.

Proof. w at any time can take one of the following values: (i) $1 - \beta$, (ii) $\mathsf{T}(w)$ or (iii) $\tau(\tau(w))$. [54] shows that $\tau^n(w) \forall\ w$, tends to $\frac{\alpha}{\alpha+\beta}$ as $n \longrightarrow \infty$ ($\tau^n(w)$ means that the function τ is called n times on w). It can be easily shown that $\mathsf{T}(w)$ is a decreasing function in w such that $0 < \mathsf{T}(1), \mathsf{T}(0) < 1$. □

Lemma 8. $\mathsf{T}(w)$ *is a convex function in* w.

Proof. For $0 < w_1, w_2, \lambda < 1$:

$$\begin{aligned}\lambda\mathsf{T}(w_1) + (1-\lambda)\mathsf{T}(w_2) - \mathsf{T}(\lambda w_1 + (1-\lambda)w_2)&\\ = (1-\beta)^2\tau(\alpha)[\lambda(1-\lambda)(w_2 - w_1)^2]&\\ + (1-\beta)(2-2\beta-\alpha)[\lambda w_1(1 + w_1(2w_2(1-\beta)-1))&\\ + (1-\lambda)w_2(1 + w_2(2w_1(1-\beta)-1))] \geq 0.&\end{aligned}$$

□

The value function $V_\gamma(w)$ can be obtained by value iteration as a uniform limit of cost functions for finite horizon problems, which are continuous and convex [51, 93]. $V_\gamma(w)$, as the upper envelope of a family of straight lines (cost functions), is thus convex in w (proved by [95]).

Theorem 7. *For a link with a defined cost function under unreliable feedback, there will exist at least one value* w^* *such that* $V_\gamma(w^*; a = 1) = V_\gamma(w^*; a = 0)$ *only if* $r > \frac{cd-\beta c_p}{1-\beta}$.

Proof. At $w = 0$:
$V_\gamma(0; a = 1) = c_p + \gamma V_\gamma(\tau(\alpha))$ and $V_\gamma(0; a = 0) = c_d + \gamma V_\gamma(\tau(\alpha))$. Since $c_d > c_p$, then $V_\gamma(0; a = 1) < V_\gamma(0; a = 0)$ (recall from Section 4.2 that if $c_d < c_p$ the optimal action is to constantly transmit over the link).

At $w = 1$:
$V_\gamma(1; a = 1) = r(1-\beta) + \beta c_p + \gamma[(1-\beta)V_\gamma(1-\beta) + \beta V_\gamma(\alpha)$ and $V_\gamma(1; a = 0) = c_d + \gamma V_\gamma(\tau(1-\beta))$. By convexity of $V_\gamma(w)$ we have $V_\gamma(\tau(1-\beta)) \leq (1-\beta)V_\gamma(1-\beta) + \beta V_\gamma(\alpha)$. If $r > \frac{cd-\beta c_p}{1-\beta}$ we get $V_\gamma(1; a = 1) > V_\gamma(1; a = 0)$.

Both $V_\gamma(w; a = 0)$ and $V_\gamma(w; a = 1)$ depend on $V_\gamma(f(w))$ (where $f(w)$ is some function of w) and are convex (due to the convexity of $V_\gamma(w)\ \forall\ w$). $V_\gamma(w; a = 0)$ and $V_\gamma(w; a = 1)$ will thus intersect in at least a single point as shown in Figure 4.3 concluding the proof. □

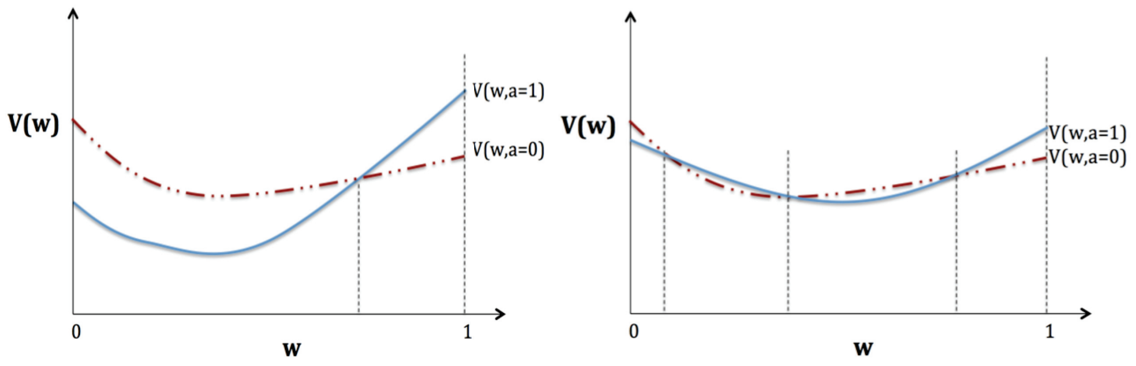

Figure 4.3 – Behavior of value functions for transmitting and idling under *unreliable feedback*.

The optimal policy under *unreliable feedback* is to transmit only in the intervals where $V_\gamma(w; a = 0) < V_\gamma(w; a = 1)$, which by Figure 4.3 may span multiple disjoint intervals over $w \in [0, 1]$. Consequently, a simple threshold policy (as that under the other feedback mechanisms) is not necessarily guaranteed, i.e., the optimal policy may not be a simple threshold policy.

4.7 Establishing Synchronous Communication

The transmission on a link is subject at any time to a non-zero probability of message loss, which may lead to a finite but unbounded delivery time for messages.

In this section, we show how synchronous communication over such a link can be guaranteed with high probability. For presentation simplicity, we carry our analysis for the optimal transmission under the *perfect feedback* mechanism. Similar analyses can be easily conducted for the other feedback schemes.

We determine the probability distribution of the total time, X, required to deliver the last message in the finite queue of size N. As such being able to guarantee with high probability that such a message is delivered within a specific time, say δ, implies that any message will get delivered in time $\leq \delta$.

More precisely, define the waiting time of the i^{th} message in the queue to be the time required for this message to reach the top of the queue. Let X_i designate the time to successfully transmit the i^{th} message in the queue, given it has zero waiting time. X_i is thus the time it takes message i to get from the top of sender's queue to the receiver's side. In queuing theory, X_i is known as the service time. Given N messages in the queue, $X = X_1 + X_2 + ... + X_N$. We are interested in the minimum δ such that $P[X < \delta] = 1 - \epsilon$. We show in what follows how to determine the value of δ for the link.

Lemma 9. *X_i $\forall i$, in a Gilbert-Elliot link model, are independent and identically distributed random variables, given the initial link belief $w_0 = 1 - \beta$.*

Proof. The following three facts apply to every message at the top of the queue:

1. If a message reaches the top of the queue at time $t \in \mathcal{T}$, then the link state at $[t-1, t[$ was good. A message only reaches the top of the queue if the previous message is no longer there, i.e., has been successfully transmitted.

2. Given that the link was good at $[t-1, t[$ the probability that the link state stays good or becomes bad is independent of the message. Thus for any message reaching the top of the queue, the probability distribution of the link being good is the same.

3. The optimal transmission policy is deterministic and independent of the message, but dependent on the state of the link, which by 1 and 2, is the same for every message reaching the top of the queue.

As a result the probability distribution of successfully transmitting a message is the probability distribution of the link being in the good state after being observed to be good some t time steps ago, where t by the third fact is only dependent on the policy. Due to the fact that the transmission policy does not change for a given link, neither do the link stochastic parameters, this probability distribution is fixed and thus independent and identical for every message. By assuming $w_0 = 1 - \beta$, the very first message will have identical distribution as well. □

The probability distribution of X, $f_X(k) = P[X = k]$, is obtained by the convolution of the distributions of X_i's.

$$f_X(k) = \sum_{k_1=1}^{k} \sum_{k_2=1}^{k} \dots \sum_{k_{N-1}=1}^{k} f_{X_1}(k_1) \cdot f_{X_2}(k_2) \dots f_{X_N}(k - \sum_{i=1}^{N-1} k_i)). \tag{4.11}$$

The minimum δ such that $P[X < \delta] = 1-\epsilon$ can be found by $\underset{\delta}{\operatorname{argmin}} \left\{ \delta : \sum_{k=1}^{\delta} f_X(k) >= 1 - \epsilon \right\}$. $f_X(k)$ for a general queue of size N is hard to express in a closed form. $f_X(k)$ can however be obtained offline by a simple algorithm implementing the function in (4.11). For theoretical interest, we alternatively obtain a closed form of an upper bound on δ.

Theorem 8. *The time to deliver all N messages in the queue with probability $1 - \epsilon$ is upper bounded by*

$$\delta = \left\lceil \frac{N \cdot E[X_i]}{\epsilon} \right\rceil .$$

Proof. The average waiting time of the N^{th} message is

$$E[X] = E[\sum_{i=1}^{N} X_i] = \sum_{i=1}^{N} E[X_i] = N \cdot E[X_i].$$

By Markov's inequality we have:

$$P[X > \delta] < \frac{E[X]}{\delta} = \frac{N \cdot E[X_i]}{\delta}.$$

Multiplying by -1 and then adding 1 to both sides of the inequality leads to

$$P[X \leq \delta] > 1 - \frac{N \cdot E[X_i]}{\delta}.$$

Since we need $P[X < \delta] = 1 - \epsilon$, then an upper bound on δ is:

$$\operatorname*{argmin}_{\delta} \left\{ \delta : 1 - \frac{N \cdot E[X_i]}{\delta} \geq 1 - \epsilon \right\},$$

which after simple calculation leads to $\delta = \left\lceil \frac{N \cdot E[X_i]}{\epsilon} \right\rceil$. $\square$

Next we compute closed form expressions of $f_{X_i}(k)$ and the average service time $E[X_i]$ for all optimal reliable policies.

4.7.1 Constantly transmit

A message reaches the top queue only if the message proceeding it gets successfully transmitted, inferring that the link state was good. Since this policy always transmits, a message arriving to the top of the queue at time t gets successfully transmitted at $t + 1$ with probability $1 - \beta$ (i.e., link is good at $t + 1$), at $t + 2$ with probability $\beta\alpha$ (i.e., link is bad at $t + 1$ and good at $t + 2$), so on and so forth. The probability distribution of X_i is:

$$f_{X_i}(k) = P[X_i = k] = \begin{cases} 1 - \beta & if\ k = 1, \\ \beta\alpha(1-\alpha)^{k-2} & if\ k \in \{2, 3, ..., \infty\}. \end{cases}$$

The average time to successfully transmit a message on the link given that it is on the top of the queue, $E[X_i]$, is:

$$E[X_i] = \sum_{k=1}^{\infty} k \cdot f_{X_i}(k) = 1 - \beta + \sum_{t=2}^{\infty} \beta\alpha t (1-\alpha)^{t-2} = \frac{\alpha + \beta}{\alpha}.$$

We illustrate for this *constantly transmit* policy how to compute the average number of message waiting in the queue. Assume a geometric arrival process and denote by λ the average arrival rate per unit time. Let $\bar{\lambda} = 1 - \lambda$. Given a queue of fixed size N, we create a finite state machine (FSM) for the number of messages in the queue. The designed FSM results in states $\{0, 1, 1', 2, 2', ..., N, N'\}$, where S and S' are states with the same number of messages in the queue but which may shift to different states and with different probabilities. This results from time varying behavior of the link, which leads to service rates that change with time.

We determine the transitions between these states and represent in the following transition matrix:

$$I = \begin{pmatrix} \bar{\lambda} & \lambda & 0 & 0 & 0 & 0 & 0 & ... \\ \bar{\lambda}(1-\beta) & \lambda(1-\beta) & \bar{\lambda}\beta & 0 & \lambda\beta & 0 & 0 & ... \\ \bar{\lambda}\alpha & \lambda\alpha & \bar{\lambda}(1-\alpha) & 0 & \lambda(1-\alpha) & 0 & 0 & ... \\ 0 & \bar{\lambda}(1-\beta) & 0 & \lambda(1-\beta) & \bar{\lambda}\beta & 0 & \lambda\beta & ... \\ 0 & \bar{\lambda}\alpha & 0 & \lambda\alpha & \bar{\lambda}(1-\alpha) & 0 & \lambda(1-\alpha) & ... \\ 0 & 0 & 0 & \bar{\lambda}(1-\beta) & 0 & \lambda(1-\beta) & \bar{\lambda}\beta & ... \\ 0 & 0 & 0 & \bar{\lambda}\alpha & 0 & \lambda\alpha & \bar{\lambda}(1-\alpha) & ... \\ ... & ... & ... & ... & ... & ... & ... & ... \end{pmatrix} \tag{4.12}$$

The rows of the matrix correspond to states at time t while the columns correspond to states at time $t+1$. Extracting the information from the transition matrix above results in the following system of equations:

$$\begin{cases} x_0 = \bar{\lambda}x_0 + \bar{\lambda}(1-\beta)x_1 + \bar{\lambda}\alpha x_{1'}; \\ x_1 = \bar{\lambda}x_0 + \lambda(1-\beta)x_1 + \lambda\alpha x_{1'} + \bar{\lambda}(1-\beta)x_2 + \bar{\lambda}\alpha x_{2'}; \\ x_{1'} = \bar{\lambda}\beta x_1 + \bar{\lambda}(1-\alpha)x_{1'}; \\ x_i = \lambda(1-\beta)x_i + \lambda\alpha x_{i'} + \bar{\lambda}(1-\beta)x_{i+1} + \bar{\lambda}\alpha x_{(i+1)'} & 2 \le i \le N-1; \\ x_{i'} = \lambda\beta x_{i-1} + \lambda(1-\alpha)x_{(i-1)'} + \bar{\lambda}\beta x_i + \bar{\lambda}(1-\alpha)x_{i'} & 2 \le i \le N; \\ x_N = \lambda(1-\beta)x_N + \lambda\alpha x_{N'}. \end{cases}$$

Solving this system of equations allows to determine an important value/metric, the probability of having i messages in the queue: $P_i = x_i + x_{i'}$. The average number of waiting messages in the queue can be calculated by:

$$\sum_{i=0}^{N} iP_i.$$

4.7.2 Back-off on bad

This transmission scheme keeps transmitting on the link as long as the observed state is good. It however ceases transmission for some time T (Theorem 3) after observing the link in the bad state after which transmission is resumed. Such form of transmission can be optimal only in positively correlated links. The probability distribution of X_i is:

$$f_{X_i}(k) = P[X_i = k] = \begin{cases} 1-\beta & if\ k = 1, \\ \beta\tau^{\mathsf{T}}(\alpha)(1-\tau^{\mathsf{T}}(\alpha))^{(\frac{k-1}{\mathsf{T}+1}-1)} & if\ k \in \{\mathsf{T}+2, 2(\mathsf{T}+1)+1, ..., \infty\}. \end{cases}$$

The average service time $E[X_i]$ is:

$$\begin{aligned}E[X_i] &= \sum_{k=1}^{\infty} k \cdot f_{X_i}(k) = (1-\beta) + \sum_{t=0}^{\infty}[(t+1)\mathsf{T}+1]\beta\tau^{\mathsf{T}}(\alpha)t(1-\tau^{\mathsf{T}}(\alpha))^t \\ &= \frac{\beta(\mathsf{T}+1)+\tau^{\mathsf{T}}(\alpha)}{\tau^{\mathsf{T}}(\alpha)}.\end{aligned}$$

4.7.3 Skip if good

This policy occurs in negatively correlated links and results in the following distribution:

$$f_{X_i}(k) = P[X_i = k] = \begin{cases} \tau(1-\beta) & if\ k = 2, \\ (1-\tau(1-\beta))\alpha(1-\alpha)^{k-3} & if\ k \in \{3, ..., \infty\}. \end{cases}$$

The average service time $E[X_i]$ is:

$$\begin{aligned}E[X_i] &= \sum_{k=1}^{\infty} k \cdot f_{X_i}(k) = 2\tau(1-\beta) + \sum_{t=0}^{\infty}(t+3)(1-\tau(1-\beta))\alpha(1-\alpha)^t \\ &= \frac{4(1-\tau(1-\beta))-\alpha(1-2\tau(1-\beta))}{\alpha}.\end{aligned}$$

4.8 Overview of Related Past Work

In this section, we discuss some existing work on related problems and settings. The purpose of this section is to understand what has been done in related areas and highlight the contribution of the results obtained in this chapter. We namely discuss (i) reliable links and dynamic link failures in distributed algorithms, (ii) communication over time-varying links under POMDP settings and (iii) restless bandits.

Previous work on distributed algorithms addressed the issue of achieving reliable communication over lossy links at different levels [96–98]. For example Aguilera et al. implemented a failure detector allowing a quiescent reliable communication when processes can fail [96]. Guerraoui et al. defined the stubborn link abstraction which is weaker than a reliable link but strong enough to solve important distributed problems such as consensus [97]. Another work by Basu et al. studied the solvability of problems in a system with process crashes and message losses [98]. Aside from reliable communication, certain distributed algorithms approaches studied systems with dynamic communication failures. Multiple efforts [99, 100] addressed the k-*consensus* problem, which requires only k processes to eventually decide. Moniz et al. [99] considered a system where message transmissions can be faulty: after some unknown time at most $\lceil \frac{n}{2} \rceil (n-k)+k-2$ faulty transmissions occur at each round. The number of faults per round prior to this is unrestricted. In a different setting, Moniz et al. [100] considered a communication system where a process sending a message will send it to all other processes sharing that link. Their algorithm tolerates up to f Byzantine processes and requires the number of omission faults affecting correct processes to be at most $\lceil \frac{n-f}{2} \rceil (n-k-f)+k-2$ in infinitely many rounds, a fairness assumption to guarantee liveness.

The work that we have done in this chapter goes one step further than that of [96–100] as it tries to achieve energy-optimal algorithms. Also, in comparison with previous work accounting for links failures through omission faults, we do not specify any bounds on the number of message losses. We hence allow through a probabilistic loss behavior, to send messages intelligently, precisely avoiding potential losses.

In a different context, existing work applied tools from Markov decision processes to solve communication problems over time-varying links, e.g., [52, 66, 101]. In their work, Laourine and Tong considered actions with a variable number of bits being sent in each action [52]. The problem addressed in this chapter is different. We consider transmission with a fixed number of bits in all actions and optimize for throughput under energy costs, which is not accounted for in Laourine and Tong's work [52]. Another work by Johnston and Krishnamurthy, applied results from optimal search theory of a Markovian target to find optimal transmission strategies [101]. They studied the problem of transmitting a single file/message over the link maximizing the average reward. In contrast, we consider an infinite sequence of messages to be transmitted and optimize for the discounted reward, which allows to favor lower latency. Most related to our work in this chapter is the work by Zhang and Wassermanin [66], which targeted achieving a suitable balance between throughput and energy consumption. The authors proved that the optimal transmission policy follows a threshold structure. Their paper assumed multiple power levels for transmission where a sender must decide on one, in case it decides to send. This energy level itself affects the probability of the message content being delivered correctly. However, in our work we assume that a link can lose messages but it does not manipulate their content. Besides, we define the optimal policy in terms of the costs and we obtain closed form solutions in terms of the system parameters. It is also important to note that all the above mentioned work have mainly considered positively correlated links (see Section 4.4) and error-free feedback. To the best of our knowledge, this chapter is thus the first to analyze all cases of link memory, i.e., memoryless, positively and negatively correlated cases and to study optimal transmission under lossy feedback.

Another closely related area to the problem studied in this chapter is that of restless multi-armed bandits that was applied mainly to cognitive radio networks [53, 54, 102]. In restless multi-armed bandits, there is a set of N independent projects that evolve over time and can yield some reward once they are activated based on their state at that time. A player is required to activate one of these projects at each time in such a way that maximizes the total long-term expected reward. The problem was studied in a setting where projects evolve according to the GE model where the parameters are known [53, 54, 102]. Zhao et al. prove that when projects are independent and identically distributed, a myopic policy, which activates the project with the highest belief, is optimal under the positive correlation assumption [54]. Another work by Guha et al. also studied this problem under the positive correlation assumption and independent but not necessarily identically distributed links [53]. They propose an approximation algorithm with a performance guarantee of 2. Among the work on restless bandits, that by Liu and Zhao is perhaps the closest to the work in this chapter [54]. The authors compute a closed form of an index known as the *Whittle index* (see [103] for a complete definition of this index), which measures the attractiveness of activating a project. They propose to activate at every time the project with the highest index. Our work in this chapter uses a notion similar to the *subsidy* of the Whittle index to derive its closed forms. However, the link considered here has non-trivial differences from the link considered in all cited work on restless bandits [53, 54, 102], as we permit for feedback to be lost. We also allow for positive rewards and negative costs to coexist and allow for costs to be incurred at two levels: when the link is idled and when a transmission fails.

4.9 Chapter Summary

We presented in this chapter an analytic study describing how energy-optimal reliable communication can be built, with high probability synchrony, over unreliable links. The analysis was conducted for a time-varying lossy link that could for example capture the unreliable behavior of wireless links. The importance of such a communication abstraction lies in the fact that communication is the basic building block of any distributed algorithm. Consequently algorithms built on top of this abstraction can inherently deliver energy efficient services and potentially to the whole network, e.g., through reliable broadcast primitives. In contexts such as sensor networks, energy efficiency is crucial to prolong the lifetime of the network which typically comprises battery powered devices that are tedious to recharge.

We discussed four forms of $Ack/Nack$ feedback mechanisms.

We first studied a reliable feedback mechanism, where a sending process is notified about the success/failure of the previous transmission. Our results in this chapter showed the following:

- *Optimal Transmission.* Despite the fact that POMDPs are P-SPACE hard, this chapter derived explicit solutions proving that the optimal transmission scheme, conforms to a computationally inexpensive and easily implementable structure.
- *Reliability.* We showed that the optimal transmission policy, under certain energy-throughput trade-offs and link parameters[4], can stop transmission for good. Suspending transmission as such prevents reliable communication. We hence identified the necessary conditions of system parameters to achieve reliability. We proved that when reliability is guaranteed, the optimal transmission policy takes one of three forms: *constantly transmit, back-off on bad* and *skip if good*.
- *Synchrony.* Despite the dynamic message losses, we established methods to obtain high probability guarantees on the total time required to successfully send a message over the link.

[4]An example could be when the energy cost relative to transmitting is high and the link is rarely in the good state.

We then studied the impact of lossy feedback (feedback which can be lost) on the optimal transmission by considering different feedback mechanisms. Assuming a mechanism where feedback about the link state is periodically sent, we showed that a myopic greedy policy [102] is optimal and reliable. If feedback is sent only regarding the times when the sender transmits over the link (regardless if the transmitted message is successfully received or not), we showed that the optimal transmission will be similar to that in which feedback is reliable. In other words, closed form expressions allowing an easy implementation of the policy can be obtained. However, if we assume that the feedback is only sent when transmission is successful, we showcased that the optimal transmission policy does not necessarily comply with an easily implementable structure.

In a nutshell this chapter's main contributions can be stated as follows:

- A first full analytic study of optimal transmission policies of time-varying links for several reliable and lossy feedback.
- A derivation of explicit and closed form solutions to implement optimal transmission policies.
- An establishment of the necessary conditions for reliable communication using optimal transmission policies.
- Methods to obtain synchronous message transfers with high probability.

5 Probabilistic and Temporal Failure Detectors

This chapter goes one step beyond communication to investigate failure detectors, which are software abstractions that build directly on top of the communication services of a distributed system. In fact, the guarantees provided by communication within a distributed system determine what types of failure detectors can be implemented in that system. In this sense, failure detectors themselves can be viewed as a higher level abstraction of communication guarantees.

Nevertheless, a celebrated aspect of the failure detector approach for solving distributed computing problems is modularity: failure detectors allow the construction of algorithms using abstract failure detection mechanisms, defined by axiomatic properties, as building blocks. The minimal synchrony assumptions on communication, which enable to implement the failure detection mechanism, are studied separately.

The typical synchrony assumptions for implementing classic failure detectors [7] are generally expressed as eventual guarantees that need to hold, after some point in time, *forever* and *deterministically*. But in practice, link properties never do; synchrony assumptions may hold only probabilistically and temporarily [30–32, 37–40].

In this chapter, we study failure detectors in a realistic distributed system where asynchrony is inflicted by probabilistic synchronous communication. Given such a system, i.e., with probabilistic communication, we answer questions related to: (i) the possibility of implementing existing failure detectors for solving important distributed problems, e.g. the $\diamond\mathcal{S}$ failure detector for solving consensus [7] and (ii) the feasibility of using and applying existing asynchronous algorithms that use classic failure detectors, in systems with probabilistic communication.

5.1 Motivation

The failure detector abstraction is an elegant means to solve difficult distributed computing problems, such as the fundamental consensus problem[1], in a modular manner [8]. Roughly speaking, a failure detector is a distributed "oracle" augmented to an asynchronous system. The purpose of this oracle is to provide hints (possibly incorrect) about which processes of the system have crashed [7]. A failure detector is formally defined by high-level axiomatic properties which, in turn, encapsulate synchrony assumptions that allow problems like consensus to be solved. The task of implementing a given failure detector using synchrony assumptions becomes a separate, lower-level task.

A large body of work [104–109] has been devoted to determine which synchrony assumptions are sufficient to implement for instance the $\diamond\mathcal{S}$ failure detector, established to be the weakest, in a precise sense [110], to solve consensus-like problems. The underlying synchrony assumptions, typically adopted in the distributed community to implement $\diamond\mathcal{S}$ take for example the form of some links being eventually timely; i.e., after some point in time, these links never "delay" messages. Such assumptions, besides placing the failure detector approach under scrutiny in the distributed computing community itself [111], are questionable from the networking perspective. We elaborate further on this.

Questioning Failure Detectors. As discussed by Dwork et al. in [62] as well as Lamport in [5], in practice, consensus requires the system to be "good" only sufficiently long, while the weakest failure detector to implement consensus requires parts of the system after some point to be "good" forever [104–109]. This controversy was raised and discussed by Charron-Bost et al. [111], who even suggested to abandon failure detectors and seek a better computing model.

From a networking perspective, the dependence of failure detectors on a "deterministic forever" synchrony condition, suggests that failure detectors may be practically unfit. This results from the fact that typical synchrony guarantees provided by networks are, at best, probabilistic. In fact, a considerable amount of research on packet transmissions confirms that synchrony guarantees in various kinds of networks, e.g., wired power line networks [38], hybrid wired/wireless networks [112] and wireless networks [65, 89, 113], are indeed probabilistic[2] (as discussed and shown in Chapter 3). Consequently, failure detectors, like $\diamond\mathcal{S}$, might not be possible to implement in such networks and thus algorithms designed on top of such failure detectors cannot be practically used. We argue that this in fact is not the case!

Objective. This chapter aims at bridging the gap between the modular distributed computing approach, based on failure detectors to circumvent the impossibility of fundamental distributed problems, and the networking view of communication link characteristics. In this view (especially in the context of real-world physical systems like sensor networks and smart grids), link characteristics are probabilistic and temporary, rather than deterministic and perpetual.

[1]Consensus [5] is an essential building block of most distributed computing problems and applications such as leader election, state machine replication, atomic commit, etc.

[2] In practice, messages can be delayed at any point in time, for example as a result of bad transmission quality of the underlying channel [65, 89] or unpredictable loads on the system.

To this end, we consider in this chapter $\mathcal{N}$, a fully connected distributed system with probabilistic synchronous communication, i.e., links can probabilistically lose messages, but otherwise respect a known bound on the maximum transmission time. Such losses inherently inflict *asynchrony*, as it can take arbitrarily long until a reliable/successful message transmission between any two processes happens. As opposed to the partial synchrony typically considered in distributed computing, $\mathcal{N}$ does not require the delay of a reliable/successful message, after some point in time, to satisfy a specific bound forever. Solutions on the basis of our system $\mathcal{N}$ are thus applicable to a wide range of real networked systems.

Given system $\mathcal{N}$, we show in this chapter that the classic eventual failure detectors introduced by Chandra and Toueg [7], such as $\diamond\mathcal{S}$, cannot be implemented in $\mathcal{N}$. We propose new probabilistic failure detector abstractions as a solution that preserves modularity of failure detectors, precisely, by allowing existing asynchronous algorithms using eventual failure detectors to still be used to solve certain distributed problems, such as the fundamental consensus problem, in systems like $\mathcal{N}$.

First we start by describing system $\mathcal{N}$ more formally, providing elaborate details about the underlying assumptions.

5.2 Overview of System $\mathcal{N}$

We consider a distributed system $\mathcal{N}$ consisting of a finite set Π of $n > 1$ processes, $\Pi = \{p_1, p_2, ..., p_n\}$, which communicate by message passing. We assume, without loss of generality, that processes have access to a global clock with discrete time events denoted by $t : \{1, 2, 3, ...\}$ (the global clock is used for presentation simplicity, in Appendix B.2 we show that a global clock can be substituted by local clocks that do not have to be synchronized).

Processes can send and/or receive a message, at these discrete time events. The time interval between consecutive events in t is assumed to be an upper bound on the propagation delay (t_{pg}) over any link connecting any two processes. Processing delays are assumed to be negligible compared to communication delays.

Communication Links. The links interconnecting processes are assumed to be uni-directional uni-cast links. In particular, every pair of processes (p_i, p_j) is connected by two uni-directional links: l_{ij} and l_{ji}. These links exhibit changes in their transmission quality, as the quality of the underlying channels might depend on various propagation conditions. We thus assume that a link l_{ij} has a probability $0 < P_{ij}(t) < 1$ of losing the message sent at time t (if any is sent). This captures the very idea that a link is not always reliable and can lose messages for an unbounded but finite period[3]. The value of $P_{ij}(t)$ can change with time; at each time $t : \{1, 2, 3, ...\}$, $P_{ij}(t)$ may have any value in $(0, 1)$. We refer to such links as *probabilistic*. A probabilistic link is an instance of the *fair-loss* link [97], where a message sent infinitely often is received infinitely often.

[3]Note that $P_{ij}(t)$ is assumed to be strictly less than 1 and greater than 0. However, our theoretical results can be extended to the case where $0 \leq P_{ij}(t) \leq 1$, such that $0 < P_{ij}(t) < 1$ occurs infinitely often and that $P_{ij}(t) = 0$ or $P_{ij}(t) = 1$ occur for some bounded duration.

Faulty Processes. Processes can fail by crashing, i.e., by halting prematurely. We consider that process crashes are permanent, i.e., no crashed process can recover. We use $\mathcal{C}$ to denote the number of *correct* processes, i.e., processes that do not fail. We assume that $\mathcal{C} \geq 1$. When a process p_i sends a message m to process p_j (for $i \neq j$) and m is successfully propagated by link l_{ij}, p_j receives/delivers m. However if m is lost by link l_{ij}, p_j receives nothing.

Reliable Message Transmission. Despite probabilistic losses, a reliable message transmission, be it a unicast or a broadcast can still be achieved in $\mathcal{N}$ [96, 114, 115]. Reliable transmissions can be provided via abstractions running on top of the probabilistic links of system $\mathcal{N}$. For example, a reliable link abstraction would guarantee the following with probability 1: a message sent by a correct process p_i to another correct process p_j, will be delivered by p_j at some future point in time[4]. We provide next the complete specifications of a reliable link and a reliable broadcast primitive in $\mathcal{N}$. A reliable link abstraction guarantees:

1. *Reliable delivery:* If a correct process p sends a message m to a correct process q at time t, then with probability 1 the following is guaranteed: there exists some time $t' > t$ where q delivers m. In particular, q can deliver m at $t + T$ with positive probability $\forall\, T \in \{1, 2, 3, ...\}$.
2. *No duplication:* No message is delivered by a process more than once.
3. *No creation:* If some process q delivers a message m with sender p, then m was previously sent to q by p.

A reliable broadcast primitive can also be defined in $\mathcal{N}$:

1. *Validity:* If a correct process p broadcasts a message m at time t, then with probability 1 the following is guaranteed: at some time $t' > t$ all correct processes will have delivered m. Precisely, a correct processes can deliver m at $t + T$ with positive probability $\forall\, T \in \{2, 3, ...\}$.
2. *No duplication:* As for a reliable link.
3. *No creation:* If some process q delivers a message m with sender p, then m was previously broadcast by p.
4. *Agreement:* If message m is delivered by some process at time t, then with probability 1 the following is guaranteed: at some time $t' \geq t$, m is delivered by every correct process.

[4]**CRUCIAL**: The phrase "the following is guaranteed with probability 1: there is a time when event X occurs" does not mean that there is a point in time where event X occurs with probability 1 but rather that over the infinite course of time event X will certainly occur. For example, if a fair coin is flipped an infinite number of times, then with probability 1, there is a time when the coin lands on head, however there is no point in time where flipping the coin lands on head with probability 1.

The reliable link abstraction can be achieved over a probabilistic link l_{ij}, for example by deploying buffers and message re-transmissions [96, 114–116] (as shown in Chapter 4). Typically, process p_i keeps re-transmitting a message m forever or in practice until some acknowledgment is obtained for m. In this sense, the message transmission delay of a message m is measured by the number of time slots elapsed from p_i's first attempted transmission of m until the time when m is successfully received by p_j. We do not elaborate on implementation details of such an abstraction, as Chapter 4, in addition to existing work, already addresses this problem in systems where links lose messages [96, 115]. Clearly, such a reliable link abstraction does not provide any deterministic bounds on message transmission delays, as message losses may span unbounded duration. However the reliable link abstraction in $\mathcal{N}$ offers instead a probability distribution on the delay of a message.

The reliable broadcast primitive can be built for example using the reliable link abstraction in system $\mathcal{N}$. Algorithms such as those detailed in [117] can be directly applied and will result also in a reliable broadcast where the delay to deliver a broadcast message, despite possibly being arbitrarily long, admits a probability distribution.

5.3 Probabilistic Temporal Failure Detection

In a system augmented with a failure detector, each process has access to a local failure detector module [7]. This module monitors other processes in the system and typically maintains a set of those that it currently suspects to have crashed. Chandra and Toueg [7] defined various kinds of failure detectors based on their achievable properties, namely *completeness* and *accuracy*.

Roughly speaking, the completeness property describes the failure detector's ability to suspect crashed processes, while the accuracy property defines the failure detector's ability of not suspecting correct processes. For instance, the $\diamond\mathcal{S}$ failure detector, established to be the weakest, in some sense [110], to solve consensus in an asynchronous system, guarantees the following two properties:

1. *Strong completeness:* eventually every process that crashes is permanently suspected by every correct process.
2. *Eventual weak accuracy:* there is some time instant $t_G \in \{1, 2, 3, ...\}$ after which some correct process is never suspected by any correct process.

5.3.1 The Impossibility of Eventual Weak Accuracy

We now establish several lemmata proving certain guarantees on message delivery and process failures.

Lemma 10. *In system $\mathcal{N}$, for any finite period Δt and at any time instant $t_s \in \{1, 2, 3, ...\}$, all messages that are sent on link l_{ij} during the interval $t_s + \Delta t$, can be lost with positive probability.*

Proof. Recall that $P_{ij}(t)$ is the probability with which the link l_{ij} loses a message at time t. Let $P_{ij}(t \cap t')$ be the probability that l_{ij} loses the messages (if any is sent) at time t and time t'. Since $0 < P_{ij}(t) < 1\ \forall t$, then

$$0 < P_{ij}(t) = \frac{P_{ij}(t \cap t')}{P_{ij}(t'|t)} < 1\ \forall\, t',\ t. \tag{5.1}$$

By (7.1), $P_{ij}(t'|t) > 0$ (and $0 < P_{ij}(t \cap t') < 1$). By induction, we have $P_{ij}(t'|t, t+1, ..., t'-1) > 0\ \forall\ t' >\ t$. Denote by $\mathsf{B}(t)$ the event that l_{ij} losses all messages (if any is sent) for the interval $t + \Delta t$, for any finite period Δt. Then the probability of $\mathsf{B(t)}$ happening is:

$$\begin{aligned} Pr(\mathsf{B(t)}) &> P_{ij}(t \cap t+1 \cap t+2 \cap ... \cap t+\Delta t) \\ &= P_{ij}(t) \times P_{ij}(t+1|t) \times ... \times P_{ij}(t+\Delta t|t, t+1, ..., t+\Delta t - 1) > 0. \end{aligned}$$

Given $P_{ij}(t) < 1$, then we have $0 < Pr(\mathsf{B(t)}) < 1$. □

Lemma 11. *Consider any finite period Δt, any time instant $t_s \in \{1, 2, 3, ...\}$ and any subset of processes ΔP. In system $\mathcal{N}$, all processes $\notin \Delta P$ can lose, with positive probability, all messages sent in the interval $t_s + \Delta t$ from and to processes in ΔP.*

Proof. The probability to have any subset of processes losing all messages exchanged with all remaining processes for any finite period Δt, depends on the individual probabilities of the relative individual links losing all sent messages during the interval Δt.

Following from Lemma 10, any link in the system can drop all messages (if any were sent) for a finite but unbounded time, with a positive probability. Denote by $\mathsf{B_{ij}}(t)$ the event that l_{ij} losses all messages (if any were sent) in the interval $t + \Delta t$, for any finite period Δt. Then by Lemma 10, $0 < Pr(B_{ij}(t)) < 1$.

Following the arguments as in proof of Lemma 10, we have:

$$0 < Pr(\bigcap_{i,j \in [1,n]} B_{ij}(t)) < 1\ \forall\, i,\ j \in [1, n],$$

where n is the total number of processes in the system. This concludes the proof. □

Lemma 12. *In system $\mathcal{N}$, no correct process can determine with probability 1, at any point in time, that some other process in the system has crashed, i.e., for any finite period a correct process can be (with probability> 0) indistinguishable from a crashed one.*

Proof. Following from Lemma 11, a single process p can lose all messages exchanged with the whole network (i.e., from and to p) with positive probability for any finite time.

Consider an execution $e1$ where p crashes at time t and another execution $e2$ where p loses all communication at time t for any finite period. Then executions $e1$ and $e2$ are indistinguishable to all processes (except p) for the whole time in which communication is lost, i.e., any finite time by Lemma 11. $\square$

With these lemmata, we can prove that $\diamond\mathcal{S}$ cannot be implemented in system $\mathcal{N}$.

Theorem 9. *It is impossible to implement "$\diamond\mathcal{S}$ with probability 1" in $\mathcal{N}$ even if at most one process can crash. That is, in the presence of process crashes, it is impossible to have an algorithm in $\mathcal{N}$ that guarantees strong completeness deterministically and which ensures, with probability 1, the following: there is a time after which some correct process is never suspected by all correct processes.*

Proof. We proceed by contradiction. Without loss of generality, assume a system $\mathcal{N}$ of $n = 2$ processes, p_1 and p_2. Suppose that there exists an algorithm $\mathcal{A}$, that guarantees both strong completeness and eventual weak accuracy.

Consider three executions: (i) $e1$: an execution where p_1 fails at some time instant t_s in $\{1, 2, 3, ...\}$, (ii) $e2$: an execution where p_2 fails at t_s and (iii) $e3$: an execution where p_1 and p_2 are both correct but all messages exchanged between p_1 and p_2 are lost during the interval $t_s + \Delta t$. By the strong completeness property of $\mathcal{A}$, in executions $e1$ and $e2$ there is a finite period, say $\Delta t'$, after which p_2 suspects p_1 and p_1 suspects p_2 respectively. By Lemma 11, execution $e3$ is a valid execution in $\mathcal{A}$ and Δt can be arbitrarily long, specifically $\Delta t \geq \Delta t'$. Thus, by the strong completeness of $\mathcal{A}$, p_1 suspects p_2 and p_2 suspects p_1 in $e3$. By Lemma 11, execution e_3, such that $\Delta t \geq \Delta t'$, can occur with positive probability at any time instant in $\{1, 2, 3, ...\}$. This implies that $\mathcal{A}$ cannot guarantee with probability 1 the following: there exists some time after which some correct process is *never* suspected (i.e., remains trusted forever) by any correct process. This violates the eventual weak accuracy of $\mathcal{A}$. $\square$

As a consequence of Theorem 9, we study how to vary the properties of $\diamond\mathcal{S}$, defining a variant $\diamond\mathcal{S}^*$, which is implementable in our system $\mathcal{N}$.

5.3.2 Probabilistic $\&$ Temporal Failure Detectors

We define a new probabilistic weak accuracy property that holds temporarily for periods that can be arbitrary long. Combined with strong bounded completeness, these two properties define a new failure detector $\diamond\mathcal{S}^*$.

Definition 1. *Failure detector $\diamond\mathcal{S}^*$ guarantees both of the following properties:*

1. ***Strong bounded completeness**: every process that crashes is permanently suspected by every correct process after a maximum of T_D time slots of the actual crash.*
2. ***Probabilistic $\&$ temporal weak accuracy:** Consider any finite duration Δt. With probability 1 the following occurs: there exists infinitely many time instants t_G, such that a unique correct process is not suspected by any correct process for the interval $t_G + \Delta t$.*

Theorem 10. *It is possible to implement $\diamond\mathcal{S}^*$, in system $\mathcal{N}$, assuming $n-1$ processes can crash.*

Proof. Let t_s be any time instant in $\{1, 2, 3, ...\}$ and Δt be any finite duration.

Lemma 13. *There is a positive probability that all messages sent by a correct process to all other correct processes, in the interval $t_s + \Delta t$, are not lost (i.e., successfully received).*

Proof. Let $\mathsf{E_{ij}}(t_s)$ be the following predicate: All messages sent by a correct process p_i to a correct process p_j, in the interval $t_s + \Delta t$, are not lost, i.e., successfully received by p_j. The probability that predicate $\mathsf{E_{ij}(t_s)}$ occurs is: $0 < Pr(\mathsf{E_{ij}(t_s)}) < 1$. This can be easily deduced from the proof of Lemma 10, given that the probability of a message (sent from p_i to p_j) being not lost at any time instant $t_s \in \{1, 2, 3, ...\}$ is $0 < 1 - P_{ij}(t_s) < 1$. Let $\mathsf{E}(t_s)$ be the following predicate: All messages sent by a correct process p_i to every correct process $p_j \in \mathcal{C}$, in the interval $t_s + \Delta t$ are not lost. Since $0 < Pr(\mathsf{E_{ij}}(t_s)) < 1$, the probability of predicate $\mathsf{E}(t_s)$ happening, as in the proof of Lemma 11, is $Pr(\mathsf{E}(t_s)) = Pr(\bigcap_{j:\{p_j \in \mathcal{C}\}} E_{ij}(t_s)) > 0$. □

Assume algorithm $\mathcal{A}$ executing the following: (i) all processes periodically, at every time event $t = \{1, 2, ..., \infty\}$, broadcast messages (i.e., they send messages to all other processes in the system) and (ii) initially all processes trust (do not suspect) each other. At every time instant in $\{2, 3, 4, ...\}$, process p_i suspects another process p_j only if p_i receives no new message from p_j, otherwise p_i trusts p_j.

The strong bounded completeness of $\diamond\mathcal{S}^*$ is ensured by $\mathcal{A}$. A process that crashes at time instant $t_{crash} \in \{1, 2, 3, ...\}$ stops sending messages and thus by (ii) will be suspected at all times $> t_{crash} + 1$ by all correct processes forever (that is with $T_D = 1$). Let's denote by $\mathsf{E_p(t_s)}$ the following predicate: during the interval $t_s + \Delta t$, all messages sent by a correct process p are successfully received by all correct processes. By (ii) of algorithm $\mathcal{A}$, $\mathsf{E_p(t_s)}$ implies that process p is not suspected by any correct process during the interval $t_s + \Delta t$. Following from Lemma 13, the probability of observing $\mathsf{E_p(t_s)}$ is greater than zero. Note that in $\mathcal{A}$ any correct process can be selected as the unique correct process.

Since in $\mathcal{A}$ processes keep sending messages to all other processes forever (infinitely) and since for any time instant $t_s \in \{1, 2, 3, ...\}$ $P(\mathsf{E_p(t_s)}) > 0$, then with probability 1 the following is satisfied: there exists infinitely many time instants t_G when predicate $\mathsf{E_p(t_G)}$ happens. $\mathcal{A}$ thus guarantees the accuracy of $\diamond\mathcal{S}^*$, concluding the proof. □

In Appendix B.1, we discuss the implementability of other types of probabilistic failure detectors; namely $\mathcal{P}^*$, a probabilistic variant of the perfect failure detector $\mathcal{P}$, and $\diamond\mathcal{P}^*$, a probabilistic variant of $\diamond\mathcal{P}$. In this main part of the chapter, however, we solely focus on $\diamond\mathcal{S}$ and $\diamond\mathcal{S}^*$ for better illustration.

5.4 $\diamond\mathcal{S}^*$ Bounds and Algorithms

We study in this section the communication overhead of $\diamond\mathcal{S}^*$ and present a $\diamond\mathcal{S}^*$ optimal algorithm (communication-wise).

5.4.1 Lower Bounds

First, we identify the bounds on the number of processes and links that need to respectively send and carry messages forever in any algorithm implementing $\diamond\mathcal{S}^*$.

Theorem 11. *Consider any algorithm $\mathcal{A}$ that implements $\diamond\mathcal{S}^*$ in system $\mathcal{N}$ of $n \geq 2$ processes, where $n-1$ processes can crash. Then, $\mathcal{C}-1$ distinct processes send messages infinitely often in $\mathcal{A}$ with probability > 0.*

Proof. Assume that t_s is any time instant in $\{1, 2, 3, ...\}$ and Δt is any finite duration. If no correct process sends messages infinitely often, i.e., all correct processes stop sending messages at some point in time, say t, then $\diamond\mathcal{S}^*$ cannot be implemented. This holds, since after time t every correct process becomes indistinguishable from a crashed process (w.r.t. to all other processes in $\mathcal{N}$). By the strong bounded completeness of $\diamond\mathcal{S}^*$, every correct process suspects all processes in $\mathcal{N}$ after some bounded duration. This violates the probabilistic eventual weak accuracy property of $\diamond\mathcal{S}^*$.

Thus to implement $\diamond\mathcal{S}^*$ in $\mathcal{N}$ some correct process(es) should send messages infinitely often. We now prove Theorem 11 by showing that in system $\mathcal{N}$ with $n \geq 2$ processes, where $n-1$ processes can crash, it is impossible to have with probability 1 an implementation of $\diamond\mathcal{S}^*$ where eventually, only $c : \{0 < c < \mathcal{C}-1\}$ correct processes send messages infinitely often.

Consider $\bar{c}$ to be the subset of correct process that stop sending messages after time instant t_s and consider the following two executions: (i) $e1$: all processes in c crash at time instant $t_{crash} > t_s$ and (ii) $e2$: all messages exchanged between processes in c and processes in $\bar{c}$ in the interval $t_{crash} + \Delta t$ are lost. By Lemma 12, execution $e2$ is valid, as it has a positive probability of happening. For processes in $\bar{c}$ executions $e1$ and $e2$ cannot be distinguishable in any finite amount of time (since Δt is any finite duration). Therefore, after some time (T_D) processes in $\bar{c}$ suspect all processes in c. If no process in $\bar{c}$ starts to send a message afterwards then if all processes in c did crash no correct process in the system will send messages (a violation). Thus some process(es) in $\bar{c}$ should send messages, which in the case of $e2$, i.e., if processes in c are still alive, results in more than c process sending messages. Since execution $e2$ occurs with a positive probability, then it is impossible to guarantee with probability 1 that only $c : \{0 < c < \mathcal{C}-1\}$ correct processes send messages infinitely often. This concludes the proof. □

N.B. Theorem 11 does not mean that each process sending messages infinitely often, needs to do so by broadcasting (i.e., by sending the message to all other processes in the system). A process may send messages to any subset of the processes in the system. We show now that Theorem 11 can be circumvented, in the sense that $\diamond\mathcal{S}^*$ algorithms can be implemented such that, with probability 1, less than $\mathcal{C} - 1$ processes send messages infinitely often. It can be done by limiting the maximum number of processes that can crash.

Theorem 12. *Given an algorithm $\mathcal{A}$ that implements $\diamond\mathcal{S}^*$ in $\mathcal{N}$ with $n \geq 2$ processes of which at most $f < \frac{n}{2} - 1$ processes may crash, then the number of processes sending messages infinitely often in $\mathcal{A}$ can be less than $\mathcal{C} - 1$.*

Proof. Consider an algorithm $\mathcal{A}$ which deterministically selects any $f + 1$ processes to keep sending messages infinitely often after some point in time to all processes in $\mathcal{N}$, while all other processes stop sending messages completely. Since the maximum number of processes that may fail is f, then $\mathcal{A}$ guarantees that at least one correct process will send messages infinitely often and at maximum $f + 1$ will send messages infinitely often. By the proof of Theorem 10, it is clear that $\diamond\mathcal{S}^*$ can be implemented in $\mathcal{N}$ even if only one correct process sends messages, to all other processes in $\mathcal{N}$, infinitely often. This proves that $\mathcal{A}$ implements $\diamond\mathcal{S}^*$ such that at most $f+1$ processes send messages infinitely often. $f + 1 < \frac{n}{2} < \mathcal{C} - 1$ (since $\mathcal{C} \geq n - f$). $\square$

We now determine the number of links that need to carry messages infinitely often in algorithms implementing $\diamond\mathcal{S}^*$. Despite the asynchrony caused by probabilistic message loss, system $\mathcal{N}$ can, with positive probability, reach a point in time where links can be timely (i.e., ensure that the delay of a reliable message transmission respects some bound) for any finite duration. We define next what it means for algorithms to be optimal in $\mathcal{N}$. Let L_{min} be the minimum number of links required to carry messages forever to implement failure detector $\mathcal{X}$ in a synchronous system[5]. Let $\mathcal{A}$ be an algorithm that implements failure detector $\mathcal{X}$, then:

Definition 2. *$\mathcal{A}$ is optimal, if L, the number of links carrying messages infinitely often in $\mathcal{A}$, satisfies:* $\lim_{\Delta t \to \infty} L = L_{min}$*, where Δt is an interval in which links are timely.*

Theorem 13. *The minimum number of links which need to send messages forever to implement $\diamond\mathcal{S}^*$ in a synchronous system where $n - 1$ processes may crash is $\mathcal{C}$ (possibly $\mathcal{C} - 1$ depending on what processes crash).*

Proof. First we prove that it is impossible to implement $\diamond\mathcal{S}^*$ if $\mathcal{C} - 2$ links send messages infinitely often. The proof is by contradiction. Assume an implementation $\mathcal{A}$ of $\diamond\mathcal{S}^*$ in which only $\mathcal{C} - 2$ links carry messages forever. Then there is in $\mathcal{A}$ at least one correct process p which eventually (i.e., at some point t in time) does not exchange messages with any other correct process.

Assume an execution $e1$ of $\mathcal{A}$ with $\mathcal{C} > 1$ correct processes (including p) and another execution $e2$ of $\mathcal{A}$ similar to $e1$ however where p crashes after time t (the time when p eventually stops

[5]In a synchronous system processing and message delays are bounded. This means that messages can be lost as long as they can be re-transmitted successfully ensuring that total transmission time satisfies the delay bound.

exchanging messages). $e1$ and $e2$ are indistinguishable to all processes (other than p) and thus processes in $e2$ will keep using the same number of links. However in $e2$ since the number of correct processes is less, then $\mathcal{C} - 3$ links should be used which contradicts that $e1$ and $e2$ are indistinguishable.

Now assume an implementation $\mathcal{A}'$ of $\diamond\mathcal{S}^*$ in which only $\mathcal{C} - 1$ links carry messages forever. Since there is $\mathcal{C}$ correct processes, such an implementation is only possible if correct processes are arranged in a tree topology (of which a star and a linear list are a special case). In such an arrangement the root of the tree sends heartbeat messages, indirectly, to the rest of the correct processes. Consider an execution $e1$ of $\mathcal{A}'$ in which $\mathcal{C}$ processes are correct and let t be the point in time after which only $\mathcal{C} - 1$ links carry messages forever. Consider now $e2$, an execution identical to $e1$ up to t, but where a leaf process p (assumed correct in $e1$) crashes at time t (a leaf process has no successor processes). $e1$ and $e2$ are indistinguishable, to all processes above p in the tree (in this case all processes since p is a leaf node). Hence the process sending messages to p will not stop sending messages to p in $e2$, although the number of correct processes in $e2$ is one less than in $e1$, resulting in $\mathcal{C}$ links being used forever. However, if the process p (which crashes in $e2$) is not a leaf node, then p can be suspected by processes lower in the tree (or following it in a linear list) and initiate a procedure to eliminate communication with p and restore the fact that $\mathcal{C} - 1$ links are used, concluding our proof. $\square$

We present next an optimal $\diamond\mathcal{S}^*$ implementation in $\mathcal{N}$.

5.4.2 An Optimal $\diamond\mathcal{S}^*$ Implementation

We now present an optimal algorithm (Algorithm 1) implementing $\diamond\mathcal{S}^*$.

We assume that processes are arranged in a logical linear list, where p_1 is at the head and p_n is at the tail. An intermediate process p_i is preceded by process p_{i-1} and followed by process p_{i+1}. When links in the system are timely for some finite interval, the number of links carrying messages infinitely often converges to $\mathcal{C}$ if at least one process crashes (possibly to $\mathcal{C} - 1$ if process p_n, at the tail of the logical linear list, does not crash) and to $\mathcal{C} - 1$ when no crashes occur. Recall that a timely link ensures that the delay of a reliable/successful message respects some bound; in this case we assume it to be the specified *time-out*.

The basic idea underlying Algorithm 1 is that a process at location x in the list always suspects all processes succeeding it, i.e., processes at locations $[x + 1, ..., n]$. The goal of Algorithm 1 is to achieve two things:

1. Every correct process permanently suspects all crashed processes preceding it after T_D time slots of the crash.
2. When links are timely, no correct process suspects the first correct process in the logical linear list.

Algorithm 1 An Optimal $\diamond\mathcal{S}^*$ Algorithm.

```
Initialize:
set $pred(p_i) = p_{i-1}$    //set to null if i=1
set $succ(p_i) = p_{i+1}$    //set to null if $i$=$n$
set $L(p_i) = \{p_{i+1}, \ldots, p_n\}$

Repeat periodically:
if $succ(p_i) \neq$ null then
    send $<$ heartbeat, L(p_i), p_i $>$ to $succ(p_i)$
end if

upon event Timeout on $pred(p_i)$ do    //$pred(p_i)$ not null
    $L(p_i) = L(p_i) \cup \{pred(p_i)\}$
    send $<$ suspicion, p_i $>$ to $pred(p_i)$
    set $pred(p_i) =$ process directly above $pred(p_i)$ in the list.

upon event receive $<$ suspicion, p_j $>$ do
    send $<$ heartbeat, L(p_i), p_i $>$ to $p_j$
    for $i < k < j$ do
        send $<$ Alive?, p_i $>$ to $p_k$
    end for
    set $succ(p_i) = p_j$

upon event receive $<$ Alive?, p_j $>$ do
    send $<$ heartbeat, L(p_i), p_i $>$ to $p_j$

upon event receive $<$ heartbeat, L(p_j), p_j $>$ do
    if $j < i \wedge index(pred(p_i)) \leq j$ then    // $index(p_i) = i$
        set $pred(p_i) = p_j$
        $update\_list(L(p_j))$
    end if
    if $j > i \wedge index(succ(p_i)) > j$ then
        set $succ(p_i) = p_j$
    end if

Function $update\_list(L(p_j))$:
$L(p_i) = L(p_j)$
remove $p_i$ from $L(p_i)$
```

Every process p_i maintains a set of suspected processes $L(p_i)$ and two variables $pred(p_i)$ and $succ(p_i)$ to respectively refer to the current predecessor process which is monitored by p_i and the current successor process to which p_i periodically (e.g., every t) sends heartbeat messages $<$ heartbeat, L(p_i), p_i $>$. Note that process p_i at the head of the list has $pred(p_i) =$ null whereas process p_j at the tail of the list has $succ(p_i) =$ null. We assume that processes have unique identifiers (names) and that they know their position in the list. Process p_i at all times suspects all processes down the list including the tail, i.e., $p_j \in L(p_i) \, \forall \, p_j : \{i < j \leq n\}$.

A process p_i suspects $pred(p_i)$ which it is monitoring, when a *time-out* expires (possibly some multiple of the sending period). In case of suspicion, p_i sends through a reliable link abstraction (as discussed in Section 5.2), a message $<$ suspicion, p_i $>$ to $pred(p_i)$ and sets its $pred(p_i)$ to the process before $pred(p_i)$ in the list (regardless if that process is in $L(p_i)$ or not) and updates its set of suspected processes $L(p_i)$ accordingly. Upon its receipt of a $<\mathsf{suspicion}, \mathsf{p_j}>$ message, a process p_i which is alive updates its successor to $succ(p_i) = p_j$ and will start sending $<\mathsf{heartbeat}, \mathsf{L(p_i)}, \mathsf{p_i}>$ to p_j. In addition, p_i also sends a message $<\mathsf{Alive?}, \mathsf{p_i}>$ to all the processes $p_k : \{i < k < j\}$, as p_i knows that p_j suspected all these process (p_k).

When a process p_i receives a message $<\mathsf{Alive?}, \mathsf{p_j}>$, p_i replies by sending to p_j, through a reliable link abstraction, $<\mathsf{heartbeat}, \mathsf{L(p_i)}, \mathsf{p_i}>$.

When a process p_i receives $<\mathsf{heartbeat}, \mathsf{L(p_j)}, \mathsf{p_j}>$, p_i checks if p_j precedes or succeeds it in the list. If p_j precedes p_i in the logical linear list (i.e., $j < i$) and succeeds (or is) the current predecessor of p_i, then $pred(p_i)$ and the set of suspected processes $L(p_i)$ are updated accordingly. Similarly, if p_j succeeds p_i in the logical linear list and precedes the current successor of p_i, then $succ(p_i)$ is updated.

Proof of Correctness of Algorithm 1

We first prove that Algorithm 1 implements $\diamond\mathcal{S}^*$, then we prove it is optimal. From the description of the algorithm, strong completeness is guaranteed if a crashed process p_i is suspected by all correct processes that follow it in the list within T_D time slots, i.e., by all $p_j : \{j > i\}$.

Lemma 14. *The first correct process p_j, succeeding a crashed process p_i, eventually suspects p_i permanently.*

Proof. Let us denote by t the time at which p_i crashes. By lines (11-14) of Algorithm 1 guarantees that p_j will eventually set p_i as its predecessor and will monitor it. If p_j suspects p_i before time t then if p_j hears no messages from p_i it will suspect it forever. However if p_j hears a message from p_i, then by lines (26-33) of Algorithm 1 p_j will monitor p_i again and will eventually suspect p_i by (11-14) some time after t. Since after time t, p_i will no longer send any messages, then p_i will be suspected forever by p_j. $\square$

Lemma 15. *The successor of a correct process p_i will eventually be (when links behave timely) the first correct process following p_i.*

Proof. Let us denote by p_j the first correct process that follows p_i. By lines (11-14) p_j will stop monitoring p_i and will monitor other processes only if p_j suspects p_i. However, p_j sends a suspicion messages through reliable link abstraction. Since both p_i and p_j are correct the suspicion will eventually reach p_i which by lines (16-21) will send p_j a heartbeat message through a reliable link abstraction and will set p_j as its successor. Again by the fact that the two processes are correct p_j will receive this heartbeat message and by lines (26-33) will monitor p_i again. Thus if links are timely p_j will not "time-out" on p_i. $\square$

By Lemma 15 and lines (26-33), the suspected list of a correct process p_i is propagated to all correct processes following p_i in the logical linear arrangement. By lines (35-37) all process following a crashed process will eventually suspect that crashed process permanently ensuring strong completeness. Now we prove the strong bounded completeness property of Algorithm 1, i.e., a failed process is permanently suspected by all correct processes after some bounded duration of having failed. The longest delay of suspecting a crashed process would be when the tail of the logical list has to detect the crash of the head of the list. It is important to note the following: if process p_i is monitoring process p_j, then p_i can detect the failure of p_j after "*timeout*" time slots of not hearing from p_j. For presentation simplicity, we consider a network of three process p_1 being the head and p_3 being the tail. We accordingly show that p_3 detects the failure of p_1 within a bounded duration which we compute. By induction and transitivity this could be extended to a general network of n processes.

Assume that p_1 fails at time t. Recall also that every process sends a heartbeat message at each time slot to its successor. In that case, p_2, the process monitoring p_1, permanently suspects p_1 after "timeout" time slots of not hearing from p_1. Given that a successful message transmission (not lost) between a pair of processes takes one time slot. This means that p_2 suspects p_1 in the interval $[t + timeout + 1, \infty]$ and thus within a bounded delay of "$timeout + 1$". p_3 can detect the crash of p_1 in two cases: (i) via p_2 (by seeing that p_1 is in the suspected list of p_2) or (ii) directly from p_1. The time taken for p_3 to detect p_1 in case (i) would be $timeout + 1 + T_h$, where $T_h < timeout + 1$. While in case (ii) p_3 has to suspect p_2 first, after which it monitors p_1 and suspects it. In that case p_3 would suspect p_1 in $2(timeout + 1)$ time slots of not hearing from p_2. The worst-case delay for p_3 to permanently suspect p_1 would thus that p_3 keeps hearing from p_2 until time instant "$t + timeout + 1$" and then does not hear from p_2 for a duration longer than $2(timeout + 1)$. This results in p_3 permanently suspecting p_1 in the interval $[t + 3(timeout + 1), \infty]$.

As a consequence, a failed process would be permanently suspected by all correct processes within a maximum of $3(timeout + 1)$ time slots after having failed. This proves the strong bounded completeness of Algorithm 1, given three processes.

Lemma 16. *When links are timely, all correct processes will trust the correct process at the head of the logical linear arrangement list.*

Proof. By Lemma 14, when links are timely every correct process is monitored by the first correct process following it in the list. Since the first correct process (at the head of list) does not get suspected by the processes monitoring it (as a consequence of links being timely), by Lemma 15 all correct process will eventually adapt its list of suspected processes which it is not included in by lines (35-37). □

The probabilistic accuracy can be insured by Lemma 16. At any point in time Lemma 16 has a positive probability of happening. Since we consider infinite time instants, then with probability 1 the following happens: there are infinitely many time instants t_G such that after each instant links in the network behave timely for the interval $[t_G, t_G + \Delta t]$, where Δt is any finite duration.

Now we show that Algorithm 1 is optimal. Let T be the time after which all faulty processes have crashed. Then after T and by Lemma 15, whenever the links become timely for any finite time the set of links sending heartbeat messages will be either $\mathcal{C} - 1$ if process p_n (at the tail of the list) is correct or $\mathcal{C}$ if p_n is faulty.

5.5 Decisive Problems

In this section, we discuss what happens to deterministic algorithms using $\diamond\mathcal{S}$ to solve *decisive problems*, e.g., consensus (we give examples of decisive problems beyond consensus in Section 5.5.2), when put in $\mathcal{N}$ which provides $\diamond\mathcal{S}^*$ guarantees instead.

Definition 3. *A decisive problem is a problem which can be solved when a single irrevocable global decision is reached. Any decisive problem P requires that both of the following two properties are satisfied: (i) Termination: there is a point in time after which every correct process will have decided and (ii) Integrity: No process can decide more than once.*

Clearly consensus is one such problem, as the consensus abstraction guarantees: (i) *Validity:* A value decided is a value proposed, (ii) *Integrity:* No process decides more than once, (iii) *Agreement:* No two processes decide differently and (iv) *Termination:* there is a point in time after which all correct processes would have decided. For illustration, we first focus on consensus, then we discuss decisive problems.

5.5.1 Consensus with $\diamond\mathcal{S}^*$

We first show, for an exemplary existing consensus algorithm, that $\diamond\mathcal{S}^*$ can replace $\diamond\mathcal{S}$: the result would be solving "consensus with probability 1", in system $\mathcal{N}$. Then we present a general form of this result.

A Rotating Coordinator Algorithm. The basic idea behind the seminal rotating coordinator algorithm of [7] is that processes alternate in a role of "leader" until one of them succeeds in imposing a decision. The algorithm assumes a correct majority and uses two abstractions: (i) reliable links and (ii) reliable broadcast. Both reliable links and reliable broadcast can be implemented in our system $\mathcal{N}$ (in the sense specified in Section 5.2).

The algorithm is round-based, i.e., the processes move incrementally from one round to the other. Process p_i is the "leader" of every round $k : k \mod n = i$. In such a round, process p_i does the following: (i) p_i selects among a majority the latest adopted value (latest w.r.t. round), (ii) p_i sends that value to all processes and waits for the acknowledgment of the majority and (iii) once p_i succeeds in imposing that value on a majority, p_i uses reliable broadcast to send its decision to all and decides.

It is important to note that p_i succeeds if it is not suspected by the majority (processes that suspect p_i inform p_i and move to the next round, including p_i).

Theorem 14. *The algorithm of [7] implements "consensus with probability 1" in $\mathcal{N}$ using $\diamond\mathcal{S}^*$ (instead of $\diamond\mathcal{S}$).*

Proof. It is easy to see that $\diamond\mathcal{S}^*$ guarantees the strong completeness of $\diamond\mathcal{S}$ and that the reliable links and reliable broadcast in system $\mathcal{N}$ (see Section 5.2) guarantee respectively the properties of the reliable links and reliable broadcast depicted in [7]. Thus the proof of correctness provided in [7] remains true, except for the parts relying on the accuracy of $\diamond\mathcal{S}$, namely termination. Thus it is sufficient to prove that the accuracy of $\diamond\mathcal{S}^*$ guarantees that all processes decide.

Consider t_{rand} to be any point in time after all faulty processes have crashed. Since the algorithm of [7] operates in asynchronous rounds, then at time t_{rand} processes might be at different rounds. We denote by r the largest round among all processes at time t_{rand} and by $\Delta r_{t_{rand}}$ the maximum difference between the rounds of the processes at time t_{rand}. Note that from [7], $\Delta r_{t_{rand}} \leq n$, n being the total number of processes in the system.

In the algorithm of [7], after time t_{rand}, a process, be it a leader or not, completes a round when a bounded number of messages (unicast or broadcast messages) is sent/received or when it suspects the leader of that round. Let M be the maximum number of messages for a process to complete a round.

Let $T_M \geq T_D$ be the amount time for exchanging the M messages, such that the probability of exchanging M messages in T_M time slots is positive (the properties of reliable links and reliable broadcast primitive defined in Section 5.2 guarantee the existence of such a T_M). Recall that the accuracy of $\diamond\mathcal{S}^*$ guarantees that with probability 1 the following holds: for any finite duration Δt, there exists infinitely many time instants $t_G \in \{1, ..., \infty\}$ such that some correct process, say q, is not suspected by all correct processes for the interval $[t_G, t_G + \Delta t]$.

Consider now $r' \geq r$ to be the round in which q becomes leader. Thus with positive probability, all processes can reach round r' after $T_M \cdot (\Delta r_{t_{rand}} + (r' - r))$ time slots from t_{rand}. If q is not suspected by any of the processes, then with positive probability every process decides after T_M time slots from reaching round r'. In other words, if process q is not suspected by any correct process in the interval $[t_{rand}, t_{rand} + T_M \cdot (\Delta r_{t_{rand}} + (r' - r) + 1)]$, then there is a positive probability that all processes decide.

Since t_{rand} is any point in time, after all processes have crashed, then we can assume $t_{rand} = t_G$, such that $\Delta t = T_M \cdot (\Delta r_{t_{rand}} + (r' - r) + 1)$. By the accuracy of $\diamond\mathcal{S}^*$, with probability 1: there exists infinitely many t_G time instants (after all faulty processes have crashed) such that process q is not suspected by all correct processes for the interval $[t_G, T_G + T_M \cdot (\Delta r_{t_{rand}} + (r' - r) + 1)]$.

Since there is a positive probability of all processes deciding $T_M \cdot (\Delta r_{t_{rand}} + (r' - r) + 1)$ time slots after t_G and there are infinitely many t_G time instants, then with probability 1 we have the following: there is a point in time after which all correct processes would have decided.

□

Definition 4. *An asynchronous algorithm $\mathcal{A}$ that solves a decisive problem P is said to be $\diamond\mathcal{S}^{\mathcal{N}}$-bounded if $\mathcal{A}$ satisfies the following properties:*

1. *$\mathcal{A}$ uses as external blocks only the failure detector $\diamond\mathcal{S}$ and communication primitives implementable in $\mathcal{N}$, such as reliable links and reliable broadcast (see specification in Section 5.2).*
2. *Consider that there exists a point in time, t_G, after which some correct process is never suspected by all correct processes. Then $\mathcal{A}$ needs a bounded number of messages to be sent after t_G and until P is solved (i.e., all correct processes decide).*

In fact many of the consensus algorithms using $\diamond\mathcal{S}$ in the literature are $\diamond\mathcal{S}^{\mathcal{N}}$-bounded. This makes our results applicable to wide range of existing algorithms.

Theorem 15. *Any asynchronous algorithm that uses $\diamond\mathcal{S}$ to solve consensus and is $\diamond\mathcal{S}^{\mathcal{N}}$-bounded, solves "consensus with probability 1" in $\mathcal{N}$ when using $\diamond\mathcal{S}^*$ instead.*

Proof can be seen for the more general result of Theorem 16.

5.5.2 Decisive Problems with $\diamond\mathcal{S}^*$

Now we generalize the result of Theorem 15 for *decisive problems* in general.

Theorem 16. *Any asynchronous algorithm $\mathcal{A}$ that uses $\diamond\mathcal{S}$ to solve a decisive problem P and is $\diamond\mathcal{S}$-bounded, solves "P with probability 1" in $\mathcal{N}$, when $\diamond\mathcal{S}^*$ is used instead* [6].

Proof. $\diamond\mathcal{S}^*$ guarantees the strong completeness of $\diamond\mathcal{S}$. Thus w.r.t. $\mathcal{A}$, the difference between $\diamond\mathcal{S}^*$ and $\diamond\mathcal{S}$ is in the provided accuracy property. The accuracy of $\diamond\mathcal{S}$ is a property which holds at some unknown point in time. As a result, any algorithm $\mathcal{A}$ that solves a decisive problem P using $\diamond\mathcal{S}$, guarantees all safety properties required by P regardless of the accuracy of $\diamond\mathcal{S}^*$. $\mathcal{A}$ hence uses the accuracy of $\diamond\mathcal{S}$ to guarantee liveness, in particular termination, i.e., there is a time after which all correct processes decide. It thus suffices to prove that with respect to $\mathcal{A}$ and with probability 1 the following is satisfied: The accuracy of $\diamond\mathcal{S}^*$ guarantees that there is a point in time after which all processes decide.

Assume the existence of an external clock (not accessible but merely used as a reference to clarify the proof construction). Let t_{start} denote the time instant at which $\mathcal{A}$ starts executing. Using $\diamond\mathcal{S}$ in $\mathcal{A}$ to solve P implies that after t_{start} there is a time when all processes decide and P is solved (see Definition 3). Precisely, after some correct process is never suspected by all correct processes, all correct processes executing $\mathcal{A}$ should exchange a finite bounded number of messages after which P would be solved.

[6] Solving "P with probability 1" means guaranteeing all safety properties of P deterministically and ensuring termination with probability 1.

Let M denote the upper bound on the number of messages (be them uni-casts or broadcasts) needed by $\mathcal{A}$ from the time some correct process is never suspected by all correct processes until P is solved. All events that could occur have a bounded delay (process speeds, crash detection, etc.), except for reliable message transmissions (be them uni-casts or broadcasts). Using communication primitives as the reliable links and reliable broadcast in $\mathcal{N}$, the delay for delivering a single message may be arbitrarily long. However it is possible, with positive probability, that a message gets delivered after a known fixed delay, e.g., in x time slots after being sent (see reliable transmission Section 5.2). Thus and without loss of generality, at any point in time where some correct process in never suspected by all correct processes, it is possible (with positive probability) for the M messages to be exchanged within a known fixed duration, say T_M, after which all processes would have decided.

From Definition 1, the accuracy of $\diamond\mathcal{S}^*$ guarantees with probability 1 that: there exists infinitely many time instants t_G after which a unique correct process is not suspected by any correct process for the interval $t_G + T_M$. Since there are infinitely many such t_G time instants and at each t_G there is a positive probability for the M messages to be exchanged within T_M, then with probability 1 the following happens: there is a time after which all processes would have decided within T_M time slots and thus P would be solved. $\square$

Other Decisive Problems. Besides consensus, there can be many other decisive problems, e.g., non-blocking atomic commit (NBAC), $k-$set agreement, fast consensus. Some of these decisive problems, such as NBAC, are solved using $\diamond\mathcal{P}$. In Appendix B.3 we show that it is possible to formulate $\diamond\mathcal{P}^*$, a variant of $\diamond\mathcal{P}$ (in the main part of the paper we concentrate on $\diamond\mathcal{S}$). We also show in Appendix B.3 that Theorem 16 can be extended to the set of decisive problems solvable with $\diamond P$ when replaced by $\diamond P^*$, thus covering a wider set of problems, besides consensus.

5.6 Existing Failure Detector-Consensus Algorithms $\&$ Assumptions

Fault-tolerance has been addressed in many domains and at many levels [83, 118–122], for example, to predict failures and figure out their sources and patterns [123–127], to detect transient process failures [128], etc. We briefly survey in this section, existing work on failure detectors. We also mention some related work on consensus with probability 1 and in systems with message losses. When necessary, we compare and contrast between existing work and the work presented in this chapter.

5.6.1 Failure Detectors

Minimal Synchrony

A large body of work studied Ω, a failure detector abstraction equivalent to $\diamond\mathcal{S}$ [129], and its implementation under different synchrony assumptions [104–106, 108, 109]. These implementations either posed assumptions on the behavior of correct and faulty processes or required links (or a subset of links) to be reliable, fair and/or eventually timely. See Table 5.1 for a summary of the assumptions adopted by systems implementing Ω or $\diamond\mathcal{S}$.

In contrast, we focus on $\mathcal{N}$, a system which, from the failure detection point of view, is weaker than the systems considered above. In fact, we prove that $\diamond\mathcal{S}$ properties cannot be guaranteed, with probability 1, in $\mathcal{N}$. We investigate properties of failure detectors that are implementable in $\mathcal{N}$.

Minimal Communication

Another track of research addressed the communication overhead of failure detector implementations [105, 130–133]. Aguilera et al. [105] showed that an implementation of Ω in a system S (see Table 5.1) requires all processes to periodically send messages and the minimum number of links carrying messages forever can be at least $(n^2 - 1)/4$. On the other hand, stronger systems S^+ and S^{++} (Table 5.1) allow *efficient* implementations, where only one process broadcasts messages forever on $n - 1$ links. For systems with eventually timely correct processes and links (also reliable), Larrea et al. [132, 134] provided algorithms for $\diamond\mathcal{S}$ and $\diamond\mathcal{P}$, where $2n$ links carry messages forever in the worst case, and for Ω where $n - 1$ links carry messages forever respectively. Follow-up work [130, 131] defined communication optimality: in systems with at least one faulty process, the number of correct processes $\mathcal{C}$ equals the minimum number of links necessary to implement $\diamond\mathcal{P}$, $\diamond\mathcal{S}$ and Ω. If the correct process with the smallest id has eventually timely output links, communication-optimal implementations of $\diamond\mathcal{S}$ and $\diamond\mathcal{P}$ exist [133].

Our work in this chapter is different, as it investigates the communication overhead of $\diamond\mathcal{S}^*$, a weaker variant of $\diamond\mathcal{S}$, in system $\mathcal{N}$, where links never become timely forever. We show that when links in $\mathcal{N}$ start behaving in a timely fashion for some interval, the number of links carrying messages in our implementation of $\diamond\mathcal{S}^*$ will converge to $\mathcal{C}$, possibly $\mathcal{C} - 1$. Thus in the best case (i.e., $\mathcal{C} - 1$) our implementation of $\diamond\mathcal{S}^*$ in $\mathcal{N}$, where $n - 1$ processes can crash, circumvents the bound for $\diamond\mathcal{S}$ when at least one process crashes using $\mathcal{C}$ links [131]. Our $\diamond\mathcal{S}^*$ algorithm is inspired by the ring structure algorithms [131] for communication-optimal implementations of $\diamond\mathcal{P}$. However, our implementation is different as it assumes a linear arrangement of processes and manages suspicions differently to accommodate for properties of $\diamond\mathcal{S}^*$.

Failure Detectors with Probabilistic Guarantees

A different line of research explored failure detector implementations with probabilistic guarantees. Chen et al. in [21] studied the quality of service (QoS) of failure detectors in systems where message delays and message losses follow probability distributions. They proposed a set of metrics, among them (i) how fast actual failures are detected and (ii) how well false detections are avoided. In [16], Bertier et al. proposed an implementation supporting a short detection time based on estimations of the expected arrival of monitoring messages, and on adapting QoS to the specific application needs. In [18], Gupta et al. quantified the optimal network load of

Table 5.1 – Assumptions of systems considered when implementing failure detectors.

System	Properties
$\mathcal{N}$ *(this chapter)*	*All links* lose messages probabilistically and fairly *Propagation delay*, in case of no loss, is bounded *Processes* can crash and processing delays are negligible w.r.t. communication delay
S [105]	*Processes* can be arbitrarily slow and can crash, but have a max. execution speed *Links* can be arbitrarily slow and lossy with at least one eventually timely source (i.e., a timely correct process whose *output* links are eventually timely)
S^+ [105]	S with at least one correct process whose *input and output* links are fair
S^{++} [105]	S and such that all links are fair
[130–132]	*All links* are reliable and eventually timely
[108]	*All links* are reliable. Some correct process hears within interval δ from f other processes (f is the maximum number of processes that can crash)
[104]	*All links* are fair and there is one correct process with f output links eventually timely
[109]	*Links* can lose or delay messages At least one correct process that can reach all other correct processes through eventually timely links

failure detector algorithms as a function of the failure detection time and the probability of falsely suspecting a correct process. Hayashibara et al. [17] proposed φ-failure detectors capable of adapting to the application requirements and network conditions dynamically, by assigning a value to every known process representing the confidence that it is alive.

In contrast, we evaluate the eventual guarantees of (binary) failure detectors implemented in systems with probabilistic message losses. In particular we investigate implications of such guarantees to solving distributed computing problems, namely consensus. We also study the efficiency of implementing these failure detectors from a communication overhead perspective, defining optimality in terms of the number of links that need to carry messages forever rather than evaluating the real-time performance.

Rethinking Failure Detection

Several researchers challenged (directly or indirectly) the failure detection approach. Biely et al. [135] showed that the asynchronous model augmented with Ω is equivalent to several models where the links from at least one process (the source) are timely. In comparison with Biely et al. [135], our work avoids the necessity of eventually timely link(s), for the solvability of problems such as consensus in the presence of asynchrony.

Charron-Bost et al. [111] highlighted a paradox about the failure detection approach which we re-affirm, in a non-deterministic environment, in this chapter: Dwork et al. [62] and Lamport [5] showed that sufficiently long finite "good" periods make consensus solvable while Chandra et al. [110] show that consensus cannot be solved without permanent agreement on a leader from some time on. Given this inconsistency in results, Charron-Bost et al. [111] showed that the "discrepancy" is due to the two-layered structure of the failure detector approach itself. Precisely, authors attribute this "artificial difficulty" to the interface between the failure detector layer and the asynchronous system layer to which the failure detector is augmented and the lack of timing control by failure detectors on the asynchronous system. Charron-Bost et al. [111] concluded that it may be better to look at consensus without using failure detectors. Avoiding any dependence on real time, Cornejo et al. [136] proposed *asynchronous failure detectors* (AFDs) and used them to address challenges related to the hierarchy robustness of failure detectors. They also investigated the relationship between the weakest failure detector and partial synchrony. They showed that a large class of problems, termed as *finite problems*, such as consensus, do not encode the same information about process crashes as their weakest failure detectors do (another validation of the observation in [111]).

In this chapter, we take an opposite route compared to [111] and [136]. We refine the notion of a failure detector with explicit dependence on real time to address a similar paradox as that of [111]: "consensus with probability 1" can be implemented in system $\mathcal{N}$ (without the need for randomization within the algorithm) while "$\diamond\mathcal{S}$ with probability 1" is impossible to implement in $\mathcal{N}$. We define $\diamond\mathcal{S}^*$, a variant of $\diamond\mathcal{S}$ implementable in $\mathcal{N}$ and show that such a transformation in the failure detector notion allows deterministic consensus algorithms based on failure detectors to solve "consensus with probability 1" in poorly behaved systems. In this sense, we succeed to repress the "artificial difficulty" highlighted in [111] about the failure detector model, showing that "consensus with probability 1" can be solved in poorly behaving systems (as viewed from a networking perspective) using failure detectors, without requiring an eventually forever agreement on a process that will never crash.

5.6.2 Consensus

Omission Faults

Some researchers explored consensus in systems with message losses without relying on eventual guarantees. Santoro and Widmayer [137] showed that consensus is impossible if $n-1$ of the n^2 possible messages sent in a round can be lost. In contrast, Schmid et al. [138] showed that consensus can be solved even in the presence of $O(n^2)$ moving omission and/or arbitrary link failures per round, provided that both the number of affected outgoing and incoming links of every process is bounded and that all processes are correct. Soraluze et al. [139] considered the general omission model, where processes can fail either by permanently crashing or by omitting messages.They defined a failure detector requiring the existence of a majority of *well-connected* processes, which do not crash, and are able to communicate in both directions and without omissions, either directly or indirectly, with a majority of processes.

In contrast, in this chapter we do not bound the number of messages lost at any point in time and does not require any process to be connected at all times with any other process, yet we can implement failure detectors and consensus.

Randomized Consensus Algorithms

Randomized algorithms ensuring "consensus with probability 1" have also been explored. Approaches based on *coin-flips* [140–142] or *probabilistic schedulers* [114] lead to consensus algorithms with probabilistic factors. In systems with dynamic communication failures, multiple randomized algorithms [99, 100] addressed the k*-consensus* problem, which requires only k processes to eventually decide. Moniz et al. [99] considered a system with correct processes and a bound on the number of faulty transmission. In a wireless setting, where multiple processes share a communication channel, Moniz et al. [100] devise an algorithm tolerating up to f Byzantine processes and requires a bound on the number of omission faults affecting correct processes.

Our work does not employ randomization in the algorithm, we focus on deterministic algorithms in probabilistic networks. Moreover, instead of designing new consensus algorithms, we reuse existing deterministic algorithms relying on failure detectors to solve "consensus with probability 1" in $\mathcal{N}$.

5.7 Chapter Summary

We investigated failure detection in systems embodying asynchrony via probabilistic synchronous communication. In contrast to the conventional distributed computing assumptions when building failure detectors, which hinged on link synchrony guarantees that need to hold deterministically forever, we adopted a more realistic link behavior motivated by networking views on actual packet loss. We showed that "$\diamond\mathcal{S}$ with probability 1" cannot be implemented given such link behavior ($\diamond\mathcal{S}$ being established as the weakest failure detector to implement consensus), despite the fact that "consensus with probability 1" can be implemented without requiring any randomness in the algorithm itself.

This suggests two things: (i) $\diamond\mathcal{S}$ is somehow too strong for consensus[7], at least in a probabilistic environment as $\mathcal{N}$ and (ii) deterministic algorithms solving consensus using $\diamond\mathcal{S}$, cannot be practically put in use to solve consensus in systems like $\mathcal{N}$, as $\diamond\mathcal{S}$ itself cannot be guaranteed in $\mathcal{N}$. This "consensus/failure detector" paradox can be viewed as a re-affirmation of the paradox highlighted in [111], this time in a non-deterministic environment. However, in contrast with [111], we followed a different route.

[7]It is important to recall that $\diamond\mathcal{S}$ is the weakest, *amongst all failure detectors*, to solve consensus [110]. This does not mean however that $\diamond\mathcal{S}$ is equivalent to consensus in a computability sense: one cannot implement $\diamond\mathcal{S}$ from consensus.

Whereas [111] suggested to abandon the failure detector approach, we proposed instead a way to circumvent the paradox by refining the failure detector notion itself, while preserving its usefulness as an algorithmic building block. We defined a probabilistic failure detector $\diamond\mathcal{S}^*$. Roughly speaking, $\diamond\mathcal{S}^*$ requires, with probability 1, that the properties of $\diamond\mathcal{S}$ are satisfied for periods that can be arbitrary long. We proved that $\diamond\mathcal{S}^*$ can be implemented in $\mathcal{N}$, which implies, at least from a failure detector perspective, that our system $\mathcal{N}$ is weaker than the systems considered so far to build $\diamond\mathcal{S}$-like failure detectors [104–109]. More importantly, we showed that the celebrated rotating coordinator algorithm of [7] to solve consensus using $\diamond\mathcal{S}$, actually solves "consensus with probability 1" in $\mathcal{N}$, when $\diamond\mathcal{S}^*$ is used instead of $\diamond\mathcal{S}$. We then generalized this result, in three directions: (i) to hold for all deterministic consensus algorithms satisfying some $\diamond\mathcal{S}^{\mathcal{N}}$-bounded condition (Section 3) (not only the rotating coordinator algorithm), (ii) to hold for all *decisive problems* (Section 5.5), not only consensus and (iii) to hold for other failure detectors, e.g., $\diamond\mathcal{P}$.

In particular, we showed that any deterministic algorithm, which solves a *decisive problem* using $\diamond\mathcal{S}$ ($\diamond\mathcal{P}$) and is $\diamond\mathcal{S}^{\mathcal{N}}$($\diamond\mathcal{P}^{\mathcal{N}}$)-bounded, can be re-used in system $\mathcal{N}$ to solve the same problem, ensuring termination with probability 1: the result is reached by using $\diamond\mathcal{S}^*$ ($\diamond\mathcal{P}^*$) instead.

In a nutshell, this chapter provided the following:

1. A way to circumvent the "consensus/failure detector" paradox in systems with probabilistic synchronous communication ($\mathcal{N}$): we define $\diamond\mathcal{S}^*$ a probabilistic failure detector with accuracy ensured for periods that can be arbitrary long. $\diamond\mathcal{S}^*$ can be implemented in systems like $\mathcal{N}$.

2. An optimal implementation of $\diamond\mathcal{S}^*$. The optimal implementation hinges on a logical linear arrangement of the processes. When links behave in a timely manner (i.e., the delay of a reliable message transmission respects some bound), the number of links carrying messages infinitely often converges to using $\mathcal{C}$ (the number of correct processes), possibly $\mathcal{C}-1$. In the best case, i.e., $\mathcal{C}-1$, our implementation of $\diamond\mathcal{S}^*$ achieves, to the best of our knowledge, the lowest communication overhead compared to all known $\diamond\mathcal{S}$ implementations.

3. For all decisive problems P, beyond consensus (see Section 5.5.2 for decisive problems), we enable existing deterministic protocols which use $\diamond\mathcal{S}$ and $\diamond\mathcal{P}$ and which are $\diamond\mathcal{S}^{\mathcal{N}}$($\diamond\mathcal{P}^{\mathcal{N}}$)-bounded, to be reused in $\mathcal{N}$ to solve "P with probability 1": simply using $\diamond\mathcal{S}^*$ and $\diamond\mathcal{P}^*$ instead of $\diamond\mathcal{S}$ and $\diamond\mathcal{P}$ respectively leads to the desired result. In this sense, our approach succeeds in encapsulating the randomization of the probabilistic link behavior in the very abstraction of failure detection (without affecting the deterministic algorithms built on top), bridging the gap between distributed computing and networking practices.

6 Real-Time Membership

The previous chapter studied the failure detector abstraction. In its classical form, the output of failure detectors at different hosts[1] (processes) might not always be the same, as failure notifications might be received at different times or in different orders. However, to guarantee better dependability, distributed control systems (DCS) require failure notifications to be synchronized. In other words, hosts in DCSs need to agree about which hosts in the system have crashed, while detecting crashes in real time.

To this end, we investigate how to build a real-time membership abstraction (service), as a means of providing a better coordinated output than failure detectors, capable of detecting failures within a known fixed bound after failure.

This chapter proves that implementing the necessary membership properties for DCSs deterministically is impossible, when considering both, host crashes and message losses. However, we propose an algorithm, which we call *ViewSnoop*, capable of implementing the required membership properties with high probability. We show that our *ViewSnoop* algorithm provides a better dependability to DCSs compared to existing membership algorithms relying on classic heartbeats, at low additional cost. This improvement is shown both analytically as well as experimentally via an implementation in a production DCS.

6.1 Motivation

Among the many financial hazards of industrial plants, *downtime* (the time during which a plant's normal operation is halted) is one of the most expensive ($\approx$ 12,500\$/hr) [143]. *Automated control systems*, which manage these plants, hence require increased dependability. An automated control system comprises a set of control applications; each being a program that handles (parts of) an industrial system and generally adheres to hard real-time constraints. In order to tolerate crashes, control systems are often decentralized [144–147]. Such *distributed control systems* (DCSs) require however to learn about hosts (processes) which have crashed in order to initiate proper recovery measures.

[1]In the context of distributed control systems, we use the term "hosts" to refer to processes, in coherence with existing literature [12, 13].

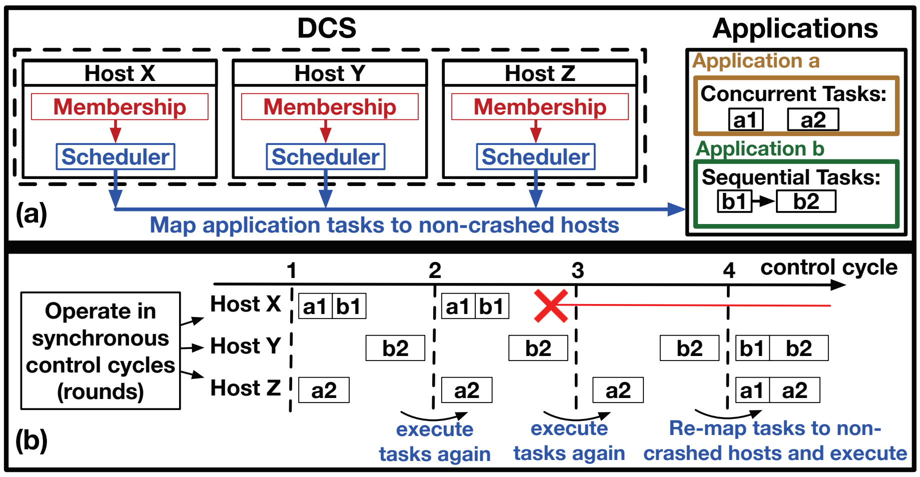

Figure 6.1 – A DCS with three hosts running two control applications.

Most control applications running on DCSs are cyclic [12, 148, 149]. Such applications consist of several small *tasks* that execute periodically. Some of these tasks run concurrently on several hosts, possibly on behalf of different control applications. A *scheduler*, a distributed DCS module, typically maps tasks to non-crashed hosts and specifies the order in which these tasks execute (Figure 7.1(a)).

A DCS cannot avoid host crashes and message losses [150–152]. Control systems typically experience host crash rates of about 10^{-5}/hr and link failure rates in the range of $10^{-5}/hr$ (permanent failures) and $10^{-3}/hr$ (transient failures) [35, 36]. In order to "recover" from crashed hosts, DCSs need to know about these crashes and react in real-time; for example, to have the scheduler re-map tasks to non-crashed hosts, ensuring proper execution of all applications [153] (see Figure 7.1(b)).

Besides the real-time necessity, hosts in DCSs need to have consistent views of which hosts in the system have crashed. Inconsistent views imply that hosts might consider different hosts as crashed. The state of the scheduler module in a DCS, as a result, might be invalidated yielding improper executions of applications and causing downtime [12]. Basically, a DCS needs a *group membership service* [1, 154–159] to coordinate the information regarding crashed hosts in real time.

In short, an ideal membership in DCSs amounts to a service that synchronously reports, to all (non-faulty) hosts, perfect information about host crashes and within fixed bounds (see Figure 6.2). In the presence of message losses, however, implementing such a service deterministically is impossible as we prove in this chapter (Section 6.3).

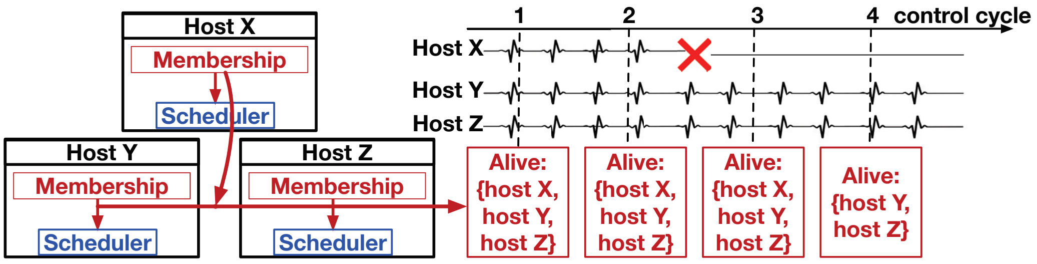

Figure 6.2 – Overview of an ideal membership service for a DCS.

In fact, known implementations of deterministic membership services either assume no message losses, provide eventual (not real-time) guarantees or use additional help [1, 160–163]. The implementable membership guarantees in contexts similar to DCSs can, at best, be probabilistic [35, 155, 164].

This chapter investigates how to define a membership with probabilistic properties suited for DCSs and establishes accordingly a *synchronous membership service* (SYMS), a new abstraction encapsulating a probabilistic form of the ideal membership properties needed by DCSs. In this chapter, we also propose an algorithm, which we call *ViewSnoop*, that implements SYMS properties with high probability (relative to other membership mechanisms and which persists with increasing system size, see Section 6.5). The main idea underlying *ViewSnoop* is to (a) have hosts maintain local suspicion lists that are not visible to the scheduler of a DCS and at the same time (b) let hosts snoop into each others' local views by modifying the structure of heartbeats, precisely by piggybacking local suspicion lists on heartbeats. Heartbeats are disseminated periodically via a broadcast primitive (not necessarily reliably) as in most DCSs to facilitate crash detection [12, 35, 154–156]. Appending suspicion lists to heartbeats helps hosts know about other alive hosts, despite possible message losses. This property increases the probability of having a global consistent view and hence a better accuracy. Combined with (a), *ViewSnoop* can, with high probability, discern message losses from host crashes, better than using sequence numbers [165] (see Section 6.5). Having losses mistaken for host crashes, and removing correct hosts as a result, not only worsens accuracy[2], but also depletes processing resources, threatening availability. This chapter also shows the performance benefits and trade-offs of *ViewSnoop* compared to existing membership algorithms relying on classic heartbeats, both analytically as well as experimentally in an industrial DCS framework.

We commence, in what follows, by describing in details how DCSs for cyclic control applications function, highlighting the mandatory constraints and requirements for proper operation.

[2]The probability of not excluding non-crashed hosts.

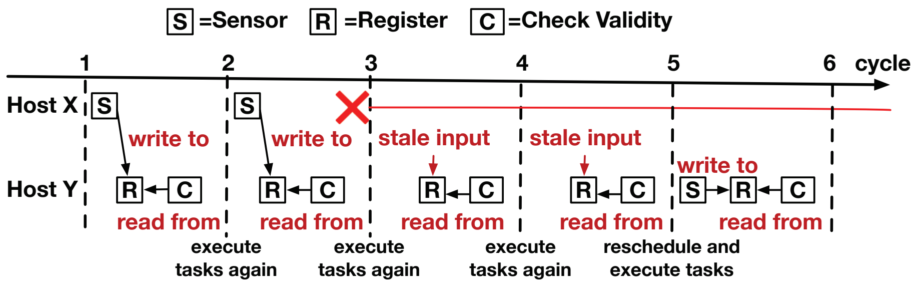

Figure 6.3 – An example of a DCS tolerating two cycles of stale input, which result from the failure of the host responsible to refresh the data.

6.2 Distributed Control Systems For Cyclic Control Applications

A distributed control system (DCS) for cyclic control applications consists of a set of hosts (processes), $\Pi = \{h_1, h_2, ..., h_n\}$, physically mapped to cores of the same or different machines. Clearly, these hosts can *fail (crash)* [35], i.e., stop executing operations. Hosts have access to local synchronized clocks with bounded skew. Accordingly, all hosts define control cycles (rounds) of the same fixed duration. Control cycles are synchronized among hosts, i.e., the start and end of a cycle occur at all hosts at the same time (with a bounded skew). During every control cycle, each host executes the tasks assigned to it by the *scheduler* (recall Figure 7.1).

Scheduler

This is a distributed module that specifies which application tasks run on which hosts and in what order. The allocation of tasks to hosts is called a *configuration*. A scheduler makes sure that all hosts can execute the assigned tasks without exceeding the total cycle duration. Moreover, the scheduler ensures that configurations allow all applications to meet their deadlines (timing constraints). As such, a host requiring some input, in some configuration, expects the value of this input to be refreshed, every cycle (e.g., by another host driving this input, see Figure 6.3). A value that is not refreshed in time is called a *stale input*. Typically in DCSs, hosts can tolerate to read stale input up to a bounded number of consecutive cycles, say s_c. If the input remains stale for more than s_c cycles, then a new configuration has to be installed.

A DCS requires to exclude a crashed host: (i) within a bounded number of cycles after crashing (real-time) and (ii) synchronously at all alive hosts, i.e., in the same cycle. Violating (ii) might lead the scheduler state to be inconsistent, resulting in hosts executing different configurations (mapping of tasks to hosts). Applications, as a result, might execute incorrectly, as communication and/or the order of execution between tasks of the same application might be invalid.

Communication

Every pair of hosts is connected by two logical uni-directional links. Links here for example abstract a physical bus or a dedicated network link. Arguably, all communication is prone to random disturbances resulting from bad channel quality, collisions, stack overflows etc. Messages can thus be lost. When there is no loss, we assume that messages have a bounded delay, say d. Configurations computed by the scheduler account for the delay d. As such, any message scheduled to be sent in cycle r, if not lost, is assumed to be received in the same cycle r.

Specifically, we model losses as follows: a message sent by h_i is received by h_j with probability p. Sending a message reliably at any point in time from one host to another, thus, can take an unbounded amount of time, due to losses and follow-up re-transmissions.

We assume, for the theoretical analysis, that p is independent of time and links and is the same for all links. Nevertheless, correlated losses, although not considered theoretically for the tractability of our analysis, can occur in our experimental evaluation (Section 6.7).

Crash monitoring

In this chapter, we consider monitoring schemes that rely only on message exchange and synchronized local clocks (time-outs). Specifically, a host sends messages, known as heartbeats, every cycle to at least one other host in the system (hosts do not send heartbeats to themselves). We assume no causality between heartbeats sent in the same cycle; the content of heartbeats sent by a host during cycle r is the same and can be affected only by the heartbeats sent at cycles $< r$. A host is “alive” at cycle r, if that host does not crash during r (during r a host is either crashed or alive). Hosts do not crash themselves on purpose, as resources for running tasks become scarcer, risking some applications to halt.

6.3 Overview of SYMS

After our description of the operation of a DCS, we identify now the ideal membership, following the traditional way of defining its properties [1, 160, 161]. For simplicity, we specify the properties for host crashes in the fail-stop model [7], i.e., without recovery (we discuss recoveries in Section 6.7).

We first introduce some terminology. We denote a view by the tuple (id, M), where M is the set of hosts declared in a view as alive (not crashed). Variable id denotes the view identifier (namely the cycle in which the view is installed). Initially all hosts install the view $V = (id, M)$, where M includes all hosts in the system. Consequent views are obtained from monitoring, as described in Section 7.2. We assume that if a host h_i receives a heartbeat sent by a host h_j in round r, then h_i cannot exclude[3] h_j in round $r + 1$.

[3]This assumption is analogous to our assumption that hosts cannot be crashed on purpose, in the sense that, as

The ideal properties for DCSs can be expressed as follows:

P1: *Monotonicity:* If a host installs a view $V = (id, M)$ and later $V' = (id', M')$, then $id < id'$ and $M' \subseteq M$.

P2: *Agreement:* If a host installs a view $V = (id, M)$ at round r, then all alive hosts at r install $V = (id, M)$ at r.

P3: *Completeness:* If a host h crashes, then after a maximum of s_c rounds elapse after the crash, all hosts that remain alive s_c rounds after the crash, install $V = (id, M)$ where $h \notin M$.

P4: *Accuracy* If a host installs view $V = (id, M)$ where $h \notin M$, for some host h, then h has already crashed.

P5: *Non-triviality:* Let $\mathcal{C}$ be the set of hosts alive at round r, during which host h installs a view $V = (id = r, M)$. Then, it is possible that $\{\mathcal{C} \cap M'\} \subseteq M$, where $V' = (id' < r, M')$ is the most recent view h installed before the view at r.

In contrast with traditional membership properties [1, 160, 161], which detect crashes eventually and not necessarily in a synchronous manner (i.e., in the same synchronous round by all hosts), completeness (P3) and accuracy (P2) here stipulate real-time and synchronous detection of crashes respectively.

Theorem 17. *No algorithm can deterministically guarantee both completeness (P3) and accuracy (P4) in a DCS with message losses.*

Proof. Both properties, P3 and P4, are related to failure detection, precisely perfect failure detection[4]. A perfect failure detector [7] cannot be implemented in case of message loss, because finite executions where a host crashes cannot be distinguished from finite executions where all messages from this host are lost [167].

For better illustration and entirety, we showcase this fact again below.

Consider an example of a DCS with two hosts, h_1 and h_2, and the following executions:

- $e1$. an execution where host h_2 fails at cycle r.
- $e2$. an execution where host h_1 and h_2 are both correct but lose all messages sent (if any) at all cycles in the range $[r, r + s_c]$.

long as h_i receives heartbeats from h_j, then h_j is certainly alive (at least up to the moment of sending the last heard heartbeat). Acting otherwise might risk the system's availability (higher downtime) as fewer hosts become available.

[4]P3 is a stronger version of the strong completeness property (defined in [7]), as it has a bound on the detection time (s_c control cycles versus eventually). P4 is the strong accuracy property. P3 and P4 together define a stronger version of the perfect failure detector [166] (a perfect failure detector with a bound on detection time).

Since the control cycle duration is fixed, a finite number of messages can be sent during a control cycle, say n_i. Execution $e2$ is valid, since $e2$ can occur with the positive probability, $(1 - p)^{n_i(s_c+1)}$, p being the probability that a sent message is successfully received. h_1 cannot monitor h_2 (to know if h_2 is alive) except through message exchange and time-outs (see Section 7.2). With respect to h_1 executions $e1$ and $e2$ are indistinguishable during $[r, r + s_c]$, for any finite value of s_c (since h_1 cannot know if h_2 has failed or all messages from h_2 are lost).

By P3, in execution $e1$ h_1 declares h_2 as failed at most by cycle $r + s_c$. Since $e1$ and $e2$ are indistinguishable during $[r, r + s_c]$ then h_1 declares h_2 as failed at most by cycle $r + s_c$ also in $e2$. This violates P4 in execution $e2$. □

Theorem 18. *No algorithm satisfying non-triviality (P5) can deterministically guarantee agreement (P2) and completeness (P3) in a DCS with message losses.*

Proof. Assume by contradiction that an algorithm $\mathscr{A}$ satisfies the non-triviality property (P5) and deterministically guarantees agreement (P2) and completeness (P3) in a DCS with losses.

Hosts in algorithm $\mathscr{A}$ install an initial view as specified by our assumption in Section 6.3, i.e., a view in which no host is excluded.

Lemma 17. *Assuming that no hosts have been excluded, an algorithm $\mathscr{A}$ satisfying P5 means that exactly one of the following cases is true.*

For every host h_j such that $h_j \in \mathcal{C}$, a host h_i installing a view $V(id, M)$ can:

Case 1. *Decide if $h_j \in M$ regardless of any received heartbeats.*

In this case h_i can decide to

(a) Include h_j in all views installed.

(b) Exclude h_j in a randomly chosen cycle.

Case 2. *Decide whether $h_j \in M$ depending on the heartbeats received by h_i.*

Proof. Any view to be installed by any host has to be constructed by the monitoring scheme depicted in Section 7.2.

Assuming that no hosts are excluded (precisely, that no views besides the initial one are installed), P5 can be restated as follows:

Consider some round r in which a host h_i in $\mathscr{A}$ installs a view $V(r, M)$. Then, there is a positive probability that $C \subseteq M$, where $\mathcal{C}$ is the set of hosts alive at round r.

For every host h_j such that $h_j \in \mathcal{C}$, h_i can:

Case 1. Decide if $h_j \in M$ regardless of any received heartbeats.

In this case h_i can decide to:

(a) Include h_j in all views installed.

(b) Exclude h_j in a randomly chosen cycle.

(c) Exclude h_j deterministically in some predetermined cycle r'.

Case 2. Decide whether $h_j \in M$ depending on the heartbeats received by h_i.

h_i cannot decide based on any other means, as Section 7.2 constrains monitoring to be solely based on exchanging heartbeats.

In fact, having h_i decide according to case 1(c) violates P5. Assume that h_j is a correct host, i.e., a host that does not crash. Then having h_i exclude h_j at any cycle r' deterministically, regardless of any received heartbeats, means that the view at r' can never include h_j as alive, which contradicts P5. As such, in order to satisfy P5, one of the other statements should be true. □

We will now show that in a system with two hosts h_1 and h_2, deciding according to Case 1 or Case 2 will not allow P2 and P3 to be satisfied deterministically.

If Case 1(a) is true, then $\mathscr{A}$ would violate completeness (P3): Consider an execution where h_1 is correct, i.e., does not crash during the entire execution of the algorithm, and h_2 crashes at some point. If Statement 1(a) is true, then h_1 would never declare h_2 as crashed.

If Case 1(b) is true then $\mathscr{A}$ might violate agreement (P2): Consider an execution where both hosts h_1 and h_2 are alive. Let r' be the cycle in which h_1 excludes h_2 from its view and let r'' be the cycle in which h_2 excludes itself from its own view. Since r' and r'' are randomly chosen, there is a positive probability that $r' \neq r''$. In that case, hosts install different views and P2 is violated.

If Case 2 is true, the following is a necessary condition to satisfy completeness (P3): a host in $\mathscr{A}$ should exclude some host after not hearing (directly or indirectly) from that host for d_t consecutive cycles, such that $d_t \leq s_c$ (given that every host sends a heartbeat at every cycle to at least one other host). According to our monitoring assumptions in a DCS (Section 7.2) any host sends heartbeats to at least one other host in the system.

In a DCS with two hosts, this means that: in every cycle, h_1 sends heartbeats to h_2 and h_2 sends heartbeats to h_1. In other words, h_1 can receive heartbeats only from h_2 and h_2 can receive heartbeats only from h_1.

Consider the case where h_2 loses all heartbeats sent by h_1 for more than s_c consecutive cycles (which can happen with positive probability). By the completeness property (P3), h_2 should install (after d_t consecutive cycles of loss) a view V excluding h_1. Also, h_2 has to decide whether to include or exclude itself from V. The decision taken by h_2, whether to exclude itself from V or not in that case, is independent of what happens to the heartbeats sent by h_2 to h_1 in the past d_t cycles (i.e., if these heartbeats are lost or not). This statement is valid since in that duration h_2 did not receive any heartbeats and thus cannot know any information.

Similarly, such a scenario can also happen with h_1. Let us refer to such a scenario, which can occur with either hosts, as Scenario S. A host in scenario S installs a view V (excluding the other host) and decides either to exclude itself from V or not. We discuss both cases below.

A host decides to exclude itself in scenario S.

Consider an execution e satisfying both conditions below:

a. h_1 and h_2 correct, i.e., never fail.

b. Starting from cycle r, all heartbeats sent by h_1 to h_2 are lost for $\alpha \cdot s_c$ cycles, i.e., for all cycles in $[r, r + \alpha \cdot s_c]$, $\forall \alpha \geq 1$, while all heartbeats sent by h_2 to h_1, in this same interval, are not lost.

Condition (b) can happen with positive probability (see proof of Theorem 17). In execution e, h_2 cannot hear any heartbeats in $[r, r + \alpha \cdot s_c]$. In this case and by the completeness property (P3), h_2 excludes h_1 and itself at cycle $r + d_t$.

We recall now the following assumption of Section 6.3: if a host h_i receives a heartbeat sent by host h_j at round r, then h_i cannot exclude h_j in round $r + 1$. Since h_1 receives all heartbeats from h_2 (regardless of the content of these heartbeats) in $[r, r + \alpha \cdot s_c]$, h_1 would still include h_2 as alive during cycle $r + d_t$, which violates agreement (P2).

It is very important to note that in $[r, r + d_t]$, h_2 stops obtaining any additional information. This is due to the fact that h_2 receives no heartbeats in that interval. Thus, at beginning of cycle r, h_2 already has all the information it needs upon which it can base its decision of whether to include or exclude itself from the view. Since execution e does not make any assumptions about heartbeats prior to cycle r, this means that the decision of h_2 to exclude itself in e covers all the possible cases in which h_2 might decide to exclude itself.

A host decides not to exclude itself in scenario S.

Consider now an execution e' satisfying both conditions below:

a. h_1 and h_2 correct, i.e., never fail.

b. Starting from cycle r, all heartbeats sent by h_1 to h_2 are lost for $\alpha \cdot s_c$ cycles and all heartbeats sent by h_2 to h_1 are lost for $\alpha \cdot s_c$ cycles, i.e., for all cycles in $[r, r + \alpha \cdot s_c]$, $\forall \alpha \geq 1$.

Condition (b) can happen with positive probability (this can be inferred from the proof of Theorem 17 and the fact that losses are independent of links). In execution e', h_2 cannot hear any heartbeats in $[r, r + \alpha \cdot s_c]$. In this case and by the completeness property (P3), h_2 excludes h_1 at cycle $r + d_t$; hence h_2 installs a view at cycle $r + d_t$ where h_2 considers only itself as alive. Similarly, also by the completeness property (P3) and the fact that a host in scenario S decides not to exclude itself, h_1 installs at cycle $r + d_t$ a view where h_1 is only alive, which violates agreement (P2).

In $[r, r + d_t]$, h_2 stops obtaining any additional information (h_2 receives no heartbeats in that interval). Thus, at beginning of cycle r, h_2 already has all the information it needs upon which it decides to include or exclude itself. The same applies for h_1. Since execution e' does not make any assumptions about heartbeats prior to cycle r, this means that the decision of h_2 not to exclude itself in e' covers all the possible cases in which h_2 might decide not to exclude itself. Combined with the previous case (the case in which h_2 excludes itself at $r + d_t$) we cover all possible cases that might affect h_2's decision.

The same can be constructed for h_1. This result concludes the proof as it shows that an algorithm that satisfies P5 and P3 has a positive probability of violating agreement (P2), given the monitoring families considered in this chapter (Section 7.2). $\square$

Both theorems hold even if only one host can crash.

Given these impossibilities, an implementable form of the desired ideal properties can only be probabilistic. We define such a probabilistic form under an abstraction we call SYMS:

SYMS 1, SYMS 3 and SYMS 5 : respectively as P1, P3 and P5 above.

SYMS 2: If some host installs view $V = (id, M)$ at round r, then with probability p_{agree}, all alive hosts at r install $V = (id, M)$ at round r.

SYMS 4: If some host installs view $V = (id, M)$ such that $h \notin M$, for some host h, then with probability $p_{accurate}$, h has already crashed.

We highlight two probabilistic metrics. The first is p_{agree}, the probability that all alive hosts agree on the view (list of hosts that are considered alive) to be installed at a round. The second is $p_{accurate}$, the probability of an excluded host to have actually crashed. To increase the dependability of a DCS, SYMS algorithms need to maximize both metrics.

6.4 The *ViewSnoop* Algorithm

ViewSnoop implements SYMS by building local suspicion lists above which membership views are constructed . Suspicion lists combined with the process of constructing views allow *ViewSnoop* to detect and act upon stale input resulting from message losses and not only host crashes (details in Section 6.5.4). In particular, *ViewSnoop* seeks to increase the probability of having synchronous consensus on views given message losses and to always detect host crashes in real-time.

Let n_i be the maximum number of heartbeats a host h_i can send in some cycle r (every heartbeat, if not lost, is received in r). For illustration, we assume that n_i is the same at all cycles and for all hosts. We first describe *ViewSnoop* for $s_c = 3$ (consecutive cycles in which a host can tolerate stale data); $s_c = 3$ represents the minimum upper bound on the number of cycles to exclude a crashed host in *ViewSnoop* (as we show below). Later in this section, we discuss how to extend *ViewSnoop* to any value of $s_c \geq 3$.

Every host in *ViewSnoop* maintains a list of suspected hosts ($local_{suspect}$). In each cycle, every host broadcasts a copy of its suspicion list tagged with the control cycle number, as a heartbeat, n_i times to all hosts. At the end of the cycle the list for the next cycle is prepared. In *ViewSnoop*, a view installed at cycle r has $id = r$.

6.4.1 *ViewSnoop*'s Functionalities

Synchronous View Agreement.

This functionality of *ViewSnoop* constructs the view that a host installs in a control cycle. At the end of the control cycle (i.e., after broadcasting the suspicions list), every host performs a *merge* on all the suspicion lists received: the result is a new view to be installed at the beginning of the next cycle. For every host h_j in the current view, a host h_i performs the merge as follows:
If h_j belongs to the $local_{suspect}$ list of h_i and h_j is in the suspected list of all heartbeats received by h_i, then h_i excludes h_j from the view to be installed in the following cycle. Otherwise h_i considers h_j alive.

Consider a host h_j that belongs to the $local_{suspect}$ list of all alive hosts at cycle r. Then the merge guarantees the following: all alive hosts at cycle $r + 1$ exclude h_j. The reason is that alive hosts at $r + 1$, can receive messages (if any is received) of hosts which append their $local_{suspect}$ lists at r.

Host Crashes Detection in Real-time.

ViewSnoop aims at excluding crashed hosts in real-time, i.e., within a fixed number of cycles, s_c. This second functionality of *ViewSnoop* ensures that a crashed host belongs to the $local_{suspect}$ list of all alive hosts (and which remain alive) at most two cycles after crashing. **By satisfying this condition, the synchronous view agreement, precisely the merge, guarantees that the crashed host gets excluded at most one cycle later, i.e., by the third cycle.**

Detecting crashes in real-time relies on the n_i heartbeats sent by every host in every cycle. Initially the $local_{suspect}$ list of host h only contains h. A host h always suspects itself. At the end of a cycle, every host updates its $local_{suspect}$ list based on the non-excluded hosts it hears from during that cycle. For example, if h_i did not receive any message from h_j (which is part of h_i's current view), then h_i places h_j in the $local_{suspect}$ list. Note that placing h_j in the suspected list does not mean that h_j is excluded from h_i's view; excluding hosts is governed by the synchronous view agreement.

h_j, thus, gets **suspected** (not excluded), at most two cycles after crashing, by all hosts that remain alive (since h_j stops sending heartbeats after crashing and might send a heartbeat and directly crash). The merge of Section 6.4.1, **excludes** h_j at most one cycle later, i.e., by the third cycle.

6.4.2 Tolerating $s_c \geq 3$ Stale Control Cycles

To tolerate $s_c \geq 3$ cycles, a host h_i needs an array variable (with one entry per host), $count_{stale}$.

The only modification to *ViewSnoop* is induced in the synchronous view agreement part, precisely the merge operation. Hosts now perform *merge* as follows: h_j is declared failed by h_i, according to the description below (otherwise h_j is declared alive).

```
For every $h_j$
IF ($count_{stale}$(h$_j$)=$s_c$) DO
      Declare $h_j$ failed
ELSE
      IF (cond1 && cond2) DO
      // cond1: $h_j$ belongs to $local_{suspect}$ of $h_i$.
      // cond2: $h_j$ is in the suspected list of all heartbeats received by $h_i$.
         $count_{stale}$(h$_j$)++;
      ELSE
         $count_{stale}$(h$_j$) = 1;
      ENDIF
ENDIF
```

6.4.3 Approximating *ViewSnoop*'s Probabilistic Guarantees

ViewSnoop guarantees property SYMS 1 due to two reasons. First, the *id* of the view to be installed at control cycle r is r, the control cycle number itself. Second, crashed hosts do not recover and if host A excludes host B at cycle r, then A does not install any view at cycles $> r$ where B is considered alive (after excluding B, A ignores any heartbeats from B). SYMS 3 is ensured in *ViewSnoop* of the following reasons:

- A host h that fails at cycle r, stops sending heartbeat message from cycle $r + 1$ onward.
- Since h sends no heartbeats at cycles in $[r + 1, \infty[$, then h is included in the suspected list of all host alive at cycles in $[r + 1, \infty[$.
- As indicated in Section 6.4.1, a host h suspected all alive hosts at cycle $r + 1$ is declared as failed at cycle $r + 2$.

ViewSnoop guarantees property SYMS 5, since: (i) every host broadcasts to all other hosts a heartbeat at every cycle and (ii) a host h_i receiving a heartbeat from a non-excluded host h_j at cycle r declares h_j as alive. In this sense, any host at cycle r has a positive probability (probability of heartbeats being not dropped) of installing a view containing the set of hosts that have not crashed or been excluded up to cycle r.

We determine, in what follows, p_{agree} and $p_{accurate}$ with which *ViewSnoop* satisfies properties SYMS 2 and SYMS 4. We approximate p_{agree} and $p_{accurate}$ in crash-free executions, i.e., only considering false suspicions resulting from message losses, since we do not assume any particular probability for host crashes (we show in Section 6.7 that crashed hosts are excluded by all alive hosts using *ViewSnoop* in at most 3 cycles).

Lemma 18. *The probability, p_{agree}, that all alive hosts in ViewSnoop agree on the view to be installed at a round can be approximated by $p_{agree} = 1 - p_{disagree}$, where:*

$$\begin{aligned}
p_{disagree} &= \sum_{k=1}^{|\mathcal{C}|} \binom{|\mathcal{C}|}{k} \left[Prob(disagree_A)\right]^k \left[1 - Prob(disagree_A)\right]^{|\mathcal{C}|-k} \\
Prob(disagree_A) &= \left[Prob(1|\pi_A) + Prob(2|\pi_A)\right] P(\pi_A) \\
P(\pi_A) &= \sum_{|\mathcal{C}|-|\pi_A|=2}^{|\mathcal{C}|} \binom{|\mathcal{C}|}{|\mathcal{C}| - |\pi_A|} (1-p)^{n_i(|\mathcal{C}|-|\pi_A|)} \left[1-(1-p)^{n_i}\right]^{|\pi_A|}, \\
Prob(1|\pi_A) &= (1-p)^{n_i \times |\pi_A|} \times \sum_{h=1}^{|\mathcal{C}|-|\pi_A|-1} \binom{|\mathcal{C}| - |\pi_A| - 1}{h} \left[1-(1-p)^{n_i(|\pi_A|+1)}\right]^h \\
&\qquad \times \left[(1-p)^{n_i(|\pi_A|+1)}\right]^{|\mathcal{C}|-|\pi_A|-h-1}, \\
Prob(2|\pi_A) &= \left[1-(1-p)^{n_i \times \pi_A}\right] \times \sum_{k=1}^{|\mathcal{C}|-|\pi_A|-1} \binom{|\mathcal{C}| - |\pi_A| - 1}{k} \left[(1-p)^{n_i(|\pi_A|+1)}\right]^k \\
&\qquad \times \left[1-(1-p)^{n_i(|\pi_A|+1)}\right]^{|\mathcal{C}|-|\pi_A|-k-1},
\end{aligned}$$

such that π_A is the the set of hosts which do not have some host A in their $local_{suspect}$ list at the beginning of cycle r and $\mathcal{C}$ is the set of alive hosts in $r-1$, assuming that no host has excluded any other host.

Proof. Consider host A and two sets of hosts:

1. π_A: hosts which do not have A in their $local_{suspect}$ list at the beginning of cycle r.
2. $\pi_{\bar{A}}$: hosts which have A in their $local_{suspect}$ at the beginning of r.

Assume that host A is not included in any of these sets. Let $\mathcal{C}$ be the set of alive hosts in $r-1$ and assume that no host has excluded any other host, then $|\mathcal{C}| = |\pi_A| + |\pi_{\bar{A}}| + 1$. Disagreement in cycle r occurs if any condition below holds:

1. Host A does not receive any message from all hosts in π_A and at least one host in $\pi_A \bigcup \pi_{\bar{A}}$ receives a message from $A \bigcup \pi_A$.

2. At least one host in $\pi_{\bar{A}}$ does not receive any message from all hosts in $A \bigcup \pi_A$ and:

   (a) Host A hears from at least one host in π_A OR

   (b) At least one host in π_A hears from some other host in $A \bigcup \pi_A$.

Condition (1) happens with probability $Prob(1|\pi_A)$:

$$\begin{aligned} Prob(1|\pi_A) =& P(\text{Host A does not receive any message from all hosts in } \pi_A) \\ & \times P(\text{at least one host in } \pi_A \bigcup \pi_{\bar{A}} \text{ receives a message from } A \bigcup \pi_A). \end{aligned}$$

In our computations, we do some approximations for presentation simplicity of closed-form expressions. The value of $Prob(1|\pi_A)$ can be approximated as follows:

$$\begin{aligned} Prob(1|\pi_A) \approx& P(\text{Host A does not receive any message from all hosts in } \pi_A) \\ & \times P(\text{at least one host in } \pi_{\bar{A}} \text{ receives a message from } A \bigcup \pi_A). \end{aligned}$$

$$P(\text{Host A does not receive any message from all hosts in } \pi_A) = (1-p)^{n_i \times |\pi_A|}.$$

$$P(\text{at least one host in } \pi_{\bar{A}} \text{ receives a message from } A \bigcup \pi_A) =$$
$$\sum_{h=1}^{|\mathcal{C}|-|\pi_A|-1} \binom{|\mathcal{C}|-|\pi_A|-1}{h} \left[1-(1-p)^{n_i(|\pi_A|+1)}\right]^h \times \left[(1-p)^{n_i(|\pi_A|+1)}\right]^{|\mathcal{C}|-|\pi_A|-h-1}.$$

So,

$$\begin{aligned} & Prob(1|\pi_A) \approx (1-p)^{n_i \times |\pi_A|} \\ & \times \sum_{h=1}^{|\mathcal{C}|-|\pi_A|-1} \binom{|\mathcal{C}|-|\pi_A|-1}{h} \left[1-(1-p)^{n_i(|\pi_A|+1)}\right]^h \times \left[(1-p)^{n_i(|\pi_A|+1)}\right]^{|\mathcal{C}|-|\pi_A|-h-1}. \end{aligned}$$

Condition (2) happens with probability $Prob(2|\pi_A)$:

$$\begin{aligned} Prob(2|\pi_A) =& P(\text{At least one host in } \pi_{\bar{A}} \text{ does not receive any message from all hosts in } A \bigcup \pi_A) \\ & \times [P(\text{Host A hears from at least one host in } \pi_A) \\ & + P(\text{At least one host in } \pi_A \text{ hears from at least one other host in } A \bigcup \pi_A)]. \end{aligned}$$

We approximate $Prob(2|\pi_A)$ as follows:

$$\begin{aligned} Prob(2|\pi_A) \approx& P(\text{At least one host in } \pi_{\bar{A}} \text{ does not receive any message from all hosts in } A \bigcup \pi_A) \\ & \times P(\text{Host A hears from at least one host in } \pi_A). \end{aligned}$$

$$P(\text{At least one host in } \pi_{\bar{A}} \text{ does not receive any message from all hosts in } A \bigcup \pi_A) =$$
$$\sum_{k=1}^{|\mathcal{C}|-|\pi_A|-1} \binom{|\mathcal{C}| - |\pi_A| - 1}{k} \left[(1-p)^{n_i(|\pi_A|+1)}\right]^k \times \left[1-(1-p)^{n_i(|\pi_A|+1)}\right]^{|\mathcal{C}|-|\pi_A|-k-1}.$$

$$P(\text{Host A hears from at least one host in } \pi_A) = 1 - (1-p)^{n_i \times \pi_A}.$$

So,

$$Prob(2|\pi_A) \approx \left[1-(1-p)^{n_i \times \pi_A}\right]$$
$$\times \sum_{k=1}^{|\mathcal{C}|-|\pi_A|-1} \binom{|\mathcal{C}| - |\pi_A| - 1}{k} \left[(1-p)^{n_i(|\pi_A|+1)}\right]^k \times \left[1-(1-p)^{n_i(|\pi_A|+1)}\right]^{|\mathcal{C}|-|\pi_A|-k-1}.$$

$Prob(1|\pi_A)$ and $Prob(2|\pi_A)$ are probabilities conditioned on the probability, $P(\pi_A)$, of having $|\pi_A|$ hosts that do not have A in their $local_{suspect}$ list at the beginning of cycle r. The probability $P(\pi_A)$ can be expressed as:

$$P(\pi_A) = \sum_{|\mathcal{C}|-|\pi_A|=2}^{|\mathcal{C}|} \binom{|\mathcal{C}|}{|\mathcal{C}| - |\pi_A|} (1-p)^{n_i(|\mathcal{C}|-|\pi_A|)} \left[1-(1-p)^{n_i}\right]^{|\pi_A|}.$$

As a result the disagreement probability can be estimated as follows:

$$Prob(disagree_A) \approx \left[Prob(1|\pi_A) + Prob(2|\pi_A)\right] P(\pi_A).$$

The probability to have a disagreement ($p_{disagree}$) is the probability to disagree about at least one alive host:

$$p_{disagree} = \sum_{k=1}^{|\mathcal{C}|} \binom{|\mathcal{C}|}{k} \left[Prob(disagree_A)\right]^k \left[1 - Prob(disagree_A)\right]^{|\mathcal{C}|-k},$$

where $p_{agree} = 1 - p_{disagree}$. □

Lemma 19. *The probability $p_{accurate}$ of a host excluded in ViewSnoop to have actually failed can be approximated by* $1 - \ [Prob(1|\pi_B) + Prob(2|\pi_B) + Prob(3|\pi_B)]\ P(\pi_B)$*, where:*

$$Prob(1|\pi_B) = (1-p)^{n_i \times \pi_B},$$

$$Prob(2|\pi_B) = \sum_{r=1}^{|\mathcal{C}|-|\pi_B|-1} \binom{|\mathcal{C}|-|\pi_B|-1}{r} \left[(1-p)^{(|\pi_B|+1)n_i}\right]^r \times \left[1-(1-p)^{(|\pi_B|+1)n_i}\right]^{|C|-|\pi_B|-r-1},$$

$$Prob(3|\pi_B) = \sum_{s=1}^{|\pi_B|} \binom{|\pi_B|}{s} \left[(1-p)^{(|\pi_B|)n_i}\right]^s \times \left[1-(1-p)^{|\pi_B|n_i}\right]^{|\pi_B|-s},$$

$$P(\pi_B) = \sum_{|\mathcal{C}|-|\pi_B|=2}^{|\mathcal{C}|} \binom{|\mathcal{C}|}{|\mathcal{C}|-|\pi_B|} (1-p)^{n_i(|\mathcal{C}|-|\pi_B|)} \left[1-(1-p)^{n_i}\right]^{|\pi_B|},$$

such π_B is the set of hosts that heard from some host B in cycle $r-1$ and $\mathcal{C}$ is the set of all alive hosts in cycle r (assuming no exclusions at all hosts).

Proof. Let E be the event that some correct host B is excluded, then $Prob(E) = 1 - p_{accurate}$.

We define the following:

1. π_B: the set of hosts which do not have B in their $local_{suspect}$ list at the beginning of cycle r.
2. $\pi_{\bar{B}}$: the set of hosts which have B in their $local_{suspect}$ list at the beginning of cycle r.
3. The set of all alive hosts in cycle r (assuming no host has excluded any other host), $\mathcal{C} = \pi_B + \pi_{\bar{B}} + \{B\}$.

A correct host B gets declared as crashed and thus is excluded by some host if any of the following happens:

1. B does not hear any message from all hosts in π_B. This event occurs with probability:

$$Prob(1|\pi_B) = (1-p)^{n_i \times \pi_B}..$$

2. At least one host in $\pi_{\bar{B}}$ does not hear any message from all hosts in $B \bigcup \pi_B$. This event occurs with probability:

$$Prob(2|\pi_B) = \sum_{r=1}^{|\mathcal{C}|-|\pi_B|-1} \binom{|\mathcal{C}|-|\pi_B|-1}{r} \left[(1-p)^{(|\pi_B|+1)n_i}\right]^r \times \left[1-(1-p)^{(|\pi_B|+1)n_i}\right]^{|C|-|\pi_B|-r-1}.$$

3. At least one host in π_B does not hear any message from all hosts in $B \bigcup \pi_B$. This event occurs with probability:

$$Prob(3|\pi_B) = \sum_{s=1}^{|\pi_B|} \binom{|\pi_B|}{s} \left[(1-p)^{(|\pi_B|)n_i}\right]^s \times \left[1-(1-p)^{|\pi_B|n_i}\right]^{|\pi_B|-s}.$$

$Prob(E|\pi_B)$, the probability that some correct host B is declared crashed given that π_B hosts heard from B, can be approximated by

$$Prob(1|\pi_B) + Prob(2|\pi_B) + Prob(3|\pi_B).$$

We have:

$$Prob(E \cap \pi_B) = Prob(E|\pi_B) * Prob(\pi_B),$$

where $Prob(\pi_B)$ is the probability that exactly $|\pi_B|$ have host B in $local_{suspect}$ list at the beginning of cycle r. Thus,

$$\begin{aligned} Prob(E) &= \sum_{|\mathcal{C}|-|\pi_B|=2}^{|\mathcal{C}|} \binom{|\mathcal{C}|}{|\mathcal{C}|-|\pi_B|} Prob(E \cap \pi_B) \\ &= \sum_{|\mathcal{C}|-|\pi_B|=2}^{|\mathcal{C}|} \binom{|\mathcal{C}|}{|\mathcal{C}|-|\pi_B|} Prob(E|\pi_B) \times Prob(\pi_B) \\ &= \sum_{|\mathcal{C}|-|\pi_B|=2}^{|\mathcal{C}|} \binom{|\mathcal{C}|}{|\mathcal{C}|-|\pi_B|} (1-p)^{n_i(|\mathcal{C}|-|\pi_B|)} \left[1-(1-p)^{n_i}\right]^{|\pi_B|} \times Prob(E|\pi_B). \end{aligned}$$

□

6.5 Analytic Evaluation of *ViewSnoop*

Probabilistic performance metrics, like p_{agree} and $p_{accurate}$, cannot be evaluated accurately via experimentation and are best measured analytically. We hence conduct extensive theoretical analyses and simulations addressing: (a) *ViewSnoop*' dependability (b) the effect of network load on the dependability of *ViewSnoop* and (c) *ViewSnoop*'s capability of differentiating between network and host failures.

We address the above points of *ViewSnoop*'s performance and compare with membership schemes based on classic heartbeats. First, we describe the different classic heartbeat-based schemes with which we compare and compute their probabilistic guarantees.

6.5.1 Classic heartbeat-based Memberships

Simple fault-exclusion mechanism (SFTM)

This mechanism is employed as the basis of most existing membership protocols with real-time guarantees [12, 35, 154, 155]. These existing protocols typically augment SFTM with additional help not supported in the context of this chapter, such as allowing hosts to be killed and using additional hardware (see Section 6.8). In SFTM, every host broadcasts a heartbeat in every cycle and at the end of a cycle, suspects and excludes all hosts from which it did not hear. A heartbeat here represents an "I am alive" message. Let us denote by p_{agree}(SFTM) and $p_{accurate}$(SFTM), the probability of installing the same view by all alive hosts ($\mathcal{C}$) in some cycle and the probability of a host declared as failed to have actually crashed respectively, given that a message loss probability is $1 - p$.

Then

$$p_{agree}(\text{SFTM}) = p^{|\mathcal{C}|(|\mathcal{C}|-1)},$$

and

$$p_{accurate}(\text{SFTM}) = 1 - \sum_{r=1}^{|\mathcal{C}|-1} \binom{|\mathcal{C}|-1}{r} (1-p)^r p^{|\mathcal{C}|-1-r}.$$

M-SFTM

A variant of SFTM where every host broadcasts n_i heartbeats per cycle, as opposed sending a single heartbeat. Sending more heartbeats makes M-SFTM more robust than SFTM to message losses in ways supported by the DCS context discussed in this chapter. At the end of a cycle, every host in M-SFTM suspects and excludes all hosts from which it did not hear any heartbeat. A heartbeat has the same structure as in SFTM. Alive hosts in M-SFTM install the same views if each host hears some heartbeat from every other alive host. Hence,

$$p_{agree}(\text{M-SFTM}) = \left[1 - (1-p)^{n_i}\right]^{|\mathcal{C}|(|\mathcal{C}|-1)}.$$

A host h is falsely excluded by at least one other alive host if at least one other host receives none of host h's heartbeats. Thus:

$$1 - p_{accurate}(\text{M-SFTM}) = \sum_{k=1}^{|\mathcal{C}|-1} \binom{|\mathcal{C}|-1}{k} \left[(1-p)^{n_i}\right]^k \left[(1-(1-p)^{n_i})\right]^{|\mathcal{C}|-1-k}.$$

Ring Algorithm

A variant of SFTM, this time not using all-to-all communication, inspired from Larrea et al. [131–133]. Hosts send heartbeats following a ring structure. Initially host 1 sends heartbeats only to host 2, host 2 to host 3, ..., and host n to host 1 (n : total number of hosts). Similar to Section 6.4, we describe the algorithm for $s_c = 3$.

The ring algorithm uses a boolean variable, "$suspect$", initially set to false.

In every cycle, host$_i$ sends a heartbeat tagged with the cycle number, n_i times to host$_{i+1}$ following it on the ring of the current system view. The heartbeat has the same structure as in SFTM. At the end of the control cycle, every host$_i$ checks if it received some heartbeat from host$_{i-1}$ of the current system view. If no such message is received, then host$_i$ updates its $suspect$ variable to true (false otherwise).

At the beginning of cycle r, every host$_i$ executes the code below.

```
Install $V = (r, M)$ such that $M = M'$, where $(r-1, M')$ is the view of the previous
control cycle.
IF ($suspect$) DO
  Broadcast $< id(\text{host}_{i-1}), crash >$ message $n_i$ times.
  Declare host$_{i-1}$ failed & install at the beginning of
          cycle $r+1$, $V = (r+1, M'')$:  host$_{i-1} \notin M''$.
  Set $suspect = false$
  Listen in control cycle $r+1$ to heartbeats of
          host$_{i-1}$ in the new $V = (r+1, M)$.
ELSE
  Install at beginning of cycle $r+1$, $V = (r+1, M)$, $M$
   being the system view at cycle $r$.
ENDIF
At the end of a control cycle $r$, every host$_i$ executes:
IF ($< id(\text{host}_x), crash >$ is received) DO
   Declare host$_x$ failed and install at beginning of cycle $r+1$, $V = (r+1, M'')$ such
that host$_{i-1} \notin M''$.
ENDIF
```

The probability of hosts installing the same view in the ring algorithm boils down to two cases:

1. If every alive host receives at least one heartbeat from the host that precedes it in the ring, i.e., if there are no false suspicions.
2. If all hosts receive the $< id(\text{host}_x), crash >$ of any host, whenever sent.

In the second case, i.e., (2), a host h_i which is falsely suspected agrees to install a view where h_i is not alive in that view. As such, according to the description of the algorithm h_i no longer receives heartbeats from the alive host h_j preceding it. Since h_i is still alive, despite being excluded form the view, h_i eventually suspects h_j and broadcasts the message $< id(\text{h}_j), crash >$. Upon receiving such messages, we face two cases:

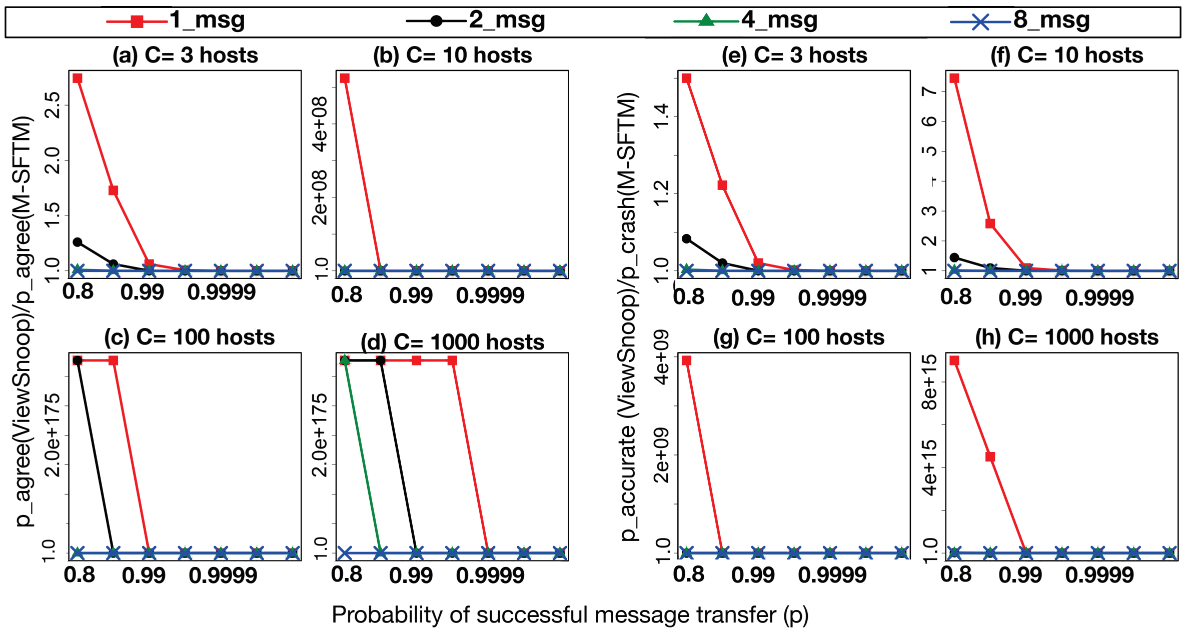

Figure 6.4 – The ratio of the agreement probability and accuracy of ViewSnoop over SFTM and M-SFTM algorithms.

1. Hosts exclude h_j and then h_j leads the host preceding it to get excluded and so on, until all hosts get eventually excluded. This case means that the accuracy of the algorithm drops to zero, as a correct host gets excluded for sure.

2. Hosts ignore messages received from the excluded host h_j. In this case, h_j becomes in disagreement with the rest of hosts in the system.

To this end, the probability of agreement among hosts in the ring algorithm is approximated with the probability that all correct hosts do not get falsely suspected. In other words:

$$p_{agree}(\text{Ring}) = [1 \;-\; (1-p)^{n_i}]^{|\mathcal{C}|}\,.$$

The probability that a correct host is falsely suspected and excluded in a cycle, i.e., the probability that a host is placed in the suspected list of at least one other host in some cycle is:

$$1-p_{accurate}(\text{Ring}) = (1-p)^{n_i} + (1-p)^{n_i}\sum_{h=1}^{|\mathcal{C}|-1}\binom{|\mathcal{C}|-1}{h}\,[(1-p)^{n_i}]^h\,[1-(1-p)^{n_i}]^{|\mathcal{C}|-1-h}\,.$$

Now we evaluate the dependability of *ViewSnoop* compared to the aforementioned classic heartbeat-based schemes.

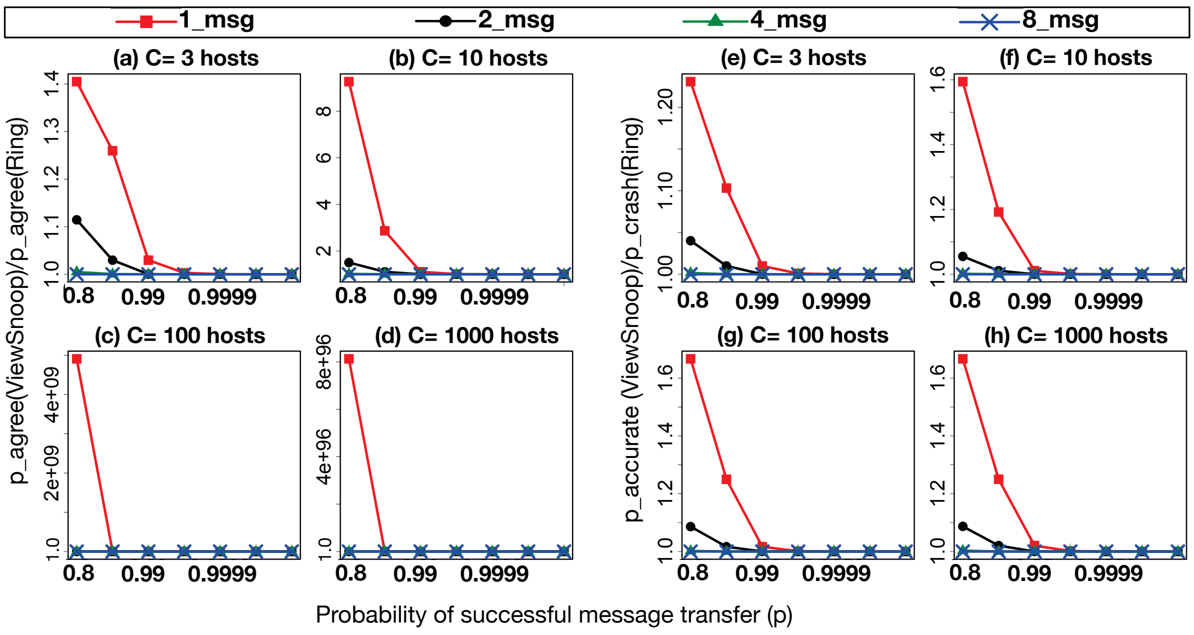

Figure 6.5 – The ratio of the agreement probability and accuracy of ViewSnoop over the ring algorithm.

6.5.2 *ViewSnoop*'s Dependability

We evaluate *ViewSnoop*'s dependability by comparing the values of p_{agree} and $p_{accurate}$ achieved by *ViewSnoop* versus those obtained using SFTM, M-SFTM and the ring algorithms. Any gain in p_{agree} and/or $p_{accurate}$ translates into a more dependable SYMS implementation and a better DCS availability[5]. We simulate the values of p_{agree} and $p_{accurate}$ for all algorithms using $n_i \in \{1, 2, 4, 8\}$ broadcast messages per control cycle. Note that for $n_i = 1$, SFTM and M-SFTM become the same algorithm.

Simulations are run for values of $p \in \{0.8,\ 0.9,\ 0.99,\ 0.999,\ 0.9999,\ 0.99999,\ 0.999999\}$ and $\mathcal{C} \in \{3, 10, 100, 1000\}$, where p is the probability of a host receiving a broadcast message successfully and $\mathcal{C}$ is the number of alive hosts in the system.

Gain in Agreement Probability

Figure 6.4 (a)-(d) and Figure 6.5 (a)-(d) report respectively the ratios of p_{agree} obtained by *ViewSnoop* w.r.t. that obtained by M-SFTM (SFTM) and ring. The actual values of p_{agree} of the different algorithms, i.e., not the ratios, are also reported in Figure 6.7 for $p = \{0.9, 0.99, 0.999\}$.

[5]Lower accuracy means more correct hosts get excluded, increasing the risk of downtime due to the lack of processing resources.

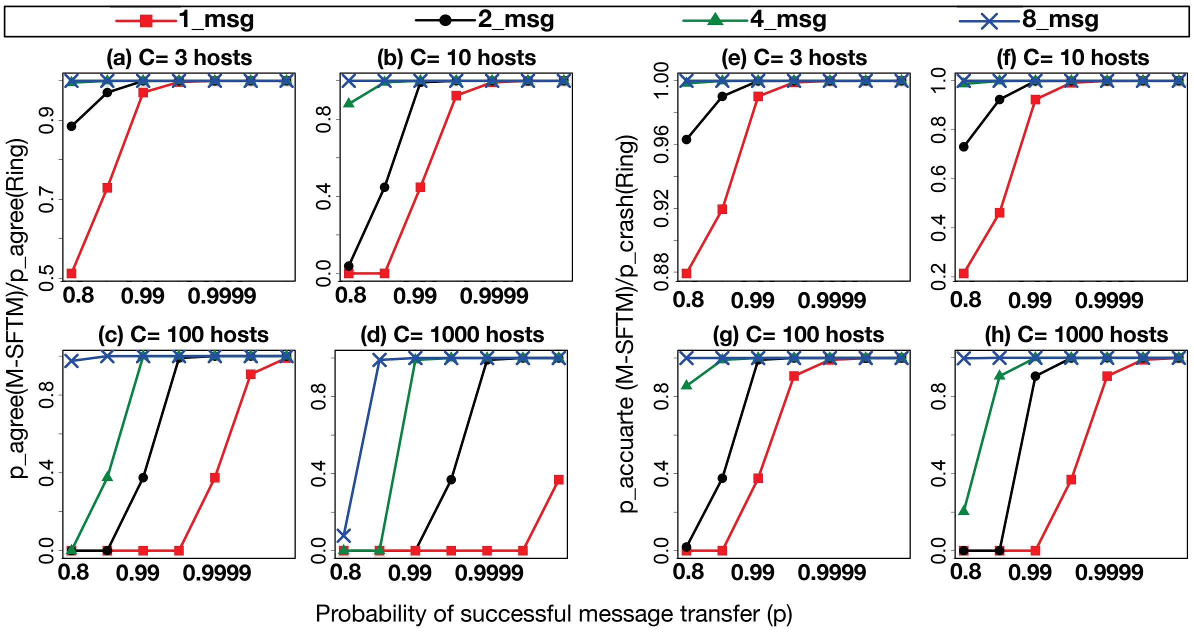

Figure 6.6 – The ratio of the agreement probability and accuracy of SFTM and M-SFTM over the ring algorithm.

The following remarks are in order:

1. *ViewSnoop* has a higher agreement probability compared to all other classic heartbeat-based mechanisms and under all settings, i.e., for all values of p, n_i and $\mathcal{C}$.

2. For a fixed value of p and a fixed n_i, the gain in the agreement probability for *ViewSnoop* over all other classic heartbeat-based mechanisms increases exponentially as the number of alive hosts $\mathcal{C}$ increases.

3. Given a fixed number of alive hosts ($\mathcal{C}$), the positive gain in the agreement probability of *ViewSnoop* compared to all other mechanisms tends asymptotically to zero (i.e., no gain) as $p \to 1$ and as n_i increases (when messages losses are fully masked all algorithms provide the same guarantees).

Gain in Accuracy

The gain in the probabilistic accuracy of *ViewSnoop* over that of M-SFTM (SFTM) and ring can be observed in Figure 6.4 (e)-(h) and Figure 6.5 (e)-(h), respectively. The actual values of $p_{accurate}$ of the different algorithms, i.e., not the ratios, are reported in Figure 6.7 for $p = \{0.9, 0.99, 0.999\}$
.

Algorithm	Message Complexity	Message Content
SFTM	$O(\mathcal{C}^2)$	2 integers
M-SFTM	$O(n_i\mathcal{C}^2)$	2 integers
Ring	$O(n_i\mathcal{C})$	2 integers
ViewSnoop	$O(n_i\mathcal{C}^2)$	2 integers $+1$ string

Table 6.1 – Message complexity of algorithms implementing SYMS.

The probability of falsely excluding an alive process achieved by all classic heartbeat-based mechanisms is lower bounded by the probability achieved by *ViewSnoop*, allowing it to have the best accuracy. This lower bound becomes tighter as $p \to 1$ and $n_i \to \infty$ while it becomes more relaxed as $\mathcal{C} \to \infty$.

Conclusion. *ViewSnoop* indeed offers a more dependable service, compared to SFTM, M-SFTM and the ring algorithm, enhancing thus the availability of a DCS. The significance of this improvement relies on: (i) the number of heartbeats sent every cycle, (ii) the reliability of the communication and (ii) the size of the DCS. For DCSs suffering from communication losses, our algorithm provides superior probabilistic guarantees for critical cyclic control applications (where the number of sent heartbeats per cycle is scarce) compared to mechanisms based on sending simple heartbeats. In fact, we show in Appendix C that both probabilities with which *ViewSnoop* implements SYMS properties, i.e., p_{agree} and $p_{accurate}$, tend to 1 as the number of hosts in the system tends to ∞.

6.5.3 The Effect of Network Load

We study the effect of network load on the guarantees of algorithms implementing SYMS. Network load can be split into: (i) message complexity, i.e., the number of heartbeat messages sent per host per cycle, and (ii) message size. We study the impact of these factors individually.

Message complexity

Table 6.1 summarizes the message complexitiesin the absence of host crashes and losses.

Observation 1. *Allowing hosts to send more heartbeats per cycle improves an algorithm's probabilistic guarantees.*

Our results in Figure 6.7 show that both, p_{agree} and $p_{accurate}$, for all algorithms, i.e., *ViewSnoop*, M-SFTM and ring, increase as n_i (the number of heartbeats sent per host per cycle) increases. This is expected as sending the same message multiple times helps mask potential losses of that message.

Observation 2. *Consider algorithms A and B with message complexities $O(A)$ and $O(B)$ respectively. If $O(A) \geq O(B)$ then the statement: "B is at most as good as A", w.r.t. the ensured reliability and availability,* ***does not*** *hold.*

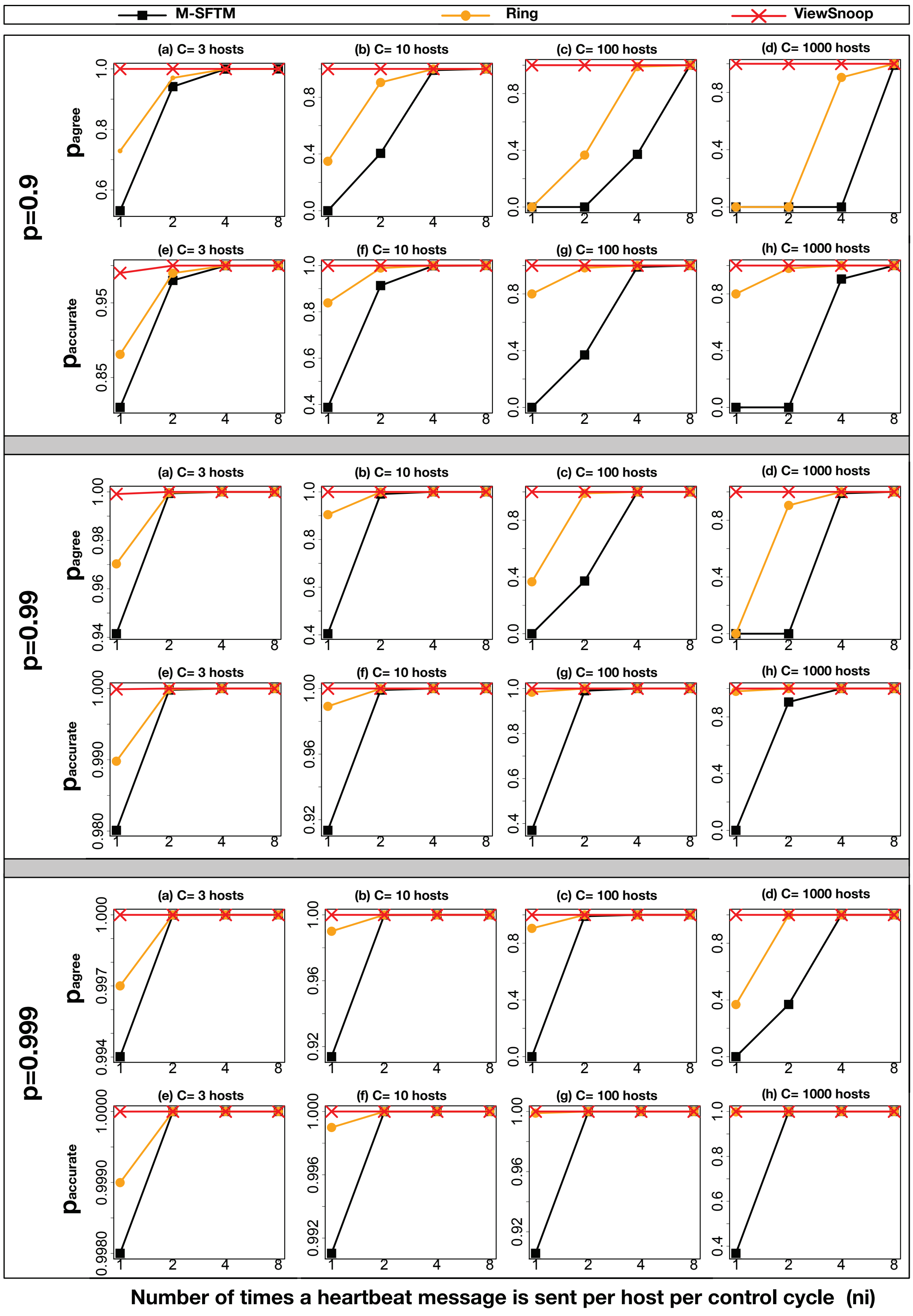

Figure 6.7 – Values of p_{agree} and $p_{accurate}$ of all algorithms versus varying values of n_i for $p = 0.9$, $p = 0.99$ and $p = 0.999$.

Our results in Figure 6.7 show that the ring algorithm (with lower message complexity) provides better guarantees than M-SFTM, both in terms of p_{agree} and $p_{accurate}$. This result also holds for various values of p (see also Figure 6.6).

Conclusion. Increasing the number of heartbeats sent per host per cycle of an algorithm implementing SYMS increases the probabilistic guarantees of that algorithm; however this relation does not hold across algorithms. In other words, the performance of distinct algorithms cannot be compared solely based on their message complexity.

Message Size and Structure.

We investigate whether *ViewSnoop* benefits from sending more information in a heartbeat than simply: "I am alive", to achieve better guarantees. We compare M-SFTM and *ViewSnoop*, as they have the same message complexity but differ in message size. Such a comparison shows the impact of exchanging local views versus simple heartbeats. Figure 6.7 shows a positive improvement in p_{agree} and in $p_{accurate}$ when local views are exchanged. The improvement over M-SFTM, for 10 hosts, is about $9.2\times$, increasing exponentially with the increasing system size.

Conclusion. Appending local views to heartbeats allows *ViewSnoop* to increase its probabilistic guarantees and thus the availability of the DCS. The improvement is most significant in large DCSs running critical applications (small n_i).

6.5.4 Distinguishing Host Crashes from Message Losses

Distinguishing host crashes from message losses is very important in DCSs. In case of message losses where hosts are still alive, a DCS can benefit from this information to update the configurations such that tasks requiring communication would not be allocated to hosts connected by bad links or different routes for communication are used instead. In all classic heartbeat-based mechanisms, roughly speaking, a host A knows the state of host B (if B is alive or not) only if A and B can communicate, even with the use of sequence numbers [165]. As long as communication between A and B is down then A has no idea about B's state.

In contrast, *ViewSnoop*, by exchanging views, allows A to know about the state of B from other hosts even if communication between A and B is down. *ViewSnoop* can detect the failure of communication, when for example host B is in the suspect list of host A while A still sees that host B is not in the suspect list of all other hosts that A hears from. Let p_{com_fail} be the probability that a host A detects correctly the communication failure between itself and another host. Note that all other classic heartbeat-based mechanisms are incapable of detecting correctly a communication failure without a trade-off in $p_{accurate}$, while *ViewSnoop* can do it with probability:

$$p_{com\_fail} = 1 - (1-p)^{n_i(|\mathcal{C}|-1)} - \sum_{k=1}^{|\mathcal{C}|-1} \binom{|\mathcal{C}|-1}{k} \left[(1-p)^{n_i}\right]^k \left[1-(1-p)^{n_i}\right]^{|\mathcal{C}|-1-k}$$
$$\times \sum_{h=1}^{|\mathcal{C}|-1-k} \binom{|\mathcal{C}|-1}{h} \left[(1-p)^{n_i}\right]^h \left[1-(1-p)^{n_i}\right]^{|\mathcal{C}|-1-k-h}$$

Notice that p_{com_fail} tends to 1 as n_i and $|\mathcal{C}|$ increase. This means that *ViewSnoop* can detect communication failures with high probability in large systems where hosts are not limited in the number of heartbeats they can send per control cycle.

6.6 Run-time, Application and Implementation Details

6.6.1 Run-time Environment

We implemented *ViewSnoop* and deployed it in FASA [12], an industrial automated DCS framework, whose behavior adheres to the description in Section 7.2.

A cycle in FASA consists of a phase called the “execution period”, followed by a phase called the “slack period” (see Figure 6.8 (a)). The execution period is the time a host utilizes for executing tasks assigned to it. The slack period is the remaining time of the cycle (used for running background operations if any is needed).

The scheduler in FASA computes global configurations statically and installs the configurations relative to the alive hosts. The scheduler, based on the configuration, knows which hosts need to communicate with each other. Accordingly, the FASA scheduler builds abstract communication *channels* between communicating hosts on top of the unidirectional links. These channels can be configured to use different underlying links.

For our experiments, we deploy *ViewSnoop* in a FASA system where the scheduler can accommodate a maximum of one failure. The scheduler embodies pre-computed configurations to re-distribute application tasks on hosts, when only a single host can crash. The failure of more than one host would cause the system to stop executing.

6.6.2 Application

We execute on FASA a cyclic control application called *Waveform* (see Figure 6.8 (b)). *Waveform* is a simplified example of an industrial control application (extracted from a real setting) that reads some input variable, performs calculations (e.g., a cascaded feedback loop), and finally writes some output to a field-bus I/O interface.

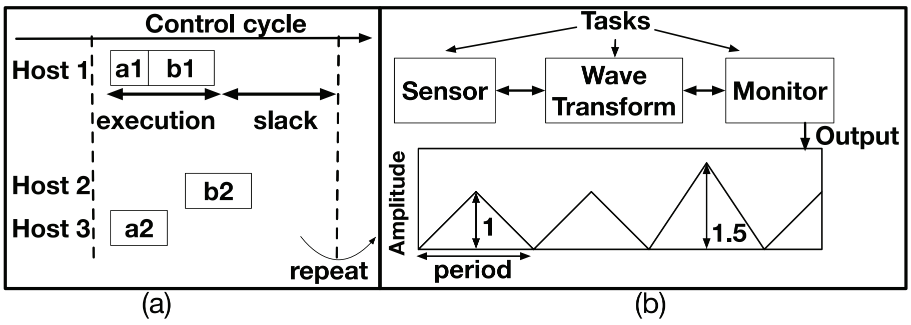

Figure 6.8 – (a) A FASA control cycle; (b) *Waveform*: an example application.

The application is executed every cycle, by periodically executing the application's tasks. In the example, a new input value is provided by the *Sensor* task at the beginning of each cycle. The input follows a triangular periodical signal. The *WaveTransform* task performs certain calculations that change the input signal. Specifically, the *WaveTransform* task in this example observes the input signal to learn its amplitude, base, and period and increases the upper half of the triangular wave amplitude by a factor of 1.5 every third period. This output is fed into a *Monitor* task, which prepares the value for output to a field-bus I/O interface.

6.6.3 ViewSnoop Implementation

ViewSnoop is implemented within the FASA distributed scheduler (see Figure 6.9) in C++. For communication, *ViewSnoop* has access to a UDP broadcast primitive (without acknowledgments and prone to communication failures).

At the beginning of every cycle, the *ViewSnoop* module on every host broadcasts a heartbeat message. In our current implementation, *ViewSnoop* sends a single heartbeat per cycle, i.e., $n_i = 1$. This scheme could be extended to multiple broadcast messages, however, taking into account the cycle time (in Section 6.5 we evaluate *ViewSnoop* with $n_i \geq 1$). The *ViewSnoop* module on each host maintains a local list of suspected hosts (see Figure 6.9). This list is implemented as a vector containing the $host\_ids$ of suspected hosts. The heartbeat message in *ViewSnoop* is implemented as an object encapsulating the control cycle number and the list of suspected hosts.

For programming simplicity, this object is parsed into a string when transmitted on the network[6]. During the slack period, the *ViewSnoop* module on h_i checks for the *current_host_ids*: the ids of the hosts from which h_i received heartbeat messages for the current cycle. *ViewSnoop*

[6]Different compression schemes can be applied before the bit-stream vector is sent as a string over the network. One such compression is done by transforming the bit-stream to the corresponding ASC_{ii} characters and then sending these characters over the network.

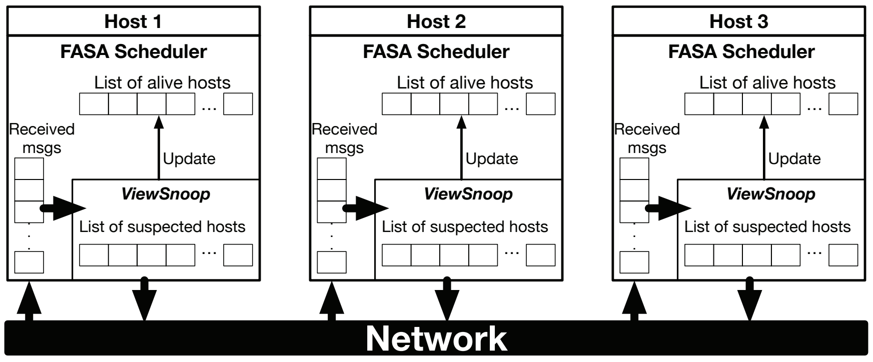

Figure 6.9 – Architecture of *ViewSnoop* within FASA.

decides based on its local list of suspected processes and the *current_host_ids* to update the list of alive hosts observable by the scheduler. Upon observing a change in the list of alive hosts, the scheduler activates the corresponding configuration.

6.7 Experimental Evaluation of *ViewSnoop*

We evaluate experimentally the performance and cost of *ViewSnoop* addressing the following points: (a) the time for detecting and excluding crashed hosts, (b) the time the system remains available in the presence of communication losses, given no host failures (c) the overhead *ViewSnoop* adds to a DCS and (d) the speed at which *ViewSnoop* accommodates host recoveries.

We compare the performance of *ViewSnoop* with that of SFTM, the mechanism employed in most existing membership protocols with real-time guarantees (described Section 6.5). Whenever needed, we inject message losses in the network at the receiver side; we assign a fixed *success probability* p with which a sent message is successfully received by a destination host (unreliable broadcast). This is besides any other message losses that can happen in the network; these losses can be correlated and can result from collisions, contention, etc, since we conduct our evaluation in a real production DCS.

Hardware Description.

We deploy *ViewSnoop* within FASA [12] and precisely in the same industrial setting in which FASA was originally implemented, tested and run [12]. That is, three Mac Minis with dual-core Intel i7-2620M @2,7GHz CPU, 4 GB RAM, and Gigabit Ethernet network connection (a similar implementation setting was also used for example in the RTCAST real-time membership protocol [154]). Our implementation on three machines is used to validate and estimate the overhead of *ViewSnoop* versus that of SFTM (also deployed in FASA [12]). Performance on larger DCSs is rigorously simulated in Section 6.5.

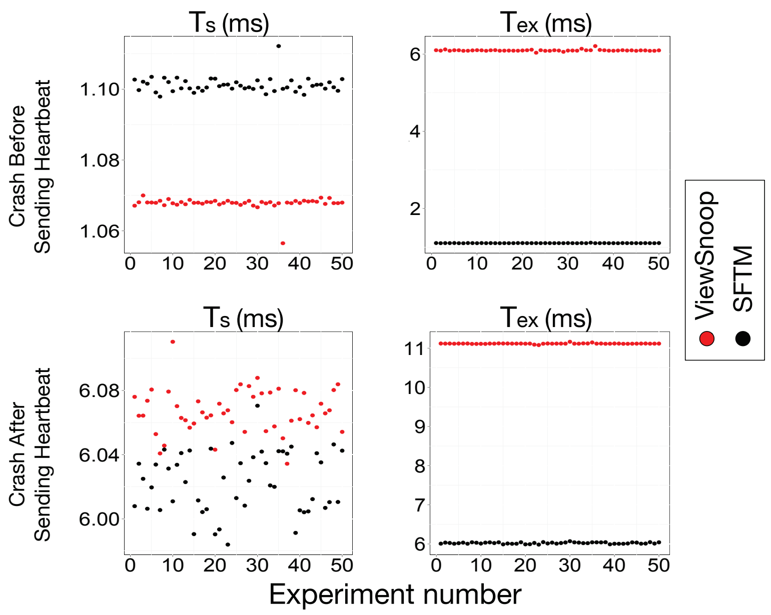

Figure 6.10 – Time to suspect (T_s) and exclude (T_{ex}) crashed hosts.

The machines are given unique ids ($1, 2$ and 3) and we refer to them as hosts. All machines run Ubuntu 12.10, Kernel: $3.5.0 - 24x86\_64$. The hosts use control cycles of 5 ms and are synchronized using PTP [168]. The cycle duration in practice varies according to the applications, e.g., 8 ms to 10 ms for substation automation and low-level robot interfaces [169] and up to 1 s for less critical temperature-drive applications.

6.7.1 Time to Exclude Crashed Hosts

We verify experimentally our theoretical claims (Section 6.4) regarding *ViewSnoop*'s speed of excluding crashed hosts, and compare them to those achieved by SFTM. To this end, we crash host 1 at the 50^{th} cycle in two manners: (i) before sending a heartbeat in the 50^{th} cycle and (ii) after sending that heartbeat. We measure the following:

1. The time span until some host in the system suspects host 1 after it fails (T_s).
2. The time span until host 1 gets excluded after its failure by all hosts in the system (T_{ex}).

Theoretically, $T_s \leq 2$ and $T_{ex} \leq 3$ cycles in *ViewSnoop* (see Section 6.4) and $T_s = T_{ex} \leq 2$ in SFTM (implied from SFTM's description earlier this section).

We repeat the experiment 50 times and report the values in Figure 6.10. Our results show that when host 1 crashes before sending a heartbeat, host 1 gets suspected, in *ViewSnoop*, after $\approx$ 1 ms,

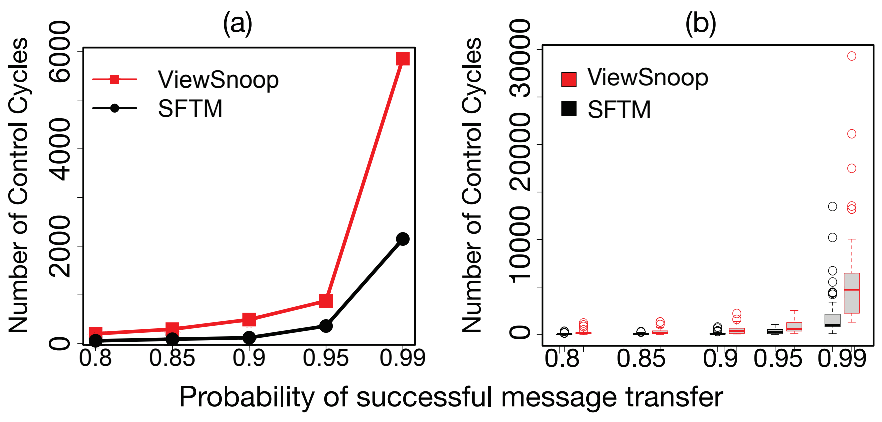

Figure 6.11 – Number of control cycles until an alive host is excluded (the higher the better): (a) mean; (b) distribution.

i.e., at the same cycle it crashed in (the 50^{th} cycle) and gets excluded from the system after 6 ms (at the 51^{st} cycle).

When host 1 crashes after sending a heartbeat, host 1 gets suspected, in *ViewSnoop*, 8 ms after crashing (at the 51^{st} cycle) and excluded from the system 11 ms after crashing (at the 52^{nd} cycle). Our results also show that, in SFTM, $T_s = T_{ex}$ and that host 1 is suspected at the same cycle as in *ViewSnoop* but excluded one cycle earlier than in *ViewSnoop*.

6.7.2 Mean Time to Failure

We assess the system's reliability by measuring the mean time to failure of *ViewSnoop*, focusing here on violating P4 of the group membership properties. To this end, we count the number of control cycles in a crash-free execution until an alive host is excluded by *ViewSnoop* by mistake.

We consider crash-free executions (we do not crash any host) and simulate message losses on the network by specifically having a receiver of a broadcast heartbeat deliver that message randomly with fixed probability $p \in \{0.8, 0.85, 0.9, 0.95, 0.99\}$; the message is dropped otherwise. We do not consider values of $p > 0.99$ in experimentation since it requires weeks or even months to obtain the desired numbers. We account, however, for such values of p in our analytic evaluation (Section 6.5).

For each value of p, we run the system for 50 times, measuring in each how long it takes the system to declare a correct host as failed. We plot the average values and the detailed distribution in Figure 6.11 (a) and (b) respectively. Our results show that *ViewSnoop* can keep a correct host in the system for a much longer time than SFTM. This means that a system with *ViewSnoop* is expected to have a higher availability (processing resources are available for longer times) and reliability (falsely suspects processes at a lower rate) than with SFTM.

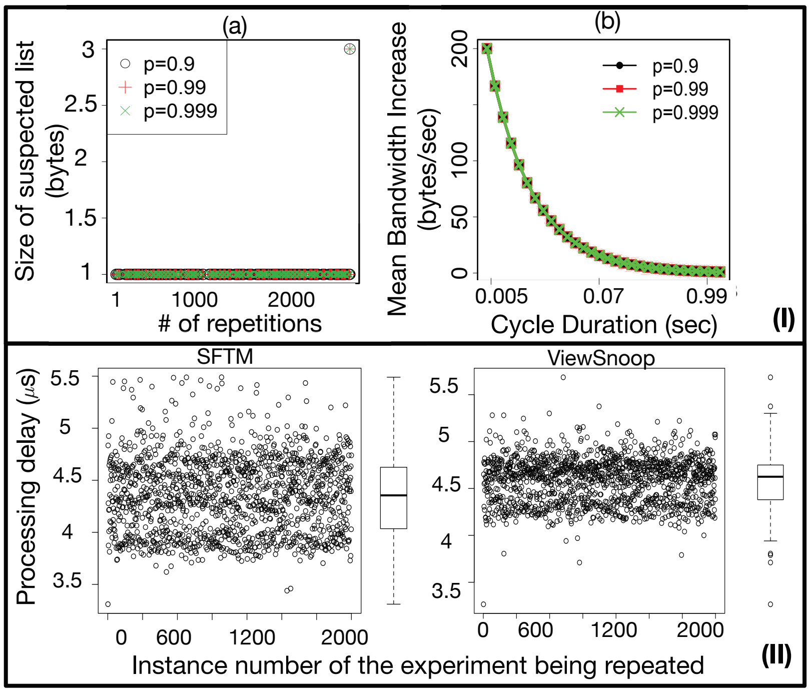

Figure 6.12 – I. Network overhead: (a) 1 out of 3000 heartbeats contained a 3 byte instead of 1 byte suspicion list; (b) *ViewSnoop*'s mean bandwidth increase versus cycle duration; II. Processing delay of SFTM and *ViewSnoop* in μs.

We also consider a variant of SFTM: a host is excluded if no heartbeat is received from that host for two consecutive cycles. This variant is considered to compare *ViewSnoop* and SFTM when having the same speed of excluding crashed hosts.

Even with this variant, *ViewSnoop* amounts to a better reliability than SFTM. The probability of excluding a correct host in *ViewSnoop* is $[(1-p)^2(1-p^2)]$ versus $(1-p)^2$ in SFTM. Correct hosts are thus expected to stay included in the system using *ViewSnoop* for a longer duration. The reason is that *ViewSnoop* allows hearing about hosts from other alive ones.

6.7.3 Costs of *ViewSnoop* on a DCS

Network cost

We quantify the network overhead of *ViewSnoop* by measuring the additional bandwidth required by our *ViewSnoop* implementation in comparison to that needed by SFTM. *ViewSnoop* requires every host to append to the heartbeat of SFTM, a string containing the ids of the suspected hosts. Such a heartbeat is sent every control cycle, in this case every 5 ms.

As in Section 6.7.2, we introduce message losses; messages are successfully received with probability $p \in \{0.9, 0.99, 0.999\}$. We run the system until it halts and record the size (in bytes) of all suspected lists appended to heartbeats in that run by all hosts. For each value of p, we repeat this experiment ≈ 20 times ($\approx$3,000 heartbeats). We report the values in Figure 6.12-I(a). Our results show that the size of the suspicion lists is consistent in all repetitions: 1 out of 3000 heartbeats contained a 3 byte instead of 1 byte suspicion list. 3 bytes lists are observed at the end of the experiments, since we run the system until one host is falsely detected as failed, after which all hosts stop operation. This causes all hosts to suspect each other increasing the size of the list. These values do not vary between the different values of p.

An Ethernet packet in SFTM is 64 bytes (IPv4) and 84 bytes (IPv6), meaning that our algorithm induces, on average, an overhead of 1.6% and 1.3% respectively, compared to SFTM. We also plot, in Figure 6.12-I(b) the average additional bandwidth of *ViewSnoop* (compared to SFTM) as a function of the cycle duration (varying from 5 ms to 1 s). The additional bandwidth for *ViewSnoop* is 200 bytes/sec for 5 ms cycles and decreases exponentially as the cycle duration increases. It is important to note though, that each frame consists of Ethernet, IP and UDP headers (18, 20 IPv4 (40 IPv6) and 8 bytes respectively), thus allowing 18 bytes of UDP payload. This payload, which already exists in mechanisms using classic heartbeats (as heartbeat messages are sent in any case), can be used by *ViewSnoop* without incurring any bandwidth increase for systems with less than $\approx$140 hosts.

Processing cost

Under the same experimental set-up as for evaluating network costs, we measure *ViewSnoop*'s processing cost, i.e., how much delay does *ViewSnoop* add to the regular processing time of hosts. To that end, we measure during a control cycle the time a host spends to check for crashed hosts, suspect hosts, update the list of suspected hosts and update the content of the heartbeat message to be sent. The statistics are consistent for the different values of p, so for brevity we report statistics for $p = 0.99$.

Figure 6.12-II shows that, on average, *ViewSnoop* requires 0.3 μs more processing time per cycle compared to SFTM. More importantly, such processing is done in the slack period of a cycle, which typically is 20% of the cycle duration, i.e., 1 ms for a 5 ms cycle. Our algorithm thus does not delay any application as it consumes on average 0.46% of the slack period, which is entirely dedicated for background operations by design. We also observe that processing delays above 5 μs occur fewer times, attributed to cases when hosts are falsely suspected.

6.7.4 Host Recoveries

For simplicity, we have discussed in the main paper SYMS and *ViewSnoop*, our implementation of SYMS, without considering host recoveries. In fact, both SYMS and *ViewSnoop* can be easily extended to encompass recoveries. In this recovery setting, the maximum number of hosts in the

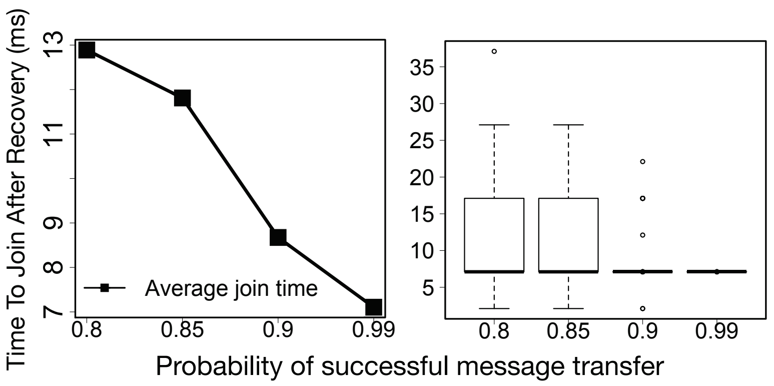

Figure 6.13 – Time for recovered hosts to join the system in *ViewSnoop*.

system is known ahead of time. However, since hosts can be excluded from the system view, due to actual crashes or communication faults, these hosts can try to enter the system again. Let $\mathcal{W}(r)$ be the set of non-crashed hosts at the beginning of cycle r (a non-crashed host does not have to be part of the system view). For coherence, an “alive” host here is a host that has not been declared yet as crashed by any host. A recovered host B becomes “alive” only starting from the cycle in which B’s view includes all alive hosts and the views of all alive hosts include B. To allow recoveries, SYMS 1 is updated to: if a host installs a view $V = (id, M)$ and then $V' = (id' = r, M')$, then $id < r$ and $\{M' - \{M \cap M'\}\} \subseteq \mathcal{W}(r)$. Accordingly, *ViewSnoop* is adapted to account for recoveries as follows:

1. A recovered host B, wanting to join the system, broadcasts heartbeats at the beginning of every cycle. Initially, B only has itself in its view. B learns about the alive hosts in the system and includes them in its view according to (2) below.

2. If a host A receives a heartbeat from host B in cycle r (where B is not in A’s view), then A removes B from its suspected list. A includes B in its view in cycle r' if A does not suspect B and all hosts that A heard from in cycle r' do not suspect B.

We implement this crash recovery extension of *ViewSnoop* and evaluate how long it takes a recovering host to be included in the view of all other alive hosts. A broadcast heartbeat is delivered by a host with probability $p \in \{0.8, 0.85, 0.9, 0.99\}$. We force host 1 to stop sending heartbeats at the 5^{th} cycle and start sending heartbeats again at the 11^{th} cycle. This behavior ensures that host 1 is declared failed by both host 2 and host 3, before trying to join the system back at the 11^{th} cycle. We measure $T_{recover}$, the time from the beginning of the 11th control cycle until host 1 is included in the view of both host 2 and host 3. For each value of p we repeat the experiment 50 times and report our results in Figure 6.12-III.

We notice that as p increases, i.e., less losses, the time for a new host to join the system approaches two control cycles. This duration is an upper bound on the time a process needs to re-join the system in the absence of message losses. We can see for $p \in \{0.8, 0.85, 0.9\}$ cases where a host can re-join in one cycle. This happens when host 2 and host 3 receive host 1's heartbeat at the 11^{th} cycle, but do not hear from each other (such fast re-joins can only happen in unreliable communication).

In comparison, in SFTM host 2 and host 3 include host 1 in their views as soon as they receive a heartbeat from host 1. Host 1, thus, joins the system in the same cycle with probability p^2, after one cycle with probability $(1-p^2)p^2$, and after s cycles with probability $(1-p^2)^{(s-1)}p^2$. In the presence of losses, host 1 joins the system, on average, after

$$\frac{1-p^2}{p^2}$$

cycles. Given no message losses, a recovered host, in SFTM, is recognized by all alive hosts in the system in less than one cycle (one cycle less than *ViewSnoop*).

6.8 Existing Memberships for Real Time

Membership services have been addressed in different contexts [145–147, 166, 170–179]. For example, detecting host crashes in real-time while guaranteeing the desired quality of service needed by applications has been targeted in [14–21]. However, none of these works addressed membership issues, precisely the issue of providing a consistent view of failures.

So in what follows, we focus on existing work on membership services in real-time context.

Kopetz and Grünsteidl [35] proposed the time-triggered protocol (TTP) for distributed real-time control applications. TTP provides many services including membership. TTP assumes time division multiple access (TDMA)[7] as means to organize sender transmission. A node is considered failed when no message is received from that node in its designated transmission slot. Also, a sender node itself can decide if it is not operating correctly, and accordingly crash itself, based on: (i) internal detection mechanisms, (ii) acknowledgments received relative to a window of previous transmissions and (iii) frame rejections (by preforming a specific CRC check). Disagreement is resolved, with high probability, by excluding nodes that do not agree with the majority. Barbosa et al. [155] devised a protocol using TDMA, where each node must acknowledge messages from k other nodes in the membership group. Membership agreement is guaranteed if $f \leq k-1$ failures occur during n consecutive transmission slots (n being the total number of nodes in the system). Rufino et al. [164] proposed failure detection and membership services to enhance the dependability of distributed systems interconnected by field-bus technologies, namely CAN, to levels comparable to TTP-based systems. The major component in their technique is the CAN enhanced layer: a combination of the CAN standard layer with some simple machinery and low-level protocols. Abdelzaher et al. [154] proposed

[7]TDMA divides the medium access into slots such that in each TDMA round nodes transmit a fixed amount of traffic in the preallocated slots.

RTCast, a multicast and membership service for real-time process groups sending periodic or aperiodic messages. *RTCast* relies on a ring topology in which processes take turn in sending heartbeats. *RTCast* relies on processes being able to crash themselves, namely when detecting receive omissions. Amir et al. proposed Totem [157], a reliable total ordered broadcast protocol assuming a ring topology. The Totem protocol depends on acknowledgments and re-transmission mechanisms to overcome communication faults and can provide real-time message delivery with high probability. Clegg and Marzullo [158] designed group membership with low overhead, low cost of handling failures and supporting simple schedulability analysis. The proposed solution abstains from sending heartbeats but rather relies on other layers to exchange enough messages to ensure failure accuracy and liveness of membership agreement.

The work in this chapter differs essentially at two levels: (i) required guarantees and (ii) implementation. On the level of guarantees, we require crashes to be synchronously detected on all alive hosts and within bounded periods from the crash. In contrast with existing work, disagreement, even for few rounds after which agreement maybe achieved, must be avoided. We thus define a probabilistic notion of agreement which we seek to maximize. Implementation wise, our protocol (*ViewSnoop*) relies on the periodic exchange of messages, however not in a TDMA fashion as opposed to [35, 155] enabling *ViewSnoop* to tolerate message collisions. Also, as opposed to [164], we do not assume a specific type of interconnection between hosts (i.e., field-bus), we just consider communication is prone to losses, not relying on additional hardware or acknowledgments. Processes in *ViewSnoop* do not kill themselves, as opposed to [35, 154], where the DCS's computing resources can be easily depleted, jeopardizing the system's availability. To this end, we employ a technique not explored in any of the previous work. Our approach relies on exchanging local views as opposed to exchanging classic heartbeats (as [35, 154, 155, 157]) and having self-crashing capabilities.

6.9 Chapter Summary

This chapter defined the membership properties required in DCSs. In their implementable form, these properties take the form of a probabilistic real-time membership service called SYMS. We proposed *ViewSnoop*, an algorithm based on exchanging local views between hosts, as a way to implement SYMS with high probability guarantees and low overhead.

We first evaluated analytically the performance metrics of *ViewSnoop* comparing with membership services based on classic heartbeats [12, 35, 154, 155]. We showed that:

1. *ViewSnoop* provides better guarantees, on both view agreement among hosts and accuracy, compared to membership services based on classic heartbeats alone, e.g., 9.2x better agreement probability and 1.6x better accuracy probability for a system with 10 hosts. This improvement increases exponentially and un-boundedly with system size.

2. *ViewSnoop* distinguishes host crashes from message losses, without jeopardizing accuracy (which all membership services based on classic heartbeats suffer from). *ViewSnoop*, thus, allows better configurations to be computed, accounting for bad links rather than excluding correct hosts.

We then reported on a full implementation of *ViewSnoop* in an industrial DCS framework, called FASA [12]. We evaluated *ViewSnoop*'s performance experimentally (on FASA), comparing it to a classic heartbeat-based implementation, deployed in most existing DCS frameworks [12, 35, 154, 155].

We showed experimentally that *ViewSnoop* is significantly more dependable than the classic heartbeat-based implementation. More precisely, *ViewSnoop* provides a higher accuracy, ranging from 2.5x up to 4x better than the classic implementation. The higher the accuracy, the fewer correct hosts are excluded. Thus, the risk of downtime, due to the lack of processing resources, becomes smaller. This improvement increases as the system size grows.

We also assessed the trade-offs underlying *ViewSnoop*'s design and implementation:

1. *ViewSnoop* notifies hosts about crashes in real-time, with a lower downtime risk compared to using classic heartbeats. The trade-off is only one extra control cycle to recognize crashes and recoveries. *ViewSnoop*, always excludes crashed hosts from the system in less than three cycles after crashing (real-time). *ViewSnoop* also allows recovering hosts, given no message losses, to join the system in less than two cycles. The classic heartbeat-based implementation requires one cycle less.

   It is crucial to note though, that if this trade-off is eliminated, precisely by allowing classic heartbeat-based memberships for an additional cycle to recognize crashes and recoveries, we show that *ViewSnoop* still provides better accuracy and availability.

2. *ViewSnoop* induces 0.3 μs (7%) processing overhead and 200 bytes/sec (11%) network overhead (for UDP over Ethernet), over the classic heartbeat-based implementation. However, the added delay neither affects FASA, nor the upper layer applications, as *ViewSnoop* fully executes within the idle time of a host, while the network overhead is 1.6% (IPv4) and 1.2% (IPv6) of the packet size being sent in the classic mechanism.

3. *ViewSnoop* periodically broadcasts suspicion lists, rather than broadcasting classic heartbeats. Broadcasting suspicion lists will likely lead to a bandwidth bottleneck in large-scale distributed systems if host crashes and losses are common. Yet, *ViewSnoop* is targeted for DCSs, i.e., environments where crashes and losses typically seldom occur [35, 36].

The main contributions of this chapter can thus be summarized as follows:

1. A specification of the membership requirements for DCSs running cyclic applications. We prove that a deterministic form of these ideal requirements is impossible to implement in a system with both host crashes and message losses. We define SYMS, a probabilistic abstraction of the requirements of DCSs. SYMS can be implemented despite message loss.

2. *ViewSnoop*, an algorithm implementing SYMS with high probability. *ViewSnoop*'s design allows it to distinguish host crashes from message losses, better than using message sequence numbers [165], and without affecting accuracy.

3. An experimental and an analytic evaluation of *ViewSnoop*'s performance showing that *ViewSnoop* provides a significantly more dependable service, enhancing a DCS's availability, compared to methods relying on classic heartbeats [12, 35, 154, 155].

7 Real-Time Distributed Shared-Memory

Abstractions such as distributed shared memory (DSM) have a direct impact on control applications. Precisely, many developers consider DSM significantly easier to program with, compared to message passing. Control application programmers are hence more motivated and programming errors are reduced considerably. Besides vastly simplifying programming applications, providing the DSM abstraction in a message-passing context allows algorithms designed for shared memory in mind to be directly used. In a nutshell, having a DSM abstraction in distributed control systems (DCSs) has many advantages.

This chapter investigates how to build a real-time distributed shared memory abstraction for DCSs. We identify the challenges required to implement such an abstraction. Precisely, we prove that, in the presence of host crashes and message losses, such an abstraction is impossible to implement using traditional algorithms that either (i) assume no knowledge about failures or (ii) are built on top of existing failure detector software blocks.

We propose a way to circumvent this impossibility by devising a white-box approach that uses an algorithm, which we call *TapeWorm*. *TapeWorm* attaches information to the crash monitoring messages of the failure detector component in DCSs and uses these messages as the sole means of communication. We prove that *TapeWorm* indeed implements the distributed shared memory abstraction for DCS applications. We also analyze the performance of *TapeWorm* and we showcase ways of adapting and optimizing our approach to various application needs and workloads.

7.1 Motivation

In multi-core machines, shared memory (provided at the hardware level) constitutes the typical means of communication for hosts (processes) [1, 180, 181]. In message-passing distributed systems, however, hosts communicate by exchanging messages over a network. Hence, shared memory, in such distributed systems, no longer physically exists but rather becomes a distributed communication abstraction built using message exchange and local memories of hosts [1, 58, 59, 180, 181].

7.1.1 Benefits of Distributed Shared Memory in DCSs

A shared memory abstraction constitutes a basic building block for implementing networked storage systems, distributed file systems and distributed key-value stores. These distributed data services are undeniably demanded in distributed control systems (DCSs) [13, 55–57, 182, 183]. For example, power grid DCSs require real-time distributed storage systems (distributed hash tables) to store and retrieve monitoring data for wind and photo-voltaic generation sources [182]; ship-board DCSs require distributed real-time data services to be embedded within the ship [56, 183]; traffic control and agile manufacturing require the presence of distributed fresh data that reflects real-world status [55–57, 183]; real-time execution platforms for DCSs, such as [13], rely on real-time replicated data structures for maintaining system and crash detection information.

In addition to its importance as a building block for various data storage services, shared memory is of immense benefit to application programmers in DCSs; programming with shared memory is considered significantly easier than working with message exchanges [1]. This programming simplicity encourages having more control application programmers and limits programming errors. Also, algorithms designed for shared memory in mind could thus be directly used in a message-passing context that provides the distributed shared memory abstraction. In short, there is a fundamental need to study real-time data abstractions, such as shared memory, in DCSs.

To this end, we investigate in this chapter how to build a shared memory abstraction for DCSs. We first derive the guarantees that need to be provided by *read* and *write* operations accessing shared memory in a DCS context [1, 58, 59, 180, 181]; such guarantees define the respective consistency level. We show (Section 7.2.4) that the needed requirements necessitate the presence of all of the following properties: *real-time termination*, *agreement* and *freshness*. Roughly speaking, real-time termination means that each operation, be it a read or a write, always completes in a bounded known duration. Agreement ensures that read operations, issued within a fixed known time-window to the same shared object, return the same value. Freshness guarantees that any value returned by a read operation is a value written by one of the last completed "c" writes, where c is a fixed known number.

The necessary consistency level, captured by the three aforementioned properties, is, however, challenging to implement when accounting for host crashes and message losses that can happen in a DCS. Control systems typically experience host crash rates of about $10^{-5}/hr$ and link failure rates (causing message loss) in the range of $10^{-5}/hr$ (permanent failures) and $10^{-3}/hr$ (transient failures) [35, 36]. In fact, we prove that the three properties listed above are impossible to achieve using traditional ways for implementing distributed shared storage [55–58], i.e., using only algorithms to which the DCS is oblivious or using a black-box approach where algorithms are built on top of existing software blocks, like failure detectors (details Section 7.3). Yet data storage services for DCSs are imminently needed [55–57, 182, 183].

7.1.2 Approach Overview

We propose a way to circumvent this dilemma by devising a white box approach utilizing existing services that typically run within DCSs [13, 74], namely failure detection. Since hosts can crash, DCSs usually employ failure detectors that provide means for detecting host crashes in real time [14–18, 74]. More specifically, failure detectors output a list of hosts suspected to have crashed. Accordingly, DCSs employ recovery mechanisms requiring to shift application-related tasks to be executed only by those hosts which are not suspected to have crashed. In this sense, the output of suspected hosts (if any exists), even if these hosts have not actually crashed, is no longer visible to applications [74].

Benefiting from this behavior, we propose a solution that guarantees the DCS consistency level (consisting of the three aforementioned properties) as seen by the applications, and not necessarily within the set of all non-crashed hosts. We design *TapeWorm*, an algorithm that attaches itself to the crash monitoring messages (heartbeats) typically exchanged on a periodical basis to detect host crashes in DCSs [13, 35, 74]. *TapeWorm* uses the underlying heartbeats, exchanged between hosts, as the sole means of transporting information. By doing so, *TapeWorm* benefits from the real-time operation of failure detectors in DCSs in the following sense. Crashed hosts, based on the exchanged heartbeats, are always suspected in real time (a deterministic guarantee) [14–18, 74]. The speed of detecting crashed hosts (detection time) should be fast enough to guarantee that applications can recover from potential host crashes and still meet their deadlines. *TapeWorm*, by using heartbeats as a transportation mechanism, can have the leverage of reaching hosts that are not suspected, providing services to such hosts in real time.

We prove that *Tapeworm* indeed implements the required three properties of shared memory among non-suspected hosts. We show that *TapeWorm* can be adapted, if need be, to provide real-time guarantees faster than those of the failure detector it relies on. Precisely, *TapeWorm* can be adapted to allow read operations to return fresher values in the allowable freshness range[1].

We also conduct a mathematical analysis computing the probability distribution on the freshness of the values returned by *TapeWorm* as well as the respective incurred bandwidth cost. Our analysis quantifies *TapeWorm*'s performance in terms of variable system parameters, such as the size of the system and the message loss rate. We also devise an optimization of *TapeWorm* for static workloads, where the time at which operations are invoked is known by all hosts in the system.

7.2 DCSs For Cyclic Control Applications

A distributed control system (DCS) consists of a set of n hosts (or processing units), denoted by $\Pi = \{h_1, h_2, ..., h_n\}$. As in any distributed system, these hosts can *fail (crash)* [1], i.e., stop executing operations. The rate of host failures in control systems is typically around $10^{-5}/hr$ [35, 36]. Hosts are considered to be synchronous, i.e., the delay d_p of performing a

[1]The freshness of writes is guaranteed since writes are typically issued by sensors, i.e., on a periodical basis.

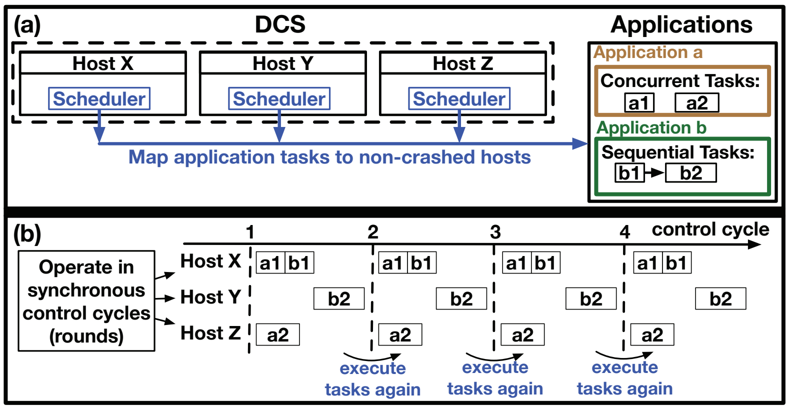

Figure 7.1 – A DCS with three hosts running two control applications.

local step has a fixed known bound. Hosts have access to local synchronized clocks with bounded skew. Using these local clocks, hosts define control cycles (rounds) of the same fixed duration. The cycle duration is $>> d_p$. These control cycles are time-wise synchronized among all hosts, i.e., the start and end of a cycle occur at all hosts at the same time (with a bounded skew). Cycle lengths vary according to applications, e.g., 8-10 ms for substation automation and low-level robot interfaces and up to 1 s for temperature-driven applications [169].

DCSs often execute control applications that are cyclic [13, 148, 149], i.e., run periodically. For example, an application might be required to periodically read values from a sensor and to activate “intensive cooling mechanisms” after t seconds of a machine exceeding a certain temperature threshold. Specifically, cyclic control applications consist of several small *tasks* that repeatedly execute. Some of these tasks run concurrently on several hosts, possibly on behalf of different control applications (see Figure 7.1).

In every cycle, each host executes the tasks assigned to it by the *scheduler*.

7.2.1 Scheduler

The scheduler is a distributed system module that specifies which application tasks run on which hosts and in what order (see Figure 7.1). The allocation of tasks to hosts is called a *configuration*. A scheduler makes sure that all hosts can execute the assigned tasks without exceeding the total cycle duration. Moreover, the scheduler ensures that configurations allow all applications to run correctly and to meet their deadlines. Upon detecting the crash of a host, the scheduler computes (taking into account the crashed host) a new configuration, which maps tasks to hosts. This re-mapping of tasks ensures that applications meet their deadlines despite host crashes.

7.2.2 Communication

Every pair of hosts in a DCS is connected by two logical uni-directional links. Hosts h_i and h_j are connected by links l_{ij} and l_{ji}. Links, in this context, for example abstract a physical bus or a dedicated network link. Arguably, all communication is prone to random disturbances, for example, bad channel quality, interference, collisions, stack overflows etc [184]. Messages can thus be lost. When there is no loss, we assume that messages have a bounded delay, say d. We specifically consider that a message sent on link l_{ij}, $\forall i \neq j$, at any time t has probability $0 < P_{ij}(t) < 1$ of getting lost.

Configurations computed by the scheduler account for the message delay d. As such, any message scheduled to be sent in a control cycle r, if not lost, is assumed to be received in cycle r.

Sending a message reliably from one host to another, however, can take an unbounded amount of time, due to losses and the required follow-up re-transmissions.

7.2.3 Failure Detection and Monitoring in a DCS

A failure detector is a distributed module that runs on every host and provides the DCS scheduler with information about which hosts have crashed [7]. The most common monitoring scheme used by failure detectors in DCSs dictates that each host periodically, i.e., in every cycle, broadcasts a heartbeat message of some structure[2] [13, 35, 74, 154–156]. Based on these heartbeats and using time-outs, a failure detector monitors which hosts in the system have crashed and which have not. As such, we consider, in this chapter, failure detector algorithms using heartbeats and time-outs alone.

Detecting crashes in real time in a DCS is crucial, for instance, in order to allow the scheduler to re-map the tasks (initially assigned to the crashed host) to other hosts, without violating application deadlines. Failure detection varies depending on application needs, but is often expected to be in the order of milliseconds. Since communication is prone to losses, real-time failure detection using heartbeats and time-outs cannot be always *accurate* [15–18, 167, 185]. Accuracy depicts a failure detector's ability of not suspecting correct hosts, i.e., hosts that do not crash. As such, at any cycle r, a host can exist in one of the following sets (assuming host crashes in the fail-stop model [7], i.e., no recoveries):

1. **Crashed hosts $\nabla\mathcal{C}(r)$:** this set includes all hosts that have crashed at some cycle up to cycle r (included).
2. **Eliminated hosts $\nabla\mathcal{E}(r)$:** this set includes all hosts that have not crashed up to and including cycle r but are suspected by the failure detector during cycle r.
3. **Alive hosts $\nabla\mathcal{A}(r)$:** is the set of hosts that have not crashed up to and including cycle r and are not suspected during cycle r.

[2]Different failure detection algorithms may send different information within heartbeats [15–18, 134].

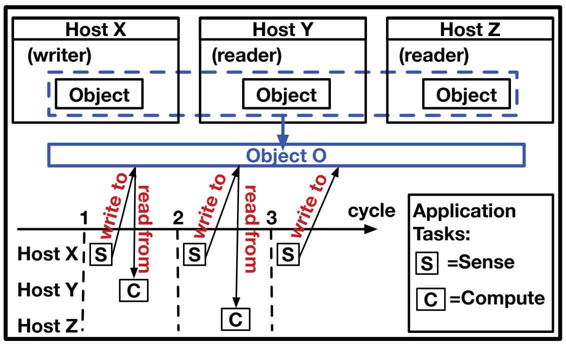

Figure 7.2 – An example of a shared memory (an object) in a DCS.

A scheduler considers all hosts belonging to the set $\nabla\mathcal{A}(r)$ functional at cycle r and computes configurations accordingly. All other hosts, i.e., hosts $\in \{\nabla\mathcal{E}(r) \bigcup \nabla\mathcal{C}(r)\}$, are considered non-functional, and as such no application tasks are allocated to execute on them [13] or their output (if any exists) is made hidden from the applications [74]. Assuming no recoveries, if $r < r'$, then $\nabla\mathcal{C}(r) \subseteq \nabla\mathcal{C}(r')$.

The scheduler, being a distributed module executing on every host (recall Figure 7.1), should maintain a consistent state at all cycles. A consistent scheduler state is achieved when the failure detector, at every host, suspects the same set of hosts, and thus provides the scheduler module at every host with the same set $\nabla\mathcal{A}(r)\ \forall\ r$ [13]. An inconsistent scheduler state means that hosts might be executing different configurations (since hosts have different $\nabla\mathcal{A}(r)$). Different configurations mean different mapping of tasks to hosts. In other words, the DCS would be experiencing "downtime" (normal operation is halted) since applications might be executing incorrectly (communication or the order of execution between tasks of the same application might be invalid) [13].

7.2.4 Shared Storage Abstraction

Applications in distributed control environments require shared memory functionalities [13, 55–57, 182, 183], as application tasks executing on different hosts may require to communicate by reading and writing to shared memory.

Shared memory can be viewed as a collection of shared object abstractions. We consider a DCS, as in [12, 13], where a single object is assigned to every task that writes a value. As such, every shared object in a DCS is a read/write object which can be written to by a single host, however, it can be read from by any number of hosts, i.e., single-writer multiple-reader (SWMR) object [1, 58, 59, 180, 181] (see Figure 7.2). For simplicity, we assume that a shared object is written to, once every cycle[3].

[3]From the reader nodes' perspective, multiple writes to an object in cycle r, can be viewed as a single write, that being the last write in r which is available for reads in cycle $r + 1$.

Properties of a DCS Shared Memory

We now determine the properties that should be satisfied by the read and write operations issued to shared memory (later in this section, we compare with properties of other shared memory abstractions).

Termination. A DCS requires each operation, be it a read or a write, to complete within a bounded amount of time after being invoked, say t_{op}. When computing configurations, the scheduler accounts for t_{op} and assigns tasks to hosts accordingly. Control cycle durations are defined such that operations invoked at some cycle and requiring a delay of t_{op} complete and return in that same cycle. Having operations with unbounded delay makes the scheduler's job, of computing configurations with a fixed cycle duration, impossible.

Read Agreement. Certain critical control applications, e.g., those for power system control, require specific consistency on the values being read by hosts in the same cycle. Namely, read operations invoked in the same cycle to the same shared object, upon completion, are required to return the same value. To better illustrate the need for such a requirement, consider a control application where hosts are required to open or close circuit breakers based on the values read from shared memory. If, in some cycle, two or more hosts read different values of the same shared object, these hosts might allow for undesirable power flows leading for example to creating islands, blackouts, overheating wires, etc [186].

Read Freshness. In a DCS, hosts requiring to read shared memory can typically tolerate some freshness range for the value being read. In other words, it is acceptable if a read operation does not return the latest value written. Such tolerance is typically supported by a DCS due to the presence of failures, message losses, potential unanticipated system delays, etc. More specifically, a read operation invoked by a host in some cycle, upon completion, is required to return a value that is written at most c cycles ago. In practice, the value of c is correlated with the *reaction time*[4] needed by certain applications. Recall, that we assume that every shared object is written to once every cycle.

To sum-up, the consistency of a DCS shared memory is governed by three main properties, formally stated below:

1. **Termination:** Any operation invoked by a non-crashed host completes in the same cycle in which it was invoked. The delay between the time an operation is invoked and the time it completes is at most t_{op}.
2. **Agreement:** Read operations invoked in the same cycle to the same shared object, upon completion, return the same value.

[4]Reaction time is the time interval from the moment some value is written until all hosts can read this value.

3. **Freshness:** A read operation invoked during cycle r, upon completion, returns a value that is written at most c cycles ago, if the writer host of that object is not suspected at r. If the writer, however, is suspected at some cycle r', then all reads invoked in cycles $\geq r'$ return a value written at some cycle in $[r' - c, r']$.

It is important to note that it is sufficient that services in a DCS, such as shared memory, ensure their respective properties at times when the scheduler state in the DCS is consistent, i.e., when the scheduler at all non-crashed hosts sees the same $\nabla\mathcal{A}(r)\ \forall\ r$. Ensuring shared memory properties, in the absence of scheduler consistency is not required, as the DCS would be down (unavailable to deliver correct services).

Comparing with Classic Abstractions

We now compare the aforementioned properties with those of classic shared objects and related abstractions [55, 56, 58, 59, 183, 187].

Atomic Objects. Informally, the atomicity property requires that each operation appears as if it was executed instantaneously at some point in time, regardless of the time taken by each operation to complete [58, 59]. Besides atomicity, atomic objects also require that operations respect their temporal order, basically, having reads return the last written value. The properties of a DCS shared object, however, require operations to complete in real time and do not require reads to return the last value written but rather return a value written within a bounded past duration from the time a read is invoked.

Temporally Consistent Objects. Temporal consistency represents the consistency level adopted in most real-time distributed databases [55, 56, 183]. The temporal consistency property is typically used to quantify the freshness of replicated values among distributed hosts. Two objects are said to be temporally consistent with each other if their corresponding timestamps are within a known fixed bound δ. In this sense, the freshness property defined here ensures temporal consistency among hosts in a DCS. Besides temporal consistency, real-time databases require real-time responses. Real-time response is similar to the termination property defined in this chapter. However, the termination property for DCSs is strictly stronger as it requires operations to complete within a bounded delay at all times rather than with high probability. The main difference is that real-time distributed databases do not require the agreement property defined above, essential in a DCS context.

Real-Time Reliable Broadcast. Since we consider SWMR shared objects, we highlight the difference with closely related abstractions like a reliable broadcast abstraction [187] in real time. Roughly speaking, real-time reliable broadcast requires a sent message to be delivered to all hosts or none in some bounded time δ. One main difference is that the shared memory abstraction we define requires not only to agree about the value to be delivered but on the time of delivery as well. Hosts invoking reads in the same cycle should deliver the same value; otherwise we do not require agreement. This means that messages should be delivered within a certain time bound after being sent, but more importantly messages should be delivered within the exact same cycle at all hosts (requiring to see that message). Another difference is the fact that a real-time reliable broadcast alone cannot ensure the freshness property; it is possible that every message sent is delivered by no one.

Control Applications For Distributed Shared Memory in DCSs

Production DCSs, such as [12, 13], adhere to the architecture and mode of operation that we consider in this chapter. In fact our distributed shared memory abstraction is inspired by the constraints and requirements governing such industrial DCSs. Nowadays, application areas of programmable logic controllers (PLCs), DCSs and SCADA overlap and include monitoring and control applications for factory automation, substation automation and smart grids [188].

7.3 Feasibility of Implementing Shared Memory in a DCS

We discuss, in what follows, the feasibility of implementing a shared object, satisfying termination, agreement and freshness, in a DCS. We first introduce the following lemma, which we rely on in our proofs later in this section.

Lemma 20. *Any algorithm that deterministically implements the termination property implements "local" operations; operations invoked by a host h_i do not wait for responses from any other host h_j $(\forall i \neq j)$ in order to complete.*

Proof. By contradiction assume that an algorithm implements termination and when a correct host h_i (i.e., does not crash) invokes an operation, h_i waits for a reply from at least one other host h_j $(i \neq j)$ in the system. Since cycle durations are fixed, a host can hence send a finite number of messages within a cycle, say m messages.

We compute in what follows the probability that host h_i loses all m messages sent to it by any host h_j in the system. Recall that $P_{ji}(t)$ is the probability with which the link l_{ji} loses a message at time t $\forall j \neq i$. Let $P_{ji}(t \cap t')$ be the probability that l_{ji} loses the messages (if any is sent) at time t and time t'. Since $0 < P_{ji}(t) < 1$ $\forall t$, then

$$0 < P_{ji}(t) = \frac{P_{ji}(t \cap t')}{P_{ji}(t'|t)} < 1 \, \forall \, t', \; t. \tag{7.1}$$

By (7.1), $P_{ji}(t'|t) > 0$ (and $0 < P_{ji}(t \cap t') < 1$). By induction, we have $P_{ji}(t'|t, t_1, ..., t_x) > 0$ $\forall \, t' > \; t, t_x$. Denote by $\mathsf{B}(t)$ the event that l_{ji} losses all messages (if any is sent) for the interval

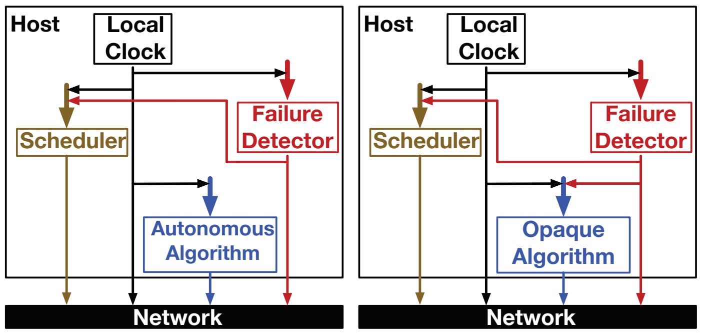

Figure 7.3 – Different algorithmic families resembling traditional approaches for building distributed shared memory.

$t + \Delta t$, where Δt is the control cycle duration. Let t_x denote the times at which h_j sends a message in $[t + \Delta t]$. Then the probability of $\mathsf{B(t)}$ happening is:

$$\begin{aligned} Pr(\mathsf{B(t)}) &= P_{ji}(t_1 \cap t_2 \cap ... \cap t_m) \\ &= P_{ji}(t_1) \times P_{ji}(t_2|t_1) \times ... \times P_{ji}(t_m|t_1, ..., t_{m-1}) > 0. \end{aligned}$$

Given $0 < P_{ji}(t) < 1\, \forall t$ and $P_{ji}(t'|t, t_1, ..., t_x) > 0\, \forall\, t' >\ t, t_x$, then we have $0 < Pr(\mathsf{B(t)}) < 1$; there is a positive probability that h_i receives no reply from h_j and thus the invoked operation does not complete in any bounded cycle duration where a bounded number of messages can be sent by a host, which violates termination. □

We now define and distinguish between different algorithmic families; these algorithmic families resemble traditional approaches for building distributed shared memory [1, 58, 59, 180, 181].

Autonomous Algorithms. These algorithms can only exchange messages using the lossy links of the DCS and use the available local synchronized clocks. In other words, these algorithms do not use any external building blocks (abstractions) but implement themselves all needed functionalities, thus the name autonomous.

The scheduler is oblivious to algorithms in this family; the scheduler does not adapt its behavior to the algorithm's decisions, i.e., changes to the algorithm result in no impact on the scheduler (Figure 7.3).

Opaque Algorithms. These algorithms can use, in addition to message exchange and local clocks, a failure detector as a black-box [7]. Roughly speaking, this means that opaque algorithms can observe the output of failure detectors and build on top of this output and the properties that this output satisfies (Figure 7.3). Different failure detector classes can be defined depending on the properties guaranteed by the output of the failure detector. Recall that we consider that the output of a failure detector is a list of suspected hosts.

The scheduler is not oblivious to the failure detector used, however, both the scheduler and the failure detector are oblivious to the algorithms in this family. It is important to note that algorithms in this family are not a subset of autonomous algorithms, since implementing the failure detector itself may not be possible via message exchange and local clocks.

Theorem 19. *No autonomous algorithm can deterministically implement termination and freshness in a DCS, where up to n-2 hosts can fail.*

Proof. By contradiction, assume the existence of an autonomous algorithm $\mathcal{A}$ that deterministically implements termination and freshness. $\mathcal{A}$ cannot know which hosts are seen as "non-suspected" by the scheduler and thus should guarantee freshness for any non-crashed host.

For illustration, we consider a DCS of two correct hosts h_1 and h_2. By Lemma 20, operations (reads and writes) satisfying termination in $\mathcal{A}$ complete without waiting for any reply from any host. Hosts in $\mathcal{A}$, however can exchange messages to ensure freshness. Consider host h_1 to be the writer of a shared object $\mathcal{O}$, i.e., h_1 invokes write operations to $\mathcal{O}$ at every cycle.

Similar to the reasoning in the proof of Lemma 20, we compute the probability that all messages sent from and to h_1 are lost for $\alpha \cdot c$ cycles $\forall\ \alpha \geq 1$.

Let $P_{21}(t \cap t')$ ($P_{12}(t \cap t')$) be the probability that l_{21} (l_{12}) loses the messages (if any is sent) at time t and time t'.
$0 < P_{21}(t), P_{12}(t) < 1\ \forall t$, then:

$$0 < P_{21}(t) = \frac{P_{21}(t \cap t')}{P_{21}(t'|t)}, P_{12}(t) = \frac{P_{12}(t \cap t')}{P_{12}(t'|t)} < 1\ \forall\, t',\ t. \tag{7.2}$$

By (7.2), $P_{21}(t'|t) > 0$ ($0 < P_{21}(t \cap t') < 1$) and $P_{12}(t'|t) > 0$ ($0 < P_{12}(t \cap t') < 1$). By induction, we have $P_{21}(t'|t, t_1, ..., t_x) > 0$ and $P_{12}(t'|t, t_1, ..., t_x) > 0\ \forall\, t' >\ t, t_x$. Denote by $\mathsf{R}(t)$ ($\mathsf{S}(t)$) the event that l_{21} (l_{12}) loses all messages (if any is sent) for the interval $t + \Delta t$, where Δt is such that $t + \Delta t$ is equal to the duration of $\alpha \cdot c$ cycles. Let m (m') be the maximum number of messages h_1 (h_2) can send in $t + \Delta t$ and t_x (t'_x) denote the times at which a message is sent from (to) h_1 during $[t, t + \Delta t]$. Then the probabilities of $\mathsf{R(t)}$ and $\mathsf{S(t)}$ happening is:

$$\begin{aligned} Pr(\mathsf{R(t)}) &= P_{21}(t_1 \cap t_2 \cap t_3 \cap ... \cap t_m) \\ &= P_{21}(t_1) \times P_{21}(t_2|t_1) \times ... \times P_{21}(t_m|t_1, ..., t_{m-1}) > 0. \end{aligned}$$

$$\begin{aligned} Pr(\mathsf{S(t)}) &= P_{12}(t'_1 \cap t'_2 \cap ... \cap t'_m) \\ &= P_{12}(t'_1) \times P_{12}(t'_2|t'_1) \times ... \times P_{12}(t'_m|t'_1, ..., t'_{m-1}) > 0. \end{aligned}$$

Given $0 < P_{21}(t),\ P_{12}(t) < 1$, $P_{21}(t'|t, t_1, ..., t_x) > 0$ and $P_{12}(t'|t, t_1, ..., t_x) > 0\ \forall\, t' >\ t, t_x$, then $0 <\ Pr(\mathsf{S(t)}), Pr(\mathsf{R(t)}) < 1$.

Consider some cycle r. The probability that all messages from and to h_1 get lost and thus all writes invoked by h_1 during cycles $[r, r + \alpha \cdot c]$ are not seen by h_2 is:

$$Pr(\mathsf{S(t)} \cap \mathsf{R(t)}) = Pr(\mathsf{S(t)}) \cdot Pr(\mathsf{R(t)}|\mathsf{S(t)}).$$

Since $0 < Pr(\mathsf{S(t)}) < 1$, then $0 < \frac{Pr(\mathsf{S(t)} \cap \mathsf{R(t)})}{Pr(\mathsf{R(t)}|\mathsf{S(t)})} < 1$ and $0 < Pr(\mathsf{S(t)} \cap \mathsf{R(t)}) < 1$.

Thus, h_2 invoking a read operation to read the value $\mathcal{O}$, for example at cycle $r + c + 1$, has a positive probability of reading a value that is not written c cycles ago, violating freshness. Recall that, by the termination property, every read operation should complete and return a value within the same cycle in which it was invoked. □

We now define what we call a *non-trivial* failure detector.

Definition 5. *A non-trivial failure detector*[5] *does not suspect a correct host, at any point in time, with positive probability, while it eventually suspects all failed hosts permanently.*

Theorem 20. *Let $\nabla\mathcal{A}(s)$ denote the set of hosts which are not suspected by a non-trivial failure detector $\mathcal{X}$ during cycle s. No opaque algorithm using $\mathcal{X}$ can deterministically implement termination and freshness for hosts $\in \nabla\mathcal{A}(s)$ $\forall s$, in a DCS, where n-2 hosts can fail.*

Proof. An opaque algorithm can observe the output of the failure detector $\mathcal{X}$ and hence can know $\nabla\mathcal{A}(s)$, the set of hosts that are not suspected by $\mathcal{X}$ during cycle s. Consider a host h_i to be the writer of a shared object $\mathcal{O}$, i.e., h_i invokes write operations to $\mathcal{O}$ at every cycle.

Consider an execution where h_i and some other host h_j (which requires to read the value of $\mathcal{O}$ after cycle $r + c$) are correct. Then from Definition 5, there is a positive probability that $\{h_i,\ h_j\} \in \nabla\mathcal{A}(s), \forall s \in [r, r + \alpha c]$ at hosts h_i and h_j.

Similar to the proof of Theorem 19, it can be inferred that there is a positive probability that all writes invoked by h_i during $[r, r + \alpha c]$, are not seen by host h_j (all messages sent by h_i during $[r, r + \alpha c]$ can be lost with positive probability). As such, all reads invoked by h_j after cycle $r + c$ do not return a fresh value. This concludes the proof. □

7.4 TapeWorm: A DCS Shared Memory Algorithm

In this section, we demonstrate a way to circumvent the impossibilities shown in Section 7.3. We present *TapeWorm*, our algorithm for implementing shared memory in DCSs. The main idea underlying *TapeWorm* is to benefit from monitoring messages (known as heartbeats) typically exchanged by the failure detector component of a DCS, precisely by attaching information to these messages. Recall from Section 7.2.3 that we consider failure detectors that detect host

[5] A failure detector providing more accurate information about correct hosts in the system is only a special case of a non-trivial failure detector.

crashes in real time, based only on the exchanged heartbeat messages and time-outs. Accordingly, the below is a necessary condition for any such failure detector that detects crashed hosts in a delay less than d_t cycles after the crash ($\forall d_t$).

Real-time detection: *If a host h_i does not hear any of the heartbeats sent by host h_j during $[r-(d_t-1), r-1]$ (directly from h_j or indirectly via other hosts), h_i suspects h_j at the beginning of cycle r.*

This suspicion places h_j in the eliminated set of hosts $\nabla\mathcal{E}$. d_t is fixed and constitutes the real-time guarantee for detecting a crashed host in the DCS.

7.4.1 A Basic TapeWorm Algorithm

For simplicity, we describe *TapeWorm* (Algorithm 2) in the context of a single shared object $\mathcal{O}$. Recall that d_t is an upper bound on the number of cycles it takes a failed host to be declared as failed by the system (precisely suspected by the failure detector). We assume that $c > d_t$, c being the bound on the data freshness.

Every host in *TapeWorm* maintains a list, $Fresh_{list}$, of size c. $Fresh_{list}$ of a host h at cycle r holds the values of $\mathcal{O}$ seen by h and tagged with control cycles in $[r-c, r]$. Later in Section 7.4.1, we show that it is sufficient for $Fresh_{list}$ to hold values of $\mathcal{O}$ seen by h and tagged with control cycles in $[r-d_t, r]$.

Consider h_w to be the host that writes to $\mathcal{O}$ (recall that any object is written to by a single host). Upon invoking a write to object $\mathcal{O}$ with a value v at cycle r, h_w updates its $Fresh_{list}$, by adding the tuple $< r, v >$ and deleting the tuple tagged with control cycle $r-c-1$. After this step, a write completes.

A host h that invokes a read operation to object $\mathcal{O}$ at cycle r, consults its $Fresh_{list}$. Host h returns the value tagged with the largest cycle number, say max_{cycle}, such that $max_{cycle} \leq r-d_t$. After this step the read invoked by h completes. If no such value exists, then a read returns the value tagged with the largest cycle within $Fresh_{list}$. Initially, for the first c cycles, reads return $\perp$, the initial value of $\mathcal{O}$ (assumed to be known by all hosts) and completes.

Every host h benefits from the underlying heartbeats sent in the DCS by piggybacking $Fresh_{list}(h)$ on these heartbeats. We consider that heartbeats are scheduled to be exchanged towards the end of a control cycle, i.e., after all invoked operations during a cycle have completed. A host h_i, upon receiving a heartbeat during cycle r from host h_j, updates its $Fresh_{list}(h_i)$ to:

$$Fresh_{list}(h_i) = Fresh_{list}(h_i) \bigcup Fresh_{list}(h_j),$$

for all values tagged with control cycles in $[r, r-c]$.

Algorithm 2 A Basic *TapeWorm* Algorithm.

```
Initialize:
set $Fresh_{list} = \{\perp\}^c$

Repeat periodically:
broadcast $< \mathsf{heartbeat}, \mathsf{Fresh}_{\mathsf{list}}, \mathsf{id} >$

upon event receive $< \mathsf{heartbeat}, \mathsf{Fresh}'_{\mathsf{list}}, \mathsf{id}' >$ do
    set $Fresh_{list} = Fresh'_{list} \bigcup Fresh_{list}$

upon event write $< \mathsf{v}, \mathsf{cycle} >$ do
    $update_{list} < \mathsf{v}, \mathsf{cycle} >$

upon event read $< cycle >$ do
    if $(cycle < c)$ then
        return $\perp$
    end if
    if $(< *, \mathsf{cyc} > \in \mathsf{Fresh}_{\mathsf{list}} : cyc = \max\{cyc \leq cycle - d_t\})$ then
        return $< *, \mathsf{cyc} >$
    end if
    return $< *, \mathsf{cyc} > \in \mathsf{Fresh}_{\mathsf{list}} : cyc = \max cyc$

Function $update\_list(< \mathsf{v}, \mathsf{cycle} >)$:
    set $Fresh_{list} = Fresh_{list} \bigcup < \mathsf{v}, \mathsf{cycle} >$
    remove $< *, cycle - c - 1 >$ from $Fresh_{list}$
```

Proof of Correctness.

We now prove the correctness of our *TapeWorm* algorithm.

Termination:

Both read and write operations in *TapeWorm* complete after performing a bounded number of local operations which requires a bounded amount of time and thus constitutes the required t_{op}. Termination is hence satisfied.

Agreement and freshness:

Recall that it is sufficient to guarantee the shared memory properties when the scheduler state is consistent (Section 7.2.4). The scheduler is consistent when all hosts $\in \nabla\mathcal{E}(r) \bigcup \nabla\mathcal{A}(r)$ at any cycle r agree on which hosts are in $\nabla\mathcal{A}(r+1)$ during cycle $r+1$ (see Section 7.2.3). Given that the scheduler state is consistent, we have the following:

Lemma 21. *For any host $h \in \nabla\mathcal{A}(r)$, i.e., belonging to the set of non-suspected hosts at cycle r, all hosts in $\{\nabla\mathcal{E}(r-1) \bigcup \nabla\mathcal{A}(r-1)\}$ have heard (directly or indirectly) a heartbeat sent by h during some cycle in* $[r-(d_t-1), r-1]$.

Proof. By the real-time detection property, a crashed host is declared failed (suspected) in less than d_t cycles after the crash. Consider a DCS of two hosts h_1 and h_2 and consider the following two executions:

- $e1$: an execution where h_2 crashes during cycle $r-(d_t-1)$ before h_2 sends any heartbeats.
- $e2$: an execution where h_1 and h_2 are both correct (do not crash) but lose all heartbeats sent (if any) during all cycles in $[r-(d_t-1), r-1]$.

Execution $e2$ can happen with positive probability as shown in the proof of Lemma 19. With respect to h_1, executions $e1$ and $e2$ are indistinguishable during $[r-(d_t-1), r-1]$, for any finite value of d_t (since h_1 cannot know if h_2 has failed or all messages from h_2 are lost).

By the real-time property of failure detection, h_1 suspects h_2 in execution $e1$ at cycle r. Since $e1$ and $e2$ are indistinguishable during $[r-(d_t-1), r-1]$ then h_1 suspects h_2 as failed at cycle r also in $e2$.

So, in order for h_2 not to be suspected at cycle r by h_1, h_1 has to hear (directly or indirectly) at least one heartbeat sent by h_2 during some cycle in $[r-(d_t-1), r-1]$. Since the scheduler state is assumed to be consistent, then all hosts $\in \nabla\mathcal{E}(r-1) \bigcup \nabla\mathcal{A}(r-1)$ have heard (directly or indirectly) a heartbeat sent by host h_i, at some cycle in $[r-(d_t-1), r-1]$, $\forall h_i \in \nabla\mathcal{A}(r)$. □

Lemma 22. *Let h be the host that writes to the shared object $\mathcal{O}$. If $h \in \nabla\mathcal{A}(r)$, then all hosts $\in \nabla\mathcal{E}(r-1) \bigcup \nabla\mathcal{A}(r-1)$ have in their $Fresh_{list}$ the value written by h at cycle $r-d_t$.*

Proof. By Lemma 21, if $h \in \nabla\mathcal{A}(r)$, then all hosts $\in \nabla\mathcal{E}(r-1) \bigcup \nabla\mathcal{A}(r-1)$ have heard (directly or indirectly) a heartbeat sent by host h at some cycle in $[r-(d_t-1), r-1]$. All heartbeats sent by h at some cycle in $[r-(d_t-1), r-1]$ contain the value written by h at cycle $r-d_t$, since (i) heartbeats are exchanged towards the end of a cycle (after operation in that cycle have completed) and (ii) h always appends to its heartbeats $Fresh_{list}(h)$, which contains all values written by h in cycles $[r-c, r]$, where $d_t < c$. In fact, it can be noticed that it is sufficient to only send $Fresh_{list}(h)$ containing values of $\mathcal{O}$ written by h in cycles $[r-d_t, r]$.

Since every host $\in \nabla\mathcal{E}(r-1) \bigcup \nabla\mathcal{A}(r-1)$ has heard (directly or indirectly) a heartbeat sent by host $h \in \nabla\mathcal{A}(r)$ at some cycle in $[r-(d_t-1), r-1]$, then every host has received from some host h_i a $Fresh_{list}(h_i)$ containing the value of object $\mathcal{O}$ written at cycle $r-d_t$. The reason is that any host h_j receiving $Fresh_{list}(h_i)$ performs: $Fresh_{list}(h_j) = Fresh_{list}(h_i) \bigcup Fresh_{list}(h_j)$, concluding the proof. □

By Lemma 22 and the algorithm description, if h, the writer host of some object $\mathcal{O}$ is not suspected at cycle r, then all hosts invoking a read to $\mathcal{O}$ during r return the value written at cycle $r-d_t$, which satisfies agreement and freshness since $c > d_t$.

If, however, h is suspected as crashed at cycle r, then by Lemma 21 and Lemma 22 all hosts $\in \nabla\mathcal{E}(r-1)\bigcup\nabla\mathcal{A}(r-1)$ have last heard (directly or indirectly), the heartbeat sent by h during cycle $r-d_t$ (otherwise h would suspected at a cycle $<\ r$). This heartbeat contains the value written by h at cycle $r-d_t$. No other heartbeats are heard from h (since otherwise h is suspected at cycle $> r$). The value tagged with $r-d_t$ hence has the highest cycle and is thus returned by any host issuing reads during cycles $\geq r$, which satisfies both agreement and freshness, as $c > d_t$. *TapeWorm* hence guarantees agreement and freshness when h is not suspected and when it is.

Note that from the proof of Lemma 22, it is sufficient that $Fresh_{list}$ of hosts is of size d_t and not c.

7.4.2 A Fresher TapeWorm Algorithm

In this section, we depict how *TapeWorm* can be modified such that read operations return fresher values, within the defined freshness interval. In other words, we describe how reads invoked in *TapeWorm* at cycle r can return values written at cycles in $[r-c,r]$, precisely in $[r-s,r]$ where $s < d_t$.

Every host in *TapeWorm* has a list, $Fresh_{list}$, that holds at cycle r the values of $\mathcal{O}$ seen by h and tagged with control cycles in $[r-c,r]$. For every value in $Fresh_{list}(h)$, h now keeps a list of host ids, called $seen_{cycle\#}$, and for each host id in $seen_{cycle\#}$ a list called $seenby_{id}$.

Upon invoking a write, with a value v, to object $\mathcal{O}$ at cycle r, h_w (the host writing to $\mathcal{O}$) appends its id to $seen_r$, besides updating its $Fresh_{list}$ as described in Section 7.4.1. After this step, a write completes.

Every host h piggybacks its $Fresh_{list}(h)$, $seen_{cycle\#}$ and $seenby_{id}$ lists to the heartbeats. Upon receiving a heartbeat at cycle r from host h_j, a host h_i updates its $Fresh_{list}(h_i)$ to:

$$Fresh_{list}(h_i) = Fresh_{list}(h_i)\bigcup Fresh_{list}(h_j),$$

for all values tagged with control cycles in $[r, r-c]$. For every value tagged with cycle r' such that r' is in $Fresh_{list}(h_j)$ but not in $Fresh_{list}(h_i)$, h_i adds its id to $seen_{r'}(h_i)$. Afterwards, for every value tagged with *cycle#* in the newly computed $Fresh_{list}(h_i)$, h_i performs:

$$seen_{cycle\#}(h_i) = seen_{cycle\#}(h_i)\bigcup seen_{cycle\#}(h_j),$$

and for every host h_k $(k \neq j)$ in the new $seen_{cycle\#}(h_i)$:

$$seenby_{h_k}(h_i) = seenby_{h_k}(h_i)\bigcup seenby_{h_k}(h_j).$$

For h_j in the new $seen_{cycle\#}(h_i)$, h_i performs:

$$seenby_{h_j}(h_i) = seenby_{h_j}(h_i)\bigcup seen_{cycle\#}(h_j).$$

A host h that invokes a read operation to object $\mathcal{O}$ at cycle r, consults its $Fresh_{list}(h)$ and returns the value tagged with the largest cycle, $max_{cycle} > r - d_t$, satisfying both properties below:

1. $seen_{max_{cycle}}$ contains the ids of all hosts that are not suspected in cycle $r+1$.
2. For every host h_k in $seen_{max_{cycle}}$, $seenby_{h_k}(h)$ contains the ids of all hosts not suspected in $r+1$.

The above two properties state that a host h returns a value at cycle r, if h knows that (i) every host in $\nabla\mathcal{A}(r+1)$ (non-suspected hosts at cycle $r+1$) has seen that value and (ii) every host in $\nabla\mathcal{A}(r+1)$ knows that every other host in $\nabla\mathcal{A}(r+1)$ has seen that value. After this step the read invoked by h completes. If no such value exists, then a read returns as it would do in the basic *TapeWorm* algorithm described in Section 7.4.1. As such, the correctness of this fresher *TapeWorm* follows from the correctness of the basic version.

7.5 Performance Analysis

In this section, we analyze certain aspects of *TapeWorm*'s performance. We specifically determine (a) the probability distribution of the values returned by read operations in the allowable freshness range, i.e., $[cycle_{read} - c, cycle_{read}]$ where $cycle_{read}$ is the cycle in which a read operation is invoked, (b) the bandwidth overhead of *TapeWorm*, and (c) optimizations of *TapeWorm* for static workloads.

7.5.1 The Freshness of Values Seen by Hosts

Recall that heartbeats, to which *TapeWorm* attaches information, are exchanged at the end of a cycle, after all tasks are executed. In other words, read and write operations get invoked and complete at a cycle r before the heartbeats at cycle r get exchanged. For simplicity, we consider $P_{ij}(t) = p \; \forall i, j$ and t. In other words, a message sent at any time and on any link has probability p of getting lost, where p is the same for all links and is independent of time and links. We study the fresher *TapeWorm* version depicted in Section 7.4.2 assuming that the writer host is correct, i.e., does not crash.

Assume that for all cycles in $[r+1, r+d_t]$:

$$\nabla\mathcal{A}(r+1) = \nabla\mathcal{A}(r+2) = ... = \nabla\mathcal{A}(r+d_t) = \nabla\mathcal{A},$$

i.e., the set of non-suspected hosts remains the same. We now compute the probability that a read, in *TapeWorm*, to a shared object $\mathcal{O}$ at any cycle in $[r+1, r+d_t]$ returns the value of $\mathcal{O}$ written at cycle r. In the fresher *TapeWorm* version, a read to $\mathcal{O}$ invoked at cycle $r+1$, returns the value written at cycle r with probability 0. This is due to the following fact: a host h sends its heartbeats at cycle s before it receives any heartbeats that some other hosts sent during s.

A read to $\mathcal{O}$ at cycle $r+2$, returns the value written at cycle r with probability $(1-p)^{|\nabla\mathcal{A}|^2(|\nabla\mathcal{A}|-1)}$.

For reads invoked at cycle $cyc \in [r+3, r+d_t-1]$: the probability that a read at cycle cyc returns the value of $\mathcal{O}$ written at cycle r can be approximated by the probability that each host in $\nabla\mathcal{A}$ hears (directly or indirectly) from every other host in $\nabla\mathcal{A}$ by cycle $cyc-2$, after which every host in $\nabla\mathcal{A}$ hears the heartbeat of every other host in $\nabla\mathcal{A}$. Let h be a host in $\nabla\mathcal{A}$. We denote by $\pi_h(cyc)$ the set of hosts in $\nabla\mathcal{A}$ that received a heartbeat from h (directly or indirectly) at some cycle in $[r+3, r+cyc-3]$ and by $\pi_{\bar{h}}(cyc)$ the set of hosts in $\nabla\mathcal{A}$ that did not receive a heartbeat from h (directly or indirectly) at some cycle in $[r+3, r+cyc-3]$. The probability that at cycle $cyc-2$ not all hosts in $\nabla\mathcal{A}$ hear from h is equal to the probability that at least one host in $\pi_{\bar{h}}(cyc)$ does not receive a heartbeat from any host in $\pi_h(cyc)$ in cycle $cyc-2$:

$$Prob(|\pi_{\bar{h}}(cyc)|) \times \sum_{x}^{|\pi_{\bar{h}}(cyc)|} \binom{|\pi_{\bar{h}}(cyc)|}{x} [(1-p)^{|\pi_h(cyc)|}]^x [1-(1-p)^{|\pi_h(cyc)|}]^{|\pi_{\bar{h}}(cyc)|-x},$$

where $Prob(\pi_{\bar{h}}(cyc))$ is the probability that $|\pi_{\bar{h}}(cyc)|$ hosts do not hear (directly or indirectly) from host h any heartbeat by cycle $cyc-3$. Thus, the probability that a read at cycle cyc returns the value of $\mathcal{O}$ written at cycle r is:

$$(1-p)^{|\nabla\mathcal{A}|} \times \prod_{\forall h \in \nabla\mathcal{A}} 1 - Prob(|\pi_h(cyc)|) \times \sum_{x}^{|\pi_{\bar{h}}(cyc)|} \binom{|\pi_{\bar{h}}(cyc)|}{x} [(1-p)^{|\pi_h(cyc)|}]^x [1-(1-p)^{|\pi_h(cyc)|}]^{|\pi_{\bar{h}}(cyc)|-x}.$$

A read to $\mathcal{O}$ invoked at cycle $r+d_t$, returns the value written at cycle r with probability:

$$1-(1-p)^{|\nabla\mathcal{A}|(|\nabla\mathcal{A}|-1)} - \sum_{cyc=r+3}^{r-d_t-1} (1-p)^{|\nabla\mathcal{A}|} \prod_{\forall h \in \nabla\mathcal{A}} 1 - Prob(|\pi_h(cyc)|) \times \sum_{x}^{|\pi_{\bar{h}}(cyc)|} \binom{|\pi_{\bar{h}}(cyc)|}{x} [(1-p)^{|\pi_h(cyc)|}]^x [1-(1-p)^{|\pi_h(cyc)|}]^{|\pi_{\bar{h}}(cyc)|-x}.$$

7.5.2 Bandwidth overhead of TapeWorm

Hosts in *TapeWorm* do not send any additional messages. However, hosts append information to heartbeats. In this section, we quantify the size of information being piggybacked to heartbeats. Every host in *TapeWorm* saves c values of object $\mathcal{O}$ relative to the last c values written to $\mathcal{O}$. In fact, it is sufficient for hosts to keep the last $d_t < c$ values of $\mathcal{O}$, as shown in Section 7.4.1. These values constitute the $Fresh_{list}$ of a host.

Consider that shared memory consists of x shared objects each capable of storing a value of m bits. The basic *TapeWorm* algorithm of Section 7.4.1 incurs a bandwidth overhead of $d_t \cdot x \cdot m$ bits/cycle per link. This overhead is relative to having each host attach the $Fresh_{list}$ to the heartbeat sent by that host every cycle.

In the fresher *TapeWorm* algorithm of Section 7.4.2 a host sends, in addition to the $Fresh_{list}$, the $seen_{cycle\#}$ and $seenby_{id}$ lists. The information in these two lists can be represented by an $n \times (n+1)$ binary matrix, where n is the total number of hosts in the system. The first column of this matrix represents information relative to the $seen_{cycle\#}$ list, and each row, excluding the first position, represents the information relative to $seenby_{id}$ list. A compact way of representing this matrix is to traverse the bits column by column (or row by row) and represent the binary data in ASCII format. The result incurs an additional overhead of

$$\frac{n(n+1)}{8}\ bits/cycle$$

per link compared to the basic *TapeWorm*.

Existing known failure detection and membership algorithms in control systems and real-time environments already implement an all-to-all broadcast mechanism for sending heartbeats and monitoring hosts [13, 35, 74, 154–156]. Often, these systems use Ethernet packets over UDP [13, 35, 154, 155] to send heartbeats. Classic heartbeats occupy only a very small fraction of the allowable minimum payload of an Ethernet packet (minimum UDP payload is 18 bytes).

Since TapeWorm only appends information to heartbeats, part of or maybe all information relative to TapeWorm (depending on the system and shared storage size) can be sent without any additional overhead by utilizing the unused payload of heartbeat messages.

7.5.3 DCS Shared Memory: Necessary and Sufficient Conditions

In this section, we determine the necessary and sufficient conditions for implementing the three properties of shared memory in a DCS (defined in Section 7.2.4).

We recall sufficient assumptions that define the family of algorithms to which *TapeWorm* belongs; we refer to this family of algorithms as *Parasite Algorithms*:

1. Hosts can exchange messages or append information to heartbeats exchanged over the DCS links. Precisely, in every cycle, each host either appends or does not append information to the heartbeat broadcasted in that cycle. Hosts have access to local synchronized clocks.

2. Hosts get suspected according to the real-time detection property: if a host h_i does not hear any of the heartbeats sent by host h_j during $[r-1, r-(d_t-1)]$ (directly from h_j or indirectly via other hosts), h_i suspects h_j at the beginning of cycle r.

Theorem 21. *A necessary condition for a parasite algorithm to deterministically implement termination, agreement and freshness in a DCS is:*
Every host h appends, to every heartbeat sent in $[r-1, r-(d_t-1)]$, any value of object $\mathcal{O}$ written during a cycle $\in [r-c, r]$, if h knows of any such value (where $r+1$ is the cycle during which a read operation is invoked by some host).

Proof. Let $r + 1$ be the cycle at which a read to object $\mathcal{O}$ is invoked by some host. To prove Theorem 21, we prove that if some host h, be it a writer or a reader, knows a value of $\mathcal{O}$ written during some cycle $\in [r - c, r]$ and does not append any such value to some heartbeat sent in $[r - 1, r - (d_t - 1)]$, one of the three properties is violated.

Consider a DCS with three hosts h_1, h_2 and h_3 sharing an object $\mathcal{O}$, where h_1 is the writer of $\mathcal{O}$. Also assume that at cycle $r + 1 = c + 1$, both h_2 and h_3 invoke a read to $\mathcal{O}$. To satisfy agreement and freshness, these reads should return the same value, that being a value written earliest at cycle 1. For illustration consider $c = d_t + 1$. In this case, since $r + 1 = c + 1$, a host h by Theorem 21 should append values of $\mathcal{O}$ to every heartbeat sent at all cycles $\in [3, c]$.

Case 1: h is the writer

Assume that h_1 decides not to append any information to the heartbeat it broadcasts at some cycle $r' \in [3, c]$. Consider an execution e satisfying all the below:

1. All three hosts are correct, i.e., no host fails.
2. Both h_2 and h_3 lose all heartbeats sent by h_1 at all cycles in $[1, r'[\bigcup]r', c]$. However, h_2 and h_3 both receive the heartbeat sent by h_1 at r'.
3. h_1 and h_2 receive all the heartbeats sent by h_3 during all cycles in $[1, c]$.
4. h_1 and h_3 receive all the heartbeats sent by h_2 during all cycles in $[1, c]$.

Execution e can happen with positive probability (since each host broadcasts a heartbeat at every cycle and every sent heartbeat has a positive probability of being lost). In execution e, the failure detector at the beginning of cycle $c + 1$, at all hosts, includes h_1, h_2 and h_3 in $\nabla\mathcal{A}$, since every host heard some heartbeat sent from every other host during the past $d_t - 1$ cycles, i.e., in $[3, c]$.

However, h_2 and h_3 do not see any value for $\mathcal{O}$ besides $\perp$, the initial value (prior to any write); h_2 and h_3 only receive the heartbeat sent by h_1 at cycle r' and this heartbeat has no values of $\mathcal{O}$ appended to it. As such, a read at cycle $c+1$ invoked by either h_2 or h_3 and satisfying termination completes and returns $\perp$ during $c + 1$, which violates the freshness property.

Case 2: h is a reader.

Assume now that h_2 does not append any value to the heartbeat it broadcasts at some cycle $r' \in [3, c]$. Consider an execution e' satisfying all the below:

1. All three hosts are correct.
2. h_3 loses all heartbeats sent by h_1 at all cycles in $[1, c]$.
3. h_2 receives all heartbeats sent by h_1, and h_1 receives all heartbeats sent by h_2 at all cycles in $[1, c]$.

4. h_3 loses the heartbeats sent by h_2 at all cycles in $[1, r'[\bigcup]r', c]$, but receives the heartbeat sent by h_2 at cycle $r' \in [3, c]$.

5. h_1 and h_2 receive all the heartbeats sent by h_3 at all cycles in $[1, c]$.

Execution e' can happen with positive probability (since messages can be probabilistically lost).

In e', the failure detector at the beginning of cycle $c + 1$, at all hosts, can include h_1, h_2 and h_3 in $\nabla\mathcal{A}$, since every host can hear (directly or indirectly) some heartbeat sent from every other host during the past $d_t - 1$ cycles, i.e., during $[3, c]$. Specifically h_3 can hear the heartbeat of h_1 indirectly via the heartbeat received from h_2 at cycle r'. However, since h_2 did not append any value for $\mathcal{O}$ at cycle r', h_3 does not see any value for $\mathcal{O}$ besides the initial value $\perp$. Note that h_2 knows values of $\mathcal{O}$ since it receives heartbeats from h_1 (h_1 is the writer of object $\mathcal{O}$ and appends values of $\mathcal{O}$ to all the sent heartbeats).

As such, a read at cycle $c + 1$ invoked by h_3 and satisfying termination completes and returns $\perp$ during $c + 1$, which violates the freshness property. $\square$

Theorem 22. *Consider that non-crashed hosts agree on the set $\nabla\mathcal{A}(r)$, such that the writer of a shared object $\mathcal{O}$ belongs to $\nabla\mathcal{A}(r)$. Then a sufficient condition for a parasite algorithm to deterministically ensure termination, agreement and freshness in a DCS, given a single object $\mathcal{O}$ to which a read operation is invoked by some host at cycle r, is:*
Each host appends any value of the shared object written at some cycle in $[r - c, r - d_t]$ (if this host has seen such a value) to every heartbeat sent during all cycles in $[r - d_t, r]$.

Proof. Consider a DCS of three hosts h_1, h_2 and h_3. Also consider h_1 to be the host that writes to $\mathcal{O}$. Let r be the cycle at which some host invokes a read to $\mathcal{O}$ and let v denote the value that h_1 writes to $\mathcal{O}$ at cycle $r - d_t$.

Lemma 23. *Consider that every host appends v (whenever it has received v) to every heartbeat sent during all cycles in $[r - d_t, r]$, where r is the cycle during which some host invokes a read operation to that shared object. Given that hosts agree on the set $\nabla\mathcal{A}(r)$, all non-crashed hosts at cycle r have v.*

Proof. Upon receiving a write to object $\mathcal{O}$ at cycle $r - d_t$, h_1 saves v and then completes, i.e., the write at h_1 locally returns before h_1 sends any heartbeat at $r - d_t$. h_1 hence appends v to all the heartbeats it sends in cycles $\in [r - d_t, r]$ (since h_1 has v). Agreeing about $\nabla\mathcal{A}(r)$, where $h_1 \in \nabla\mathcal{A}(r)$, means that all hosts in $\{\nabla\mathcal{E}(r) \bigcup \nabla\mathcal{A}(r)\}$ have heard (directly or indirectly) a heartbeat sent by h_1 during some cycle in $[r - d_t, r]$. Every host that hears v, appends v to every heartbeat it sends in $[r - d_t, r]$. This implies that every host in $\{\nabla\mathcal{E}(r) \bigcup \nabla\mathcal{A}(r)\}$, which was able to hear (directly or indirectly) a heartbeat from h_1, has received the value v. $\square$

By Lemma 23 all non-crashed hosts at cycle r would have received v. Any host that invokes a read at r can thus always complete after locally returning v, the value of $\mathcal{O}$ written at cycle $r - d_t$, deterministically. Readers and writers locally return satisfying termination. All readers at cycle r return v, thus satisfying agreement and freshness (since $c > d_t$). □

7.5.4 Optimizing *TapeWorm* under Static Workloads

We have assumed, so far, that read operations can be invoked at any cycle and this invocation time in not known. As such, in both versions of *TapeWorm*, the basic and the fresher, information about a shared object is appended to heartbeats of every host at every cycle. Based on the results of Section 7.5.3, we investigate optimizing (i) the rate of appending information to heartbeats and (ii) the number of hosts that need to append information to heartbeats in every cycle.

When workloads are static, i.e., hosts know the cycle at which read operations are invoked, then *TapeWorm* under certain assumptions can be optimized, in the sense that hosts do not need to append information on all heartbeats sent at all control cycles. Precisely, from Theorem 21 and Theorem 22, satisfying the three properties of shared memory requires two things: (i) all non-faulty hosts append information on every heartbeat only for a fixed time interval (d_t cycles) before the invocation of a read operation, and (ii) the writer host does not get suspected during that interval of d_t cycles.

This can be interpreted as follows. Let r be the cycle at which some host invokes a read operation to object $\mathcal{O}$. In static workloads, r is known by *TapeWorm*. Hosts in *TapeWorm* can now append information to heartbeats sent only during d_t cycles prior to r. This optimization is valid under the assumption that the writer host of the object $\mathcal{O}$ can communicate (directly or indirectly) with all non-crashed hosts during that interval.

7.6 Existing Real-Time Distributed Data Storage

Sharing data in real time has been addressed in various contexts, ranging from architectural design and synchronization for real-time shared memories in multi-core machines [189–192] to real-time replicated databases [55, 56, 183] and distributed memory. In this section, however, we focus on previous related work in distributed environments (similar to DCSs) rather than on works (i) on architectural memory designs or (ii) on accessing physical shared memory in multi-core machines in real time.

Aslinger and Son [55] presented two algorithms for database replication, each targeting a different data workload. The first algorithm is developed for non-static periodic workloads and ensures that replicas are updated at the minimum required rate. The second algorithm is designed for random workloads and adaptively changes the update policy of replicas based on previously observed data patterns. Both algorithms aim at increasing the scalability of real-time databases. Peddi and DiPippo [56] proposed a database replication algorithm for static periodic workloads, where all object locations and client data requirements are known a priori. The algorithm creates transactions from the operations issued to the database objects and feeds these transactions to a scheduling algorithm. If a schedule that meets all deadlines can be computed, then all read operations guarantee to return a fresh value (satisfying temporal consistency). Otherwise, the system specification must be reconsidered. Wei et al. [183] use a full replication mechanism to ensure data freshness of committed transactions in medium distributed databases (5 to 10 nodes). The algorithm consists of local heuristic feedback controllers and global load balancers. The local controllers manage the admission of incoming workloads, while the global balancers collect performance data from all sites and balance the workloads.

In contrast to [55, 56, 183], we consider message losses in addition to node failures. Overcoming the effect of message losses and meeting the requirements of shared memory in a DCS is highly non-trivial, as we show that it is impossible to be achieved (Section 7.3) without the aid of heartbeats and without requiring real-time crash detection.

Zou and Farnam [57] presented a real-time primary-backup replication scheme. The proposed scheme enforces *temporal* consistency (defined in Section 7.2.4) among data replicas and determines the corresponding rate at which update messages should be sent from the primary to the backup. Zou and Farnam [57] also discussed message losses. The authors assumed that messages can be lost with probability ρ, and denoted by P the probability of the temporal consistency desired to be achieved. Their solution to message losses, as such, dictates to increase the frequency of sending update messages from the primary to the backup to guarantee the required probability P.

In contrast to [57], we seek to deterministically guarantee the consistency of operations issued to a shared object, given system agreement regarding which hosts are considered alive and thus participate in executing tasks.

Xiong at al. [193] proposed MIRROR, a concurrency control algorithm for real-time replication control. MIRROR augments the optimistic two-phase locking (O2PL) algorithm with a state-based conflict resolution mechanism. The choice of the conflict resolution method is a dynamic function that either uses *Priority Abort* or *Priority Blocking* depending on the states of the transactions involved in the conflict. In Priority Abort, a conflict is resolved for the favor of the transaction with higher priority (by aborting a lower priority transaction currently holding the lock or blocking the lower priority transaction trying to acquire a lock held by a higher priority transaction). In Priority Blocking, a transaction is always blocked upon the encounter of a lock conflict and can acquire the lock after the lock is released. Lock requests are ordered by transaction priority.

Concurrency control mechanisms, such as [193], suffer from potential deadlock or unbounded blocking and thus do not comply with DCS-like requirements.

Other approaches addressed building real-time distributed hash tables (DHTs) [182, 194]. Qian et al. [182] designed and implemented a Chord-based DHT. Given the periodic structure of requests considered in [182], a cyclic executive is used to schedule the jobs that subsequent nodes in the Chord overlay network should execute to serve a request. Skodzik et al. [194] extended Kad, an implementation variant of the P2P Kademlia protocol, by a TDMA based mechanism in order to make Kad suitable for hard real-time constraints.

In contrast to [182, 194], we require each operation, be it a read or a write, to return and complete based on performing a bounded number of local steps. As we show in Lemma 20, waiting to reliably transmit any message in order for an operation to complete, could take an arbitrary long time in the communication system we consider in this chapter.

7.7 Chapter Summary

This chapter investigated how to build a shared memory abstraction for distributed control systems (DCSs). Such an abstraction constitutes a basic building block for real-time shared storage functionalities (like real-time DHTs, key-value stores etc.), which are highly demanded in DCSs.

We determined the guarantees that a shared memory abstraction should deliver to applications accessing it via read and write operations. We proved that such guarantees are impossible to implement deterministically, in the presence of host crashes and message losses (in the sense described in Section 7.3).

We presented *TapeWorm*, an algorithm that circumvents this impossibility and guarantees the desired shared memory properties for applications. *TapeWorm* adopts a white-box approach in which heartbeat messages of the failure detector component running in a DCS, are used as a means of transporting information. We also conducted a mathematical analysis quantifying the performance of *TapeWorm* and showcased ways for adapting and optimizing *TapeWorm* to application needs and workloads.

The main contributions of this chapter can be summarized as follows:

1. A first precise derivation of the necessary guarantees that a shared memory abstraction must provide in DCSs.
2. Theoretical proofs showing that such guarantees are impossible to implement using traditional approaches [55–59], e.g., using a black-box approach.
3. *TapeWorm*, an algorithm that circumvents the above impossibility by following a white-box approach directly utilizing failure detector algorithms of DCSs. *TapeWorm* implements the required shared memory guarantees for applications running in a DCS.
4. A mathematical analysis quantifying the performance of *TapeWorm* and showcasing ways of adapting and optimizing *TapeWorm* respectively to application needs and workloads.

Thesis Conclusions and Remarks Part III

It's time to say goodbye, but I think goodbyes are sad and I'd much rather say hello.
Hello to a new adventure.
— E. Harwell

This part summarizes the work presented in this dissertation. We recall the main motivation and challenges of the problems addressed in this thesis. We then highlight the achieved results and their impact on research in areas of automation for smart grid, sensor networks and distributed control systems.

We also discuss interesting open future questions related to the studied topics.

8 Summary and Open Questions

This thesis investigated how to build distributed abstractions in the context of automation systems for control, smart grid and sensor networks, also known as *cyber-physical systems*. Such real-world physical systems adhere to a different set of requirements and constraints compared to classic distributed systems. Existing solutions to distributed problems, developed based on classic distributed computing assumptions, may hence be not fully portable, even sometimes unusable in a cyber-physical context.

To this end, we studied how to build four abstractions, given the design constraints and needs of such automation systems. Our investigation commenced from communication links and went all the way up to applications. We recall in what follows these abstractions, summarizing our results and contributions. We discuss also potential interesting open questions.

8.1 Energy-efficient Reliable Communication

In Chapter 4 of this thesis we presented an analytic study describing how energy-efficient reliable communication, that is synchronous with high probability, can be built over unreliable links. The analysis was conducted for a time-varying lossy link capturing the dynamic communication quality of network links for the systems in consideration.

We performed our analyses considering the main forms of Ack/Nack feedback mechanisms. We obtained under reliable feedback, a closed form of the policy which determines when to transmit over the link, in order to avoid sending messages when the link is in a bad state (losing messages) and hence to minimize re-transmissions. Minimizing the number of re-transmissions leads to minimizing the total transmission energy needed for sending a message. Combined with this closed form solution, we also identified the necessary conditions under which transmission is never suspended forever, for example when the policy is "very conservative" since the energy lost due a message loss is much more expensive than taking the risk to transmit. We provided as well a probabilistic bound on the total time to deliver a message. In short, we presented an implementable form of an energy-efficient reliable communication, guaranteeing high probability synchrony.

We also performed, to the best of our knowledge, a first analysis studying the impact of lossy feedback on optimal transmission policies. Optimality of a policy in this context is defined with respect to a defined energy-throughput-latency trade-off, where the policy needs to maximize a reward function, given a weighted function consisting of rewards for transmitting messages successfully and costs for transmitting messages and losing them. We showed that easy implementable forms of the desired communication service can also be obtained depending on the utilized feedback mechanism.

Possible interesting open questions may look into the case of investigating optimal transmission policies under unreliable feedback considering multiple energy levels of transmission that can be used to send a message over the link.

8.2 Failure Detection

In Chapter 5, we investigated failure detection in systems embodying asynchrony via probabilistic synchronous communication. In contrast to the conventional distributed computing assumptions when building failure detectors, which hinged on link synchrony guarantees that need to hold deterministically forever, we adopted a more realistic link behavior motivated by networking views on actual packet loss. We showed that, under lossy probabilistic communication links, "$\diamond\mathcal{S}$ with probability 1" cannot be implemented, while "consensus with probability 1" can be implemented without requiring any randomness in the algorithm itself. We recall that $\diamond\mathcal{S}$ has been established, in some sense [110], to be the weakest failure detector to implement consensus. We accordingly refine the notion of failure detectors defining $\diamond\mathcal{S}^*$ which does not require any "forever" guarantee from the underlying network. We show that $\diamond\mathcal{S}^*$ can be implemented in system $\mathcal{N}$ and even efficiently. In addition, we show that $\diamond\mathcal{S}^*$ can replace $\diamond\mathcal{S}$ in several deterministic consensus algorithm and yields an algorithm that solves "consensus with probability 1". We also generalize this result to encompass (i) a more general set of problems, which we call decisive problems, and (ii) other eventual failure detectors besides $\diamond\mathcal{S}$.

Open questions that constitute interesting potential for future work may investigate the weakest probabilistic system to implement $\diamond\mathcal{S}^*$ or the solvability of problems, besides the ones discussed in this thesis (i.e., the set of decisive problems, Section 5.5), using our new notion of failure detectors.

8.3 Real-Time Membership

In Chapter 6, we defined the membership properties essential for the proper operation of distributed control systems (DCSs) running cyclic control applications. In their implementable form, these properties take the form of a probabilistic real-time membership service, which we called SYMS. We proposed *ViewSnoop*, an algorithm based on exchanging local views between hosts in order to implement SYMS.

We evaluated *ViewSnoop* both analytically as well as experimentally via an implementation within FASA, an industrial DCS framework. Our results convey that *ViewSnoop* provides a more dependable service, compared to membership mechanisms based on classic heartbeats, thus enhancing a DSC's reliability. Additionally, *ViewSnoop* can distinguish host failures from message losses. Schedulers in DCSs can thus compute better configurations, in the sense that tasks requiring information exchange are not allocated to hosts connected by bad links.

Devising optimizations that can be augmented to *ViewSnoop* making it adaptive to changing network conditions, are open interesting questions. Such optimizations allow *ViewSnoop* to operate at even lower costs when the network behaves good and make our algorithm more suited for very large-scale distributed systems.

8.4 Real-Time Distributed Shared Memory

In Chapter 7, we investigated how to build a distributed shared memory (DSM) abstraction for distributed control systems (DCSs). Such an abstraction makes programming control application significantly easier and results in less programming errors. Besides, DSM constitutes a basic building block for real-time shared storage functionalities (like real-time DHTs, key-value stores etc.), which are highly demanded in DCSs.

We determined the necessary guarantees that a DSM abstraction should deliver to applications accessing it. Given that hosts can crash and messages can be lost, we proved that such guarantees are impossible to implement deterministically, in traditional ways where algorithms (i) do not rely on failure detection or (ii) use failure detectors as software blocks. We presented an algorithm, which we called *TapeWorm*, capable of circumventing this impossibility. *TapeWorm*, hence, guarantees the desired shared memory properties for DCS applications. *TapeWorm* adopts a white-box approach in which heartbeat messages of the failure detector component running in a DCS, are used as a means of transporting information. We also conducted a mathematical analysis quantifying the performance of *TapeWorm* and showcased ways for adapting and optimizing *TapeWorm* to application needs and workloads.

Interesting questions that are still open for investigation could potentially address issues related to solution designs of DSM algorithms when monitoring schemes do not employ all-to-all (broadcast) communication between hosts.

Appendices and Bibliography Part IV

A Deriving Closed Form Expressions of The Belief Function w^*

This Appendix details the derivation of the closed form expressions of w^*.

A.0.1 Positively Correlated Links

$$\tau^k(w) \text{ monotonically tends to } \pi_g \text{ , as } k \to \infty. \tag{A.1}$$

$$1 - \beta > \pi_g > \alpha. \tag{A.2}$$

Case 1: $w^* \geq \pi_g$

If $w < w^*$, from (A.1) $\tau^k(w) \leq w^*$ and thus the optimal decision is to also idle the link. Following from (A.2) $V(\alpha) = c_d + \gamma c_d + \gamma^2 c_d + ... = \frac{c_d}{1-\gamma}$.

If $w > w^*$, then $V(w) = w(r - c_p) + c_p + \gamma[wV(1-\beta) + (1-w)V(\alpha)]$. If($w^* \geq 1 - \beta$), then $V(1-\beta) = \frac{c_d}{1-\gamma}$. By continuity of $V(w)$ at w^* we have $w^*(r - c_p) + c_p + \gamma[w^*V(1-\beta) + (1-w^*)V(\alpha)] = c_d + \gamma V(\tau(w^*))$, which yields: $w^* = \frac{c_d - c_p}{r - c_p}$. However, if $w^* < 1 - \beta$, then

$$\begin{aligned} V(1-\beta) &= (1-\beta)(r-c_p) + c_p + \gamma[(1-\beta)V(1-\beta) + (\beta)\frac{c_d}{1-\gamma}] \\ &= \frac{(1-\beta)(r-c_p) + c_p + \gamma\beta\frac{c_d}{1-\gamma}}{1-\gamma(1-\beta)}. \end{aligned}$$

By continuity of V at w^*

$$w^*(r-c_p) + c_p + \gamma[w^*\frac{(1-\beta)(r-c_p) + c_p + \gamma\beta\frac{c_d}{1-\gamma}}{1-\gamma(1-\beta)} + (1-w^*)\frac{c_d}{1-\gamma}] = \frac{c_d}{1-\gamma}.$$

After reduction:

$$w^* = \frac{(c_d - c_p)(1 - \gamma(1-\beta))}{r - c_p(1-\gamma) - \gamma c_d},$$

$$\frac{dw^*}{dc_d} = \frac{(1-\gamma(1-\beta))(r-(1-\gamma)c_p) - \gamma c_p(1-\gamma(1-\beta))}{(r - c_p(1-\gamma) - \gamma c_d)^2} > 0.$$

which shows that w^* is a strictly increasing function of c_d.

Case 2: $w^* < \pi_g$

By (A.1), $\tau^k(w^*) > w^* \; \forall \, k > 1$ thus

$$\begin{aligned} V(\tau(w^*)) &= \tau(w^*)(r - c_p) + c_p + \gamma[\tau(w^*)V(1-\beta) + (1-\tau(w^*))V(\alpha)] \\ &= c_p + \gamma V(\alpha) + \tau(w^*)[(r-c_p) + \gamma(V(1-\beta) - V(\alpha))]. \end{aligned} \tag{A.3}$$

By (A.2), we have $(1-\beta) > w^*$ and

$$V(1-\beta) = (1-\beta)(r-c_p) + c_p + \gamma[(1-\beta)V(1-\beta) + \beta V(\alpha)] = \frac{(1-\beta)(r-c_p) + c_p + \gamma\beta V(\alpha)}{1-\gamma(1-\beta)}. \tag{A.4}$$

$$(r - c_p) + \gamma[V(1-\beta) - V(\alpha)] = \frac{(r-c_p) + \gamma c_p - \gamma(1-\gamma)V(\alpha)}{1-\gamma(1-\beta)}. \tag{A.5}$$

By continuity of V at w^*

$$\begin{aligned} c_p + \gamma V(\alpha) + w^*[(r-c_p) + &\gamma[V(1-\beta) - V(\alpha)] = \\ & c_d + \gamma[c_p + \gamma V(\alpha) + \tau(w^*)[(r-c_p) + \gamma(V(1-\beta) - V(\alpha))]]. \end{aligned} \tag{A.6}$$

Replace (A.5) in (A.6)

$$\begin{aligned} c_p + \gamma V(\alpha) + w^*[&\frac{(r-c_p) + \gamma c_p - \gamma(1-\gamma)V(\alpha)}{1-\gamma(1-\beta)}] = \\ & c_d + \gamma[c_p + \gamma V(\alpha) + \tau(w^*)[\frac{(r-c_p) + \gamma c_p - \gamma(1-\gamma)V(\alpha)}{1-\gamma(1-\beta)}]. \end{aligned} \tag{A.7}$$

By $\tau(w^*) = \alpha + w^*(1-\beta-\alpha)$ reduce (A.7) and find w^*

$$w^* = 1 - \frac{[1-\gamma(1-\beta)][r - c_d]}{[1-\gamma(1-\beta-\alpha)][(r-c_p) + \gamma c_p - \gamma(1-\gamma)V(\alpha)]}. \tag{A.8}$$

Thus to find the value of w^*, we have to find $V(\alpha)$.

If($w^* \leq \alpha$), then

$$V(\alpha) = \alpha(r - c_p) + c_p + \gamma[\alpha V(1-\beta) + (1-\alpha)V(\alpha)] = \frac{\alpha(r - c_p) + c_p + \gamma\alpha V(1-\beta)}{1 - \gamma(1-\alpha)}.$$

By the assumption of positive memory, $1 - \beta > \alpha$, then

$$V(1-\beta) = \frac{(1-\beta)(r - c_p) + c_p + \gamma\beta V(\alpha)}{1 - \gamma(1-\beta)}.$$

Solving the two equations yields

$$V(\alpha) = \frac{\alpha(r - c_p) + c_p - \gamma c_p(1-\beta-\alpha)}{(1-\gamma)(1-\gamma(1-\beta-\alpha))}. \tag{A.9}$$

Replace (A.9) in (A.8), to get $w^* = \frac{c_d - c_p}{r - c_p}$. It is easy to see that w^* is strictly increasing function in c_d.

If ($w^* > \alpha$), then by (A.1), for some k we have

$$\tau^k(\alpha) < w^* \leq \tau^{k+1}(\alpha).$$

Thus,

$$V(\alpha) = c_d + \gamma c_d + ... + \gamma^k c_d + \alpha^{k+1} V(\tau^{k+1}(\alpha)) = \frac{1-\gamma^{k+1}}{1-\gamma} c_d + \gamma^{k+1} V(\tau^{k+1}(\alpha)). \tag{A.10}$$

$$\begin{aligned} V(\tau^{k+1}(\alpha)) &= c_p + \gamma V(\alpha) + \tau^{k+1}(\alpha)[(r - c_p)\gamma(V(1-\beta) - V(\alpha))] \\ &= c_p + \gamma V(\alpha) + \tau^{k+1}(\alpha)[\frac{(r - c_p) + \gamma c_p - \gamma(1-\gamma)V(\alpha)}{1 - \gamma(1-\beta)}]. \end{aligned} \tag{A.11}$$

Replace (A.11) in (A.10)

$$(r - c_p) + \gamma c_p - \gamma(1-\gamma)V(\alpha) = \frac{[1-\gamma(1-\beta)][r(1-\gamma^{k+2}) - c_p(1-\gamma) - \gamma(1-\gamma^{k+1})c_d]}{(1-\gamma^{k+2})(1-\gamma(1-\beta)) + \gamma^{k+2}(1-\gamma)\tau^{k+1}(\alpha)}. \tag{A.12}$$

Replace (A.12) in (A.8)

$$w^* = 1 - \frac{[r - c_d][(1-\gamma^{k+2})(1-\gamma(1-\beta)) + \gamma^{k+2}(1-\gamma)\tau^{k+1}(\alpha)]}{[1-\gamma(1-\beta-\alpha)][r(1-\gamma^{k+2}) - c_p(1-\gamma) - \gamma(1-\gamma^{k+1})c_d]}. \tag{A.13}$$

To write w^* in a more readable form, let:

$A(k) = \frac{(1-\gamma^{k+2})(1-\gamma(1-\beta)) + \gamma^{k+2}(1-\gamma)\tau^{k+1}(\alpha)}{1-\gamma(1-\beta-\alpha)}$,

$B(k) = r(1-\gamma^{k+2}) - c_p(1-\gamma)$, $D(k) = \gamma(1-\gamma^{k+1})$.

w^* can then be written as

$$w^* = 1 - A(k)\frac{r - c_d}{B(k) - D(k)c_d}. \tag{A.14}$$

We can show that w^* is strictly increasing in c_d by

$$\frac{dw^*}{dc_d} = A(k)\frac{(r-c_p)(1-\gamma)}{(B(k)-D(k)c_d)^2} > 0.$$

Since $\tau^k(\alpha) < w^* \leq \tau^{k+1}(\alpha)$ where
$\tau^k(w) = \pi_g - (1-\beta-\alpha)^k(\pi_g - w)$, then $k = \lceil \frac{\ln(1-\frac{w^*}{\pi_g})}{\ln(1-\beta-\alpha)} \rceil - 2$.

This concludes all possible cases when the link has a positive memory. Similar analysis and computations are carried for the negatively correlated case.

A.0.2 Negatively Correlated

When the link has a negative memory, i.e. $1-\beta-\alpha < 0$, we study the following cases:
Case1:$1-\beta = 0;\ \alpha = 1$

$$\begin{cases} V(1) = r + \gamma V(0) \\ V(0) = c_d + \gamma V(1) \end{cases}$$

We solve the two equations and obtain

$$\begin{cases} V(1) = \frac{r+\gamma c_d}{1-\gamma^2} \\ V(0) = \frac{c_d+\gamma r}{1-\gamma^2} \end{cases}$$

if($w^* \geq 0.5$) This means that $1-w^* < w^*$, $\tau(w^*) = 1-w^*$ and $\tau(1-w^*) = w^*$

$$V(1-w^*) = c_d + \gamma V(w^*).$$

By continuity of V at w^* we have

$$V(w^*) = c_d + \gamma V(1-w^*) = c_d + \gamma(c_d + \gamma V(w^*)) = \frac{c_d}{1-\gamma}. \tag{A.15}$$

$$\begin{aligned} \frac{c_d}{1-\gamma} &= w^*(r-c_p) + c_p + \gamma[w^*V(0) + (1-w^*)V(1)] \\ &= c_p + \gamma\frac{r+\gamma c_d}{1-\gamma^2} + w^*[(r-c_p) + \gamma[\frac{c_d+\gamma r}{1-\gamma^2} - \frac{r+\gamma c_d}{1-\gamma^2}]. \end{aligned}$$

$$w^* = \frac{c_d(1+\gamma-\gamma^2) - \gamma r - c_p(1-\gamma^2)}{(1-\gamma)[\gamma c_d + r - c_p(1+\gamma)]}. \tag{A.16}$$

$$\frac{dw^*}{dc_d} = \frac{(1+\gamma-\gamma^2)(1-\gamma)r + \gamma^2(1-\gamma)r - c_p(1-\gamma^2)}{(1-\gamma)^2[\gamma c_d + r - c_p(1+\gamma)]^2} > 0. \tag{A.17}$$

It can also be seen that w^* is strictly increasing in c_d.
if($w^* < 0.5$)
This means $(1-w^*) > w^*$, so

$$V(1-w^*) = (1-w^*)(r-c_p) + c_p + \gamma[(1-w^*)V(0) + w^*V(1)]. \tag{A.18}$$

and by the continuity of V at w^*

$$\begin{aligned} w^*(r-c_p) + c_p + \gamma[w^*V(0) + (1-w^*)V(1)] &= c_d + \gamma V(1-w^*) \\ &= c_d + \gamma[(1-w^*)(r-c_p) + c_p \\ &\quad + \gamma[(1-w^*)V(0) + w^*V(1)]]. \end{aligned}$$

$$w^* = \frac{c_d - c_p + \gamma(r + \gamma V(0) - V(1))}{(1+\gamma)[r - c_p + \gamma(V(0) - V(1))]} = \frac{c_d - c_p + \gamma^3 r}{(1-\gamma^2)[\gamma c_d + r - c_p(1+\gamma)]}. \tag{A.19}$$

$$\frac{dw^*}{dc_d} = \frac{(1-\gamma^2)(1-\gamma^4)r - c_p(1-\gamma^2)}{(1-\gamma^2)^2[\gamma c_d + r - c_p(1+\gamma)]^2} > 0. \tag{A.20}$$

which is strictly increasing in c_d.

Case2: $0 < \alpha - (1-\beta) < 1$

In this case for any $w \in [0,1]$, $\tau^{2k}(w)$ and $\tau^{2k+1}(w)$ converge from opposite directions to π_g as $k \to \infty$.

If($w^* \geq \pi_g$)

Then $\tau^k(w^*) < w^*, \ \forall\ k$. But, $V(\tau(w^*)) = \frac{c_d}{1-\gamma}$. By continuity of $V(w)$ at w^* we have

$$w^*(r-c_p) + c_p + \gamma[w^*V(1-\beta) + (1-w^*)V(\alpha)] = c_d + \gamma V(\tau(w^*)).$$

$$w^* = \frac{\frac{c_d}{1-\gamma} - c_p - \gamma V(\alpha)}{r - c_p + \gamma(V(1-\beta) - V(\alpha))}. \tag{A.21}$$

Following from $0 < \alpha - 1 - \beta < 1$, we have $1 - \beta < w_0$ and $\alpha > \pi_g$. So $V(1-\beta) = c_d + \gamma V(\tau(1-\beta))$. If($\tau(1-\beta) \leq \pi_g$) $\tau^k(1-\beta) < w^*$, and hence $V(\tau(1-\beta)) = \frac{c_d}{1-\gamma}$.

If($\tau(1-\beta) > \pi_g$), then

$$V(\tau(1-\beta)) = \tau(1-\beta)(r-c_p) + c_p + \gamma[\tau(1-\beta)V(1-\beta) + (1-\tau(1-\beta))V(\alpha)].$$

If($\alpha \leq w^*$) We have $\tau^k(\alpha) < w^*$, $\forall\, k$, and thus $V(\alpha) = \frac{c_d}{1-\gamma}$.

If($\alpha > w^*$)

$$V(\alpha) = \alpha(r-c_p) + c_p + \gamma[\alpha V(1-\beta) + (1-\alpha))V(\alpha)] = \frac{\alpha(r-c_p) + c_p + \gamma\alpha V(1-\beta)}{1-\gamma(1-\alpha)}.$$

$$r - c_p + \gamma(V(1-\beta) - V(\alpha)) = \frac{\gamma(1-\gamma)V(1-\beta) + (1-\gamma)r - c_p}{1-\gamma(1-\alpha)}.$$

Given that $\tau(1-\beta) = \alpha - (1-\beta)(\alpha - 1 + \beta) < \alpha$, divide the interval $[\pi_g, 1]$ into the 3 sub-intervals $[\pi_g, \tau(1-\beta)]$, $[\tau(1-\beta), \alpha]$ and $[\alpha, 1]$.

If($w^* \in [\alpha, 1]$), then

$V(\alpha) = V(1-\beta) = \frac{c_d}{1-\gamma}$ and thus by replacing in (A.21) we get $w^* = \frac{c_d - c_p}{r - c_p}$.

If($w^* \in [\tau(1-\beta), \alpha]$), then $V(1-\beta) = \frac{c_d}{1-\gamma}$ and

$$w^* = \frac{\frac{c_d}{1-\gamma} - c_p - \gamma\frac{\alpha(r-c_p)+c_p+\gamma\alpha V(1-\beta)}{1-\gamma(1-\alpha)}}{\frac{\gamma(1-\gamma)V(1-\beta)+(1-\gamma)r-c_p}{1-\gamma(1-\alpha)}} = \frac{c_d(1+\alpha\gamma) - \gamma\alpha r - c_p}{\gamma c_d + (1-\gamma)r - c_p}.$$

$$\frac{dw^*}{dc_d} = \frac{(1+\alpha\gamma)(1-\gamma)r + \alpha\gamma^2 r - c_p(1+\gamma^2)}{(\gamma c_d + (1-\gamma)r - c_p)^2} > 0. \tag{A.22}$$

which is strictly increasing in c_d.

If ($w^* \in [\pi_g, \tau(1-\beta)]$)

$$V(\alpha) = \frac{\alpha(r + \gamma V(1-\beta)) + c_p(1-\alpha)}{1-\gamma(1-\alpha)}.$$

$$r - c_p + \gamma(V(1-\beta) - V(\alpha)) = \frac{(1-\gamma)(r+\gamma V(1-\beta)) - c_p}{1-\gamma(1-\alpha)}.$$

$$\begin{aligned} r + V(1-\beta) &= c_d - \gamma c_p + \gamma^2 V(\alpha) + \gamma\tau(1-\beta)[r - c_p + \gamma(V(1-\beta) - V(\alpha))] \\ &= r + \gamma c_d + \frac{r + \gamma V(1-\beta)}{1-\gamma(1-\alpha)}(\gamma^3\alpha + \gamma^2(1-\gamma)\tau(1-\beta)) + \gamma^2\frac{1-\tau(1-\beta)}{1-\gamma(1-\alpha)}c_p. \end{aligned}$$

Replace in (A.21)

$$w^* = \frac{(1+\alpha\gamma(1-\gamma)+\gamma^2(1-\beta)(\alpha-(1-\beta)))c_d - \gamma\alpha r}{(1-\gamma)(\gamma c_d + r - c_p(1+\gamma))} - \frac{c_p(1-\alpha\gamma^2+\gamma^2(1-\beta)(\alpha-(1-\beta)))}{(1-\gamma)(\gamma c_d + r - c_p(1+\gamma))}.$$

which can be shown to be strictly increasing in c_d.

If$(w^* < \pi_g)$

$$V(\alpha) = \alpha(r-c_p) + c_p + \gamma[\alpha V(1-\beta) + (1-\alpha))V(\alpha)] = \frac{\alpha(r-c_p)+c_p+\gamma\alpha V(1-\beta)}{1-\gamma(1-\alpha)}.$$

$$r - c_p + \gamma(V(1-\beta) - V(\alpha)) = \frac{\gamma(1-\gamma)V(1-\beta) + (1-\gamma)r - c_p}{1-\gamma(1-\alpha)}.$$

$\tau(w^*) > \pi_g > w^*$ and thus by continuity of $V(w)$ at w^*, we obtain

$$\begin{aligned} w^* &= \frac{c_d - (1-\gamma)(c_p + \gamma V(\alpha)) + \gamma\alpha[r - c_p + \gamma(V(1-\beta) - V(\alpha))]}{[r - c_p + \gamma(V(1-\beta) - V(\alpha))][1-\gamma(1-\beta-\alpha)]} \\ &= \frac{[1-\gamma(1-\alpha)][c_d - c_p]}{[\gamma(1-\gamma)V(1-\beta) + (1-\gamma)r - c_p][1-\gamma(1-\beta-\alpha)]}. \end{aligned}$$

If$(0 \leq w^* \leq 1-\beta)$

$$V(1-\beta) = \frac{(1-\beta)(r-c_p)+c_p+\beta\gamma V(\alpha)}{1-\gamma(1-\beta)} = \frac{r[1-\beta-\gamma(1-\beta-\alpha)]+\beta c_p}{(1-\gamma)[1-\gamma(1-\alpha)(1-\beta)]}.$$

Thus w^* is found to be

$$w^* = \frac{[1-\gamma(1-\alpha)][c_d - c_p]}{[1-\gamma(1-\alpha)](r-c_p)} = \frac{c_d - c_p}{(r-c_p)}. \tag{A.23}$$

It can be clearly noticed that w^* is strictly increasing in c_d.

If$(1-\beta < w^* < \pi_g)$, then $\tau(1-\beta) > w^*$ and we have

$$r + \gamma V(1-\beta) = \frac{(\gamma c_d + r)[1-\gamma(1-\alpha)] + \gamma^2(1-\tau(1-\beta))c_p}{(1-\gamma)[1-\alpha\gamma+\gamma^2(1-\beta)(\alpha-(1-\beta))]}.$$

So the value of w^* would be

$$w^* = \frac{(c_d - c_p)(1+\gamma(1-\beta))}{[\gamma c_d - (1+\gamma)c_p + r]}.$$

whose derivative with respect to c_d is positive, meaning that w^* is strictly increasing in c_d. This concludes the negative correlated case.

B Failure Detectors Beyond $\diamond\mathcal{S}^*$

B.1 Other Probabilistic Failure Detectors

Since $\diamond\mathcal{S}^*$ can be implemented in our system, we study the possibility of implementing meaningful probabilistic variants of failure detectors which noticeably simplify the design of distributed algorithms, namely perfect failure detection. A perfect failure detector module, $\mathcal{P}$, guarantees in addition to strong completeness, the strong accuracy property, which says that no process is suspected before it crashes. Distributed algorithms, specifically those solving consensus, using $\mathcal{P}$ are easy to design due to their implicit reliance on the strong accuracy property of $\mathcal{P}$ to guarantee some safety property [117]. The liveness of such algorithms typically relies on the strong completeness. It is important to note that on the contrary consensus algorithms based on unreliable failure detectors [7] usually rely on the eventual accuracy to guarantee liveness of the algorithm. We thus define $\mathcal{P}^*$, a probabilistic variant of $\mathcal{P}$, as a failure detector that guarantees strong accuracy and probabilistic strong completeness, where the latter can be formally defined as: *Probabilistic Strong Completeness:* Eventually every process that crashes can be suspected, with positive probability, by every correct process.

Theorem 23. *It is impossible to implement the failure detector $\mathcal{P}^*$ in $\mathcal{N}$, even if at most one process can fail.*

Proof. Consider a network of $n = 2$ processes, p_1 and p_2, and the following executions:

$e1$. an execution where process p_2 fails at time t.

$e2$. an execution where processes p_1 and p_2 are both correct but get partitioned at time t.

By the probabilistic strong completeness there is a time after which p_1 in execution $e1$ has a positive probability of suspecting p_2. By Lemma 12, executions $e1$ and $e2$ can be indistinguishable to p_1 for any finite duration after t. Accordingly there is a time where p_1 has a positive probability of suspecting p_2 in execution $e2$. By the strong accuracy property of $\mathcal{P}^*$ a correct process is never suspected. Thus the probability of p_1 suspecting p_2 in $e2$ should be 0 at all times, a contradiction concluding the proof . □

We show now that $\diamond\mathcal{P}^*$, a variant of $\diamond\mathcal{P}$ can be implemented. Failure detector $\diamond\mathcal{P}^*$ guarantees: (i) *strong bounded completeness* and (ii) *probabilistic eventual strong accuracy:* Consider any finite duration Δt. With probability 1 the following occurs: there exists infinitely many time instants t_G, such that after each all correct processes are not suspected by any correct process for the interval $t_G + \Delta t$.

Theorem 24. *It is possible to implement $\diamond\mathcal{P}^*$, in $\mathcal{N}$, even if $n-1$ processes can crash.*

Proof. The proof is similar to that of Theorem 10. Following from Lemma 13, if all correct processes broadcast messages forever (send an infinite number of messages), then with probability 1, the following will be observed: any finite number of consecutive messages, e.g., Δt messages, is successfully transmitted, i.e., with no losses.

As such, an algorithm which satisfies both characteristics below for example implements $\diamond\mathcal{P}^*$:

1. All processes periodically (say with period Δt_1 chosen arbitrarily) broadcast messages forever.
2. Process p_i suspects another process p_j only if p_i receives no message form p_j for a period strictly greater than Δt_1.

□

B.2 Substituting Global Clock by Unsynchronized Local Clocks

We briefly discuss in this section how the global clock assumption can be substituted with local clocks which do not need to be synchronized. For this purpose, we redefine the system model accordingly to accommodate new notations.

We consider a distributed system $\mathcal{N}$ consisting of a finite set Π of $n > 1$ processes, $\Pi = \{p_1, p_2, ..., p_n\}$, which communicate by message passing. We assume that all processes have access to local clocks with discrete time events denoted by $t_{p_i} : \{1, 2, 3, ...\}$. A process p_i is assumed to take actions, i.e., either send or receive or both, at the discrete time events of t_{p_i}. The time interval between consecutive events in t_{p_i}, $\forall\, i$ (i.e., for all processes) is assumed to be the same such that it is an upper bound on the propagation delay (t_{pg}) over any link interconnecting any two processes. Processing delays are assumed to be negligible compared to communication delays.

Communication Links. The links interconnecting processes are assumed to be uni-directional uni-cast links. In particular, every pair of processes (p_i, p_j) is connected by two uni-directional links: l_{ij} and l_{ji}. These links exhibit changes in their transmission quality, as the quality of the underlying channels might depend on various propagation conditions. We thus assume that a link l_{ij} has a probability $0 < P_{ij}(t_{p_i}) < 1$ of losing messages at time t_{p_i}. This captures the very idea that a link is not always reliable and can lose messages for an unbounded but finite period. The value of $P_{ij}(t_{p_i})$ can change with time; specifically, at each time $t_{p_i} : \{1, 2, 3, ...\}$, $P_{ij}(t_{p_i})$ may have any value in $(0, 1)$. However, we assume that the value of $P_{ij}(t_{p_i})$ remains constant between consecutive intervals of t_{p_i}. We refer to such links as *probabilistic* links. A probabilistic link thus constitutes an instance of the *fair-loss* link [97], where a message sent by some process p_i infinitely often is received infinitely often.

Faulty Processes. Processes faults are defined as in Section 4.2.

Monitoring Schemes. Given local clocks that are not synchronized (i.e., might be skew), we suggest now, through an example, a small modification to the period at which processes send messages and that at which processes suspect each other, such that $\diamond\mathcal{S}^*$ can be implemented.

Precisely, the algorithm in Theorem 10 can be adapted as follows. First, a process p_i sends messages periodically, by sending messages at every time instant of its local clock t_{p_i}. Initially all processes trust (do not suspect) all other processes. At every odd time event of its local clock such that $t_{p_i} > 1$, process p_i suspects a process p_j if it does not receive a new message since the last odd time event. At every time t_{p_i} if p_i receives a new message from p_j, then p_i trusts p_j. If all messages sent by p_i are not lost for some finite period $t_{p_i} + \Delta t$, then all other processes p_j will not suspect p_i for some period $t_{pj} + \Delta t$. The skew (w.r.t. some global time) between the periods in which p_i is trusted by p_j is $\leq 2t_{pg} \; \forall \; j$ (recall that t_{pg} is the maximum bound on the prorogation delay). In other words, this means that, if $\Delta t > 2t_{pg}$, then is a common duration between all processes during which p_i is not suspected. This duration is at least $\Delta t - 2t_{pg}$.

A similar modification can be applied to Algorithm 1 as well, to have a valid implementation of $\diamond\mathcal{S}^*$.

B.3 $\diamond\mathcal{P}^*$ Algorithms for Decisive Problems

In this section we extend Theorem 16 for the set of algorithms that solve decisive problems using $\diamond P$. First we recall the definitions of $\diamond P$ and $\diamond P^*$.

Failure detector $\diamond\mathcal{P}$ guarantees: (i) *strong completeness* and (ii) *eventual strong accuracy:* There exists a time after which all correct processes are never suspected by any correct process.

Failure detector $\diamond\mathcal{P}^*$ guarantees: (i) *strong bounded completeness* and (ii) *probabilistic eventual strong accuracy:* Consider any finite duration Δt. With positive probability, a all correct processes are not suspected by any correct process for the interval $t_s + \Delta t$, $\forall t_s \in \{1, 2, 3, ...\}$.

Definition 6. *An asynchronous algorithm $\mathcal{A}$ that solves a decisive problem P is said to be $\diamond\mathcal{P}^{\mathcal{N}}$-bounded if it satisfies the following properties:*

1. *$\mathcal{A}$ uses as external blocks only the failure detector $\diamond\mathcal{P}$ and communication primitives implementable in $\mathcal{N}$, such as reliable links and reliable broadcast (see specification in Section 4.2).*

2. *Assume that there exists a point in time t_G when all correct process are never suspected by any correct process. Then $\mathcal{A}$ needs a bounded number of messages to be sent after t_G and until P is solved (i.e., all correct processes decide).*

Theorem 25. *Any algorithm $\mathcal{A}$ that uses $\diamond\mathcal{P}$ to solve a decisive problem P and is $\diamond\mathcal{P}^{\mathcal{N}}$-bounded, solves P in $\mathcal{N}$ guaranteeing termination (i.e., all processes decide) with probability 1, when $\diamond\mathcal{P}^*$ is used instead.*

Proof. We follow similar steps as those adopted in the proof of Theorem 16.

$\diamond\mathcal{P}^*$ provides (in a stronger form) strong completeness as $\diamond\mathcal{P}$. It thus suffices to prove that with respect to $\mathcal{A}$ and with probability 1 the following is satisfied: $\diamond\mathcal{P}^*$ provides the same accuracy as $\diamond\mathcal{P}$.

Assume the existence of an external clock. This clock is not accessible but merely used as a reference to clarify the proof construction. Let t_{start} denote the time instant at which $\mathcal{A}$ starts executing. Using $\diamond\mathcal{P}$ in $\mathcal{A}$ to solve P implies that after t_{start} there is a time when all processes decide and P is solved (see Definition 3). Precisely, after the time when all correct processes are never suspected by any correct process, all correct processes executing $\mathcal{A}$ should exchange a finite bounded number of messages after which P would be solved.

Let M denote the upper bound on the number of messages (be them uni-casts or broadcasts) needed by $\mathcal{A}$ from the time all correct processes are never suspected by any correct process until P is solved. All events that could occur after t_G have a bounded delay (process speeds, crash detection, etc.), except for reliable message transmissions (be them uni-casts or broadcasts). Using communication primitives as the reliable links and reliable broadcast in $\mathcal{N}$, the delay for delivering a single message may be arbitrarily long. However it is possible, with positive probability, that a message gets delivered after a known fixed delay, e.g., x time slots of being sent, at any time instant at which it might be sent (see specifications of reliable transmission Section 4.2). Thus and without loss of generality it is possible (with positive probability) for the M messages to be exchanged within a known fixed duration, say T_M, after which all processes would have decided and thus P would be solved.

Therefore, P can be solved with $\diamond\mathcal{P}^*$ if we prove that $\diamond\mathcal{P}^*$ can with probability 1 provide the following: there is some time instant $t_G \in \{1, 2, 3, ...\}$ after which all correct processes are not suspected by any correct process for the interval $[t_G, T_M]$. From Appendix B.1, the accuracy of $\diamond\mathcal{P}^*$ guarantees that: with positive probability, all correct processes are not suspected by any correct process for the interval $t_s + T_M$, $\forall t_s \in \{1, 2, 3, ...\}$. Since this holds for every $t_s \in \{1, 2, ..., \infty\}$, then $\diamond\mathcal{P}^*$ can with probability 1 provide that: there is some time instant $t_G \in \{1, 2, 3, ...\}$ after which all correct processes are not suspected by any correct process for the interval $[t_G, T_M]$. $\square$

C Proof of Viewsoop's High Probability Guarantees For Large Systems

Lemma 24. *The probabilistic guarantees of ViewSnoop, p_{agree} and $p_{accurate}$, both tend towards 1, as the number of hosts in the system tends towards ∞.*

Proof. Consider a host A and two sets of hosts defined below:

1. π_A: hosts which do not have A in their $local_{suspect}$ list at the beginning of cycle r.
2. $\pi_{\bar{A}}$: hosts which have A in their $local_{suspect}$ at the beginning of r.

Let $\mathcal{C}$ be the set of alive hosts at cycle r. We compute p_{agree} and $p_{accurate}$ when the number of alive hosts is very big, i.e., when $\mathcal{C} \to \infty$.

Hosts in π_A ($\pi_{\bar{A}}$) are those hosts which do not have (have) host A in their $local_{suspect}$ list at the beginning of cycle r. In other words, hosts in π_A ($\pi_{\bar{A}}$) are those hosts which received (did not receive) a heartbeat from host A in cycle $r-1$. Since an infinite number of hosts is alive ($\mathcal{C} \to \infty$), then an infinite number of hosts receive A's heartbeat in cycle $r-1$ and an infinite number of hosts do not receive A's heartbeat in cycle $r-1$, i.e., $|\pi_A|, |\pi_{\bar{A}}| \to \infty$. (An analogy with A sending a message to an infinite number of hosts is flipping a coin an infinite number of times where a head presents a successful message delivery and a tail represents a loss. Flipping a coin an infinite number of times will result in observing an infinite number of heads and an infinite number of tails).

Recall from Section 6.4.3, that disagreement in cycle r occurs if any condition below holds:

1. Host A does not receive any message from all hosts in π_A and at least one host in $\pi_A \bigcup \pi_{\bar{A}}$ receives a message from $A \bigcup \pi_A$.
2. At least one host in $\pi_{\bar{A}}$ does not receive any message from all hosts in $A \bigcup \pi_A$ and:
   (a) Host A hears from at least one host in π_A OR

(b) At least one host in π_A hears from some other host in $A \bigcup \pi_A$.

Condition (1) happens with probability $Prob(1|\pi_A)$:

$$\begin{aligned} Prob(1|\pi_A) =& P(\text{Host A does not receive any message from all hosts in } \pi_A) \\ & \times P(\text{at least one host in } \pi_A \bigcup \pi_{\bar{A}} \text{ receives a message from } A \bigcup \pi_A). \end{aligned}$$

Condition (2) happens with probability $Prob(2|\pi_A)$:

$$\begin{aligned} Prob(2|\pi_A) =& P(\text{At least one host in } \pi_{\bar{A}} \text{ does not receive any message from all hosts in } A \bigcup \pi_A) \\ & \times [P(\text{Host A hears from at least one host in } \pi_A) \\ & + P(\text{At least one host in } \pi_A \text{ hears from at least one other host in } A \bigcup \pi_A)]. \end{aligned}$$

We prove in what follows that the probability of any host in $\pi_{\bar{A}} \bigcup A$ not receiving a message from a host in π_A tends to zero, and hence $Prob(1|\pi_A), Prob(2|\pi_A)$ tend to zero as well. The probability that a host in $\pi_{\bar{A}} \bigcup A$ does not hear any heartbeat from all hosts in π_A, given that $|\pi_A|$ hosts do not have A in their $local_{suspect}$ list at the beginning of cycle r, $P(H|\pi_A)$ is: $P(H|\pi_A) = (1-p)^{n_i \times |\pi_A|}$.

However, since $|\pi_A| \to \infty$ then $P(H|\pi_A) \to 0$. This implies that every host in $\pi_{\bar{A}} \bigcup A$ hears a message from some host in π_A and thus $Prob(disagree_A) \to 0$, i.e., p_{agree} is in $1 - \mathcal{O}((1-p)^{|\pi_A|}), \forall\, p \in]0,1[$.

Following similar reasoning, if host A is correct (does not fail during the entire execution), then the probability that a host H does not declare A as crashed in cycle $r+1$ can be guaranteed if H hears from some host in π_A. This happens with probability: $P_A(H|\pi_A) = 1 - (1-p)^{n_i \times |\pi_A|}$.

Since $\pi_A \to \infty$ when $\mathcal{C} \to \infty$, then $P_A(H|\pi_A) \to 1$ in that case. The probability, $p_{accurate}$, that an excluded host has actually crashed can be restated as the probability of not excluding a correct host. This means $p_{accurate} = P_A(H|\pi_A)$ for some correct host and also tends to 1. $\square$

These results show that *ViewSnoop* implements the agreement and accuracy probabilities of the SYMS with high probability (tend to 1 as the number of hosts increases).

Bibliography

[1] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag, 2nd edition, 2006.

[2] Albert Einstein. On the electrodynamics of moving bodies. *Annalen der Physik*, 4(17), 1905.

[3] Is time travel possible?

[4] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[5] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998.

[6] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[7] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43, 1996.

[8] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2), 1985.

[9] John Eidson, Edward A. Lee, Slobodan Matic, Sanjit A. Seshia, and Jia Zou. Distributed real-time software for cyber-physical systems. *Proceedings of the IEEE*, 100(1), 2012.

[10] P. Verissimo and C. Almeida. Quasi-synchronism: a step away from the traditional fault-tolerant real-time system models. *TCOS*, 1995.

[11] P. Verissimo and A. Casimiro. The timely computing base model and architecture. *IEEE Trans. Comput.*, 51(8), 2002.

[12] Manuel Oriol, Michael Wahler, Robin Steiger, Sascha Stoeter, Egemen Vardar, Heiko Koziolek, and Atul Kumar. Fasa: A scalable software framework for distributed control systems. In *ISARCS*, 2012.

[13] Manuel Oriol, Thomas Gamer, Thijmen de Gooijer, Michael Wahler, and Ettore Ferranti. Fault-tolerant fault tolerance for component-based automation systems. In *ISARCS*, 2013.

[14] M. Bertier, O. Marin, and P. Sens. Performance analysis of a hierarchical failure detector. In *DSN*, pages 635–644, June 2003.

[15] C. Fetzer, M. Raynal, and F. Tronel. An adaptive failure detection protocol. In *PRDC*, 2001.

[16] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *DSN*, 2002.

[17] Naohiro Hayashibara, Xavier Defago, and Takuya Katayama. Two-ways adaptive failure detection with the phi-failure detector. In *ICAC*, 2003.

[18] Indranil Gupta, Tushar D. Chandra, and Germán S. Goldszmidt. On scalable and efficient distributed failure detectors. In *PODC*, 2001.

[19] L. Falai and A. Bondavalli. Experimental evaluation of the qos of failure detectors on wide area network. In *DSN*, 2005.

[20] R. Ceretta Nunes and I. Jansch-Porto. Qos of timeout-based self-tuned failure detectors: the effects of the communication delay predictor and the safety margin. In *DSN*, 2004.

[21] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. *IEEE Trans. Comput.*, 51(5), 2002.

[22] YoungMin Kwon, Kirill Mechitov, and Gul Agha. Design and implementation of a mobile actor platform for wireless sensor networks. In *Concurrent Objects and Beyond*, volume 8665 of *LNCS*. 2014.

[23] M. Cinque, D. Cotroneo, F. Frattini, and S. Russo. Cost-benefit analysis of virtualizing batch systems: Performance-energy-dependability trade-offs. In *UCC*, 2013.

[24] Erwan Le Merrer, Vincent Gramoli, Anne-Marie Kermarrec, Aline Carneiro Viana, and Marin Bertier. Energy aware self-organizing density management in wireless sensor networks. *CoRR*, 2008.

[25] Shinya Kondo, Akimitsu Kanzaki, Takahiro Hara, and Shojiro Nishio. Integration of traffic reduction and sleep scheduling for energy-efficient data gathering in wireless sensor networks. *Comput. Syst. Sci. Eng.*, 28(4), 2013.

[26] Margit Mutschlechner, Bijun Li, Rüdiger Kapitza, and Falko Dressler. Using erasure codes to overcome reliability issues in energy-constrained sensor networks. In *WONS*, 2014.

[27] Tommy Szalapski and Sanjay Kumar Madria. Energy efficient distributed grouping and scaling for real-time data compression in sensor networks. In *IPCCC*, 2014.

[28] Yash V. Pant, Truong X. Nghiem, and Rahul Mangharam. Peak power reduction in hybrid energy systems with limited load forecasts. In *ACC*, 2014.

[29] TheinLai Wong, Tatsuhiro Tsuchiya, and Tohru Kikuno. An energy-efficient broadcast scheme for multihop wireless ad hoc networks using variable-range transmission power. *IEICE*, 90-D(3), 2007.

[30] O. Hohlfeld, R. Geib, and G. Hasslinger. Packet loss in real-time services: Markovian models generating qoe impairments. In *IWQoS*, June 2008.

[31] Gerhard Hasslinger and Oliver Hohlfeld. The gilbert-elliott model for packet loss in real time services on the internet. In *MMB*, March 2008.

[32] Dacfey Dzung and Yvonne-Anne Pignolet. Dynamic selection of wireless/powerline links using markov decision processes. *SmartGridComm*, 2013.

[33] DLC+VIT4IP. D1.1 scenarios and requirements specification. Technical report, 2010. http://www.dlc-vit4ip.org/wb/media/Downloads/$D1.1\_V0.5\_20100910 - team\_$.pdf.

[34] G. Bumiller, L. Lampe, and H. Hrasnica. Power line communication networks for large-scale control and automation systems. *IEEE Communications Magazine*, 48(4), 2010.

[35] H. Kopetz and G. Grunsteidl. Ttp - a time-triggered protocol for fault-tolerant real-time systems. In *FTCS*, 1993.

[36] Elizabeth Latronico and Philip Koopman. Design time reliability analysis of distributed fault tolerance algorithms. In *DSN*, 2005.

[37] Rakesh Rao, Srinivas Akella, and Gokhan Guley. Power line carrier (plc) signal analysis of smart meters for outlier detection. In *SmartGridComm*, 2011.

[38] Jean-Philippe Vasseur and Adam Dunkels. *Interconnecting smart objects with ip: The next internet*. Morgan Kaufmann, 2010.

[39] Nouha Baccour, Anis Koubaa, Claro Noda, Hossein Fotouhi, Mário Alves, Habib Youssef, Marco Zuniga, Carlo Alberto Boano, Kay Römer, Daniele Puccinelli, Thiemo Voigt, and Luca Mottola. *Radio Link Quality Estimation in Low-Power Wireless Networks*. Springer Briefs in Electrical and Computer Eng. 2013.

[40] Dacfey Dzung, Rachid Guerraoui, David Kozhaya, and Yvonne Anne Pignolet. Source routing in time-varing lossy networks. In *NETYS*, pages 200–215, 2015.

[41] Qi Han, Yinghui Zhang, Xiaofeng Chen, Hui Li, and Jiaxiang Quan. Efficient and robust identity-based handoff authentication for eap-based wireless networks. *Concurrency and Computation: Prac. and Exp.*, 26(8), 2014.

[42] L. Marques and A. Casimiro. Fighting uncertainty in highly dynamic wireless sensor networks with probabilistic models. In *SRDS*, 2013.

[43] Tomoko Izumi, Keigo Kinpara, Taisuke Izumi, and Koichi Wada. Space-efficient self-stabilizing counting population protocols on mobile sensor networks. *Theoretical Computer Science*, 552, 2014.

[44] Adam Ji Dou, Song Lin, Vana Kalogeraki, and Dimitrios Gunopulos. Supporting historic queries in sensor networks with flash storage. *Information Systems*, 39, 2014.

[45] Ajay D. Kshemkalyani, Ashfaq A. Khokhar, and Min Shen. Execution and time models for pervasive sensor networks. *IJNC*, 2, 2012.

[46] D. Manivannan, Amrita Jyotiprada, and Navjeet Sandhu. A survey of routing protocols for wireless sensor networks. *IJNGC*, 5(2), 2014.

[47] Link scheduling in wireless sensor networks: Distributed edge-coloring revisited. *JPDC*, 68(8), 2008.

[48] Jérôme Rousselot and Jean-Dominique Decotignie. When ultra low power meets high performance: The wisemac high availability protocol. In *SenSys*, 2010.

[49] Marcel Steine, Cuong Viet Ngo, Ramon Serna Oliver, Marc Geilen, Twan Basten, Gerhard Fohler, and Jean-Dominique Decotignie. Proactive reconfiguration of wireless sensor networks. In *MSWiM*, 2011.

[50] Inigo Urteaga, Na Yu, Nicholas Hubbell, and Qi Han. Aware: Activity aware maintenance of communication structures for wireless sensor networks. *Pervasive and Mobile Computing*, 13, 2014.

[51] Edward J. Sondik. The optimal control of partially observable markov processes over the infinite horizon: Discounted costs. *Math. Oper. Res.*, 26(2), 1978.

[52] A. Laourine and Lang Tong. Betting on gilbert-elliot channels. *IEEE Trans. Wireless Commun.*, 9(2), February 2010.

[53] Sudipto Guha, Kamesh Munagala, and Peng Shi. Approximation algorithms for restless bandit problems. *J. ACM*, 58(1), 2010.

[54] Keqin Liu and Qing Zhao. Indexability of restless bandit problems and optimality of whittle index for dynamic multichannel access. *IEEE Trans. Inf. Theor.*, 56(11), 2010.

[55] A. Aslinger and S. H. Son. Efficient replication control in distributed real-time databases. In *AICCSA*, 2005.

[56] P. Peddi and L. C. DiPippo. A replication strategy for distributed real-time object-oriented databases. In *ISORC 2002*, 2002.

[57] Hengming Zou and Farnam Jahanian. A real-time primary-backup replication service. *IEEE Trans. Parallel Distrib. Syst.*, 10(6), 1999.

[58] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42, 1995.

[59] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Arindam Chakraborty. How fast can a distributed atomic read be? In *PODC*, 2004.

[60] F. C. Freiling, C. Lambertz, and M. Majster-Cederbaum. Modular consensus algorithms for the crash-recovery model. In *PDCAT*, 2009.

[61] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), 1978.

[62] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35, 1988.

[63] JP. Vasseur. Terminology in low power and lossy networks. Technical report, Cisco Systems, Inc, 2013. http://tools.ietf.org/html/draft-ietf-roll-terminology-12.

[64] JeongGil Ko, A. Terzis, S. Dawson-Haggerty, D.E. Culler, J.W. Hui, and P. Levis. Connecting low-power and lossy networks to the internet. *IEEE Communications Magazine*, 49(4), 2011.

[65] Qinqing Zhang and S.A. Kassam. Finite-state markov model for rayleigh fading channels. *IEEE Trans. Commun.*, 47(11), 1999.

[66] Danlu Zhang and K.M. Wasserman. Transmission schemes for time-varying wireless channels with partial state observations. In *INFOCOM*, 2002.

[67] E. O. Elliott. Estimates of error rates for codes on burst-noise channels. *Bell Syst. Tech. J*, 42(5), 1963.

[68] Chiping Tang and Philip K. McKinley. Modeling multicast packet losses in wireless lans. In *MSWIM*, 2003.

[69] Jean pierre Ebert, Andreas Willig, Dr. ing Adam Wolisz, and Tu Berlin. A gilbert-elliot bit error model and the efficient use in packet level simulation. Technical report, 1999.

[70] j. McDougall, J.J. John, Y. Yu, and S.L. Miller. An improved channel model for mobile and ad-hoc network simulations. In *Communications, Internet, and Information Technology*, 2004.

[71] O. Hohlfeld, R. Geib, and G. Hasslinger. Packet loss in real-time services: Markovian models generating qoe impairments. In *IWQoS*, 2008.

[72] Catherine Boutremans, Gianluca Iannaccone, and Christophe Diot. Impact of link failures on voip performance. In *NOSSDAV*, 2002.

[73] J. Poikonen and J. Paavola. Error models for the transport stream packet channel in the dvb-h link layer. In *IEEE International Conf. Commun.*, volume 4, 2006.

[74] J. Kim, G. Bhatia, R. Rajkumar, and M. Jochim. Safer: System-level architecture for failure evasion in real-time applications. In *RTSS*, 2012.

[75] Artur Czumaj, Robert Elsässer, Leszek Gąsieniec, Thomas Sauerwald, and Xin Wang. Fast message dissemination in random geometric networks. *Distributed Computing*, 26(1), 2013.

[76] Elizabeth Latronico and Philip Koopman. Design time reliability analysis of distributed fault tolerance algorithms. In *DSN*, 2005.

[77] G.S. Veronese, M. Correia, A.N. Bessani, Lau Cheuk Lung, and P. Verissimo. Efficient byzantine fault-tolerance. *IEEE Trans. Comput.*, 2013.

[78] Fabíola Greve and Sébastien Tixeuil. Conditions for the solvability of fault-tolerant consensus in asynchronous unknown networks. In *WRAS*, 2010.

[79] James Aspnes, Faith Ellen Fich, and Eric Ruppert. Relationships between broadcast and shared memory in reliable anonymous distributed systems. *Distributed Computing*, 18(3), 2006.

[80] Shuyi Chen, K.R. Joshi, M.A. Hiltunen, R.D. Schlichting, and W.H. Sanders. Using link gradients to predict the impact of network latency on multitier applications. *IEEE/ACM Trans. on Networking*, 19(3), 2011.

[81] Christof Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Trans. Comput*, 52(2), 2003.

[82] Rami Yared, Xavier Défago, Julien Iguchi-Cartigny, and Matthias Wiesmann. Collision prevention platform for a dynamic group of asynchronous cooperative mobile robots. *JNW*, 2, 2007.

[83] B. Garbinato, F. Pedone, and R. Schmidt. An adaptive algorithm for efficient message diffusion in unreliable environments. In *DSN*, 2004.

[84] Camilo Rojas, Damien Piguet, and Jean-Dominique Decotignie. Poster: Single packet link estimation. In *EWSN*, 2016.

[85] Zhen-guo Gao, Klara Nahrstedt, Weidong Xiang, Huiqiang Wang, and Yibing Li. Random network coding based schemes for perfect wireless packet retransmission problems in multiple channel networks. *WIRELESS PERS COMMUN*, 69(4), 2013.

[86] Christos H. Papadimitriou and John N. Tsitsiklis. The complexity of optimal queuing network control. *Math. Oper. Res.*, 24(2), 1999.

[87] N. Nayyar, Yi Gai, and B. Krishnamachari. On a restless multi-armed bandit problem with non-identical arms. *Allerton*, 2011.

[88] Hong Shen Wang and N. Moayeri. Finite-state markov channel-a useful model for radio communication channels. *IEEE Trans. Veh. Technol.*, 44, 1995.

[89] L.N. Kanal and A. R K Sastry. Models for channels with memory and their applications to error control. *Proc. IEEE*, 66(7), 1978.

[90] Hong-Shen Wang and N. Moayeri. Finite-state markov channel-a useful model for radio communication channels. *IEEE Trans. Veh. Technol.*, 44(1), 1995.

[91] Gerhard Hasslinger and Oliver Hohlfeld. The gilbert-elliott model for packet loss in real time services on the internet. In *MMB*, March 2008.

[92] Qing Zhao, B. Krishnamachari, and Keqin Liu. On myopic sensing for multi-channel opportunistic access: structure, optimality, and performance. *IEEE Trans. Wireless Commun.*, 7(12), 2008.

[93] J.L. Ny, Munther Dahleh, and E. Feron. Multi-uav dynamic routing with partial observations using restless bandit allocation indices. *P AMER CONTR CONF*, 2008.

[94] A Caracas, C. Lombriser, Y. A Pignolet, T. Kramp, T. Eirich, R. Adelsberger, and U. Hunkeler. Energy-efficiency through micro-managing communication and optimizing sleep. In *SECON*, 2011.

[95] Sheldon M. Ross. *Introduction to Stochastic Dynamic Programming: Probability and Mathematical*. Academic Press, Orlando, USA, 1983.

[96] Marcos K Aguilera, Wei Chen, and Sam Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. Technical report, 1997.

[97] Rachid Guerraoui, Rui Olivera, and André Schiper. Stubborn communication channels. Technical report, IC, EPFL, 1996.

[98] Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *WDAC*, 1996.

[99] Henrique Moniz, NunoFerreira Neves, Miguel Correia, and Paulo Veríssimo. Randomization can be a healer: Consensus with dynamic omission failures. In *LNCS*, volume 5805. 2009.

[100] H. Moniz, N.F. Neves, and M. Correia. Turquois: Byzantine consensus in wireless ad hoc networks. In *DSN*, June 2010.

[101] L.A. Johnston and V. Krishnamurthy. Opportunistic file transfer over a fading channel: A pomdp search theory formulation with optimal threshold policies. *IEEE Trans. Wireless Commun.*, 5, 2006.

[102] Qing Zhao, B. Krishnamachari, and Keqin Liu. On myopic sensing for multi-channel opportunistic access: structure, optimality, and performance. *IEEE Trans. Wireless Commun.*, 7(12), December 2008.

[103] P. Whittle. Restless bandits: Activity allocation in a changing world. *Journal of Applied Probability*, 25, 1988.

[104] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *PODC*, 2004.

[105] Marcos Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing omega in systems with weak reliability and synchrony assumptions. *LNCS*, 21, 2008.

[106] Achour Mostefaoui, Michel Raynal, and Corentin Travers. Time-free and timer-based assumptions can be combined to obtain eventual leadership. *IEEE Trans. Parallel Distrib. Syst.*, 17, 2006.

[107] A. Mostefaoui, E. Mourgaya, and M. Raynal. Asynchronous implementation of failure detectors. In *DSN*, 2003.

[108] Dahlia Malkhi, Florin Oprea, and Lidong Zhou. Omega meets paxos: Leader election and stability without eventual timely links. In *DISC*, 2005.

[109] Ernesto Jiménez, Sergio Arévalo, and Antonio Fernández. Implementing unreliable failure detectors with unknown membership. *Inf. Process. Lett.*, 100, 2006.

[110] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. In *PODC*, 1992.

[111] Bernadette Charron-Bost, Martin Hutle, and Josef Widder. In search of lost time. *Inf. Process. Lett.*, 110(21), 2010.

[112] Gerhard Hasslinger and Oliver Hohlfeld. The gilbert-elliott model for packet loss in real time services on the internet. In *MMB*, 2008.

[113] Dacfey Dzung, Rachid Guerraoui, David Kozhaya, and Yvonne-Anne Pignolet. Source routing in time-varying lossy networks. In *NETYS*. 2015.

[114] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4), 1985.

[115] Dacfey Dzung, Rachid Guerraoui, David Kozhaya, and Yvonne-Anne Pignolet. To transmit now or not to transmit now. In *SRDS*. 2015.

[116] Fernando Pedone, Rachid Guerraoui, and André Schiper. Exploiting atomic broadcast in replicated databases. In *Euro-Par*, 1998.

[117] Christian Cachin, Rachid. Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer, 2011.

[118] M.C. Kurt, S. Krishnamoorthy, K. Agrawal, and G. Agrawal. Fault-tolerant dynamic task graph scheduling. In *SC*, Nov 2014.

[119] V. Berten, J. Goossens, and E. Jeannot. A probabilistic approach for fault tolerant multiprocessor real-time scheduling. In *IPDPS*, 2006.

[120] Yulu Jia, George Bosilca, Piotr Luszczek, and Jack J. Dongarra. Parallel reduction to hessenberg form with algorithm-based fault tolerance. In *SC*, 2013.

[121] Ricardo Padilha, Enrique Fynn, Robert Soulé, and Fernando Pedone. Callinicos: Robust transactional storage for distributed data structures. In *USENIX ATC*, 2016.

[122] Fernando Pedone and Nicolas Schiper. Byzantine fault-tolerant deferred update replication. *J. Braz. Comp. Soc.*, 18(1), 2012.

[123] Sheng Di, Cho-Li Wang, and Franck Cappello. Adaptive algorithm for minimizing cloud task length with prediction errors. *IEEE Trans. Cloud Comput*, 2, 2014.

[124] Guillaume Aupy, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. Checkpointing algorithms and fault prediction. *J. Parallel Distrib. Comput.*, 74, 2014.

[125] Anne Benoit, Aurélien Cavelan, Yves Robert, and Hongyang Sun. Assessing general-purpose algorithms to cope with fail-stop and silent errors. In *LNCS*. 2015.

[126] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A Debardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. *Int. J. High Perform. Comput. Appl.*, 28(2), 2014.

[127] W. Dweik, M. Abdel-Majeed, and M. Annavaram. Warped-shield: Tolerating hard faults in gpgpus. In *DSN*, 2014.

[128] Joffroy Beauquier, Sylvie DelaËt, Shlomi Dolev, and Sébastien Tixeuil. Transient fault detectors. In *Distributed Computing*, volume 1499 of *LNCS*. 1998.

[129] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43, 1996.

[130] Mikel Larrea, Alberto Lafuente, Iratxe Soraluze, Roberto Cortiñas, and Joachim Wieland. On the implementation of communication-optimal failure detectors. In *LNCS*, volume 4746. 2007.

[131] Alberto Lafuente, Mikel Larrea, Iratxe Soraluze, and Roberto Cortias. Communication-optimal eventually perfect failure detection in partially synchronous systems. *J. Comput. System Sci.*, 81, 2015.

[132] Mikel Larrea, Sergio Arevalo, and Antonio Fernndez. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. *LNCS*, vol.1693, 1999.

[133] Mikel Larrea, Antonio Fernandez Anta, and Sergio Arévalo. Implementing the weakest failure detector for solving the consensus problem. *IJPEDS*, 28, 2013.

[134] M. Larrea, A. Fernandez, and S. Arevalo. Optimal implementation of the weakest failure detector for solving consensus. In *SRDS*, 2000.

[135] Martin Biely, Martin Hutle, LuciaDraque Penso, and Josef Widder. Relating stabilizing timing assumptions to stabilizing failure detectors regarding solvability and efficiency. In *SSS*, volume 4838. 2007.

[136] Alejandro Conrejo, Nancy Lynch, and Srikanth Sastry. Asynchronous failure detectors. In *PODC*, 2012.

[137] N. Santoro and P. Widmayer. Time is not a healer. In *STACS*, 1989.

[138] Ulrich Schmid, Bettina Weiss, and Idit Keidar. Impossibility results and lower bounds for consensus under link failures. *SIAM J. Comput.*, 38(5), 2009.

[139] Iratxe Soraluze, Roberto Cortiñas, Alberto Lafuente, Mikel Larrea, and Felix Freiling. Communication-efficient failure detection and consensus in omission environments. *Inf. Process. Lett.*, (6), 2011.

[140] James Aspnes, Hagit Attiya, and Keren Censor. Combining shared-coin algorithms. *J. Parallel Distrib. Comput.*, 70(3), 2010.

[141] Dan Alistarh, James Aspnes, Valerie King, and Jared Saia. Communication-efficient randomized consensus. In *LNCS*, volume 8784. 2014.

[142] Pierre Fraigniaud, Mika Göös, Amos Korman, Merav Parter, and David Peleg. Randomized distributed decision. *LNCS*, 27, 2014.

[143] Automation services reduce downtime for manufacturers, 2009.

[144] Feng-Li Lian, J. Moyne, and D. Tilbury. Network design consideration for distributed control systems. *IEEE Trans. Control Syst. Technol.*, 10, 2002.

[145] E. Anceaume, A. Fernández, A. Mostefaoui, G. Neiger, and M. Raynal. A necessary and sufficient condition for transforming limited accuracy failure detectors. *J. Comput. Syst. Sci.*, 68, 2004.

[146] Pascal Felber, Xavier Défago, Rachid Guerraoui, and Philipp Oser. Failure detectors as first class objects. In *DOA*, 1999.

[147] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *TPDS*, 21, 2010.

[148] Giuseppe Lipari, Paolo Gai, Michael Trimarchi, Giacomo Guidi, and Paolo Ancilotti. A hierarchical framework for component-based real-time systems. In *LNCS*, volume 3054. 2004.

[149] Giuseppe Lipari, Enrico Bini, and Gerhard Folher. A framework for composing real-time schedulers. *ENTCS*, 82, 2003.

[150] Jean Arlat, Zbigniew Kalbarczyk, and Takashi Nanya. Nanocomputing: Small devices, large dependability challenges. *IEEE Security Privacy*, 10, 2012.

[151] Hyong Sop Shim and Atul Prakash. Tolerating client and communication failures in distributed groupware systems. In *SRDS*, 1998.

[152] Nicola Nostro, Ilaria Matteucci, Andrea Ceccarelli, Felicita Di Giandomenico, Fabio Martinelli, and Andrea Bondavalli. On security countermeasures ranking through threat analysis. In *SAFECOMP*, 2014.

[153] Patricia López Martínez, Laura Barros, and José M. Drake. Design of component-based real-time applications. *J. Syst. Softw.*, 86, 2013.

[154] T. Abdelzaher, A. Shaikh, F. Jahanian, and Kang Shin. Rtcast: lightweight multicast for real-time process groups. In *RTTAS*, 1996.

[155] R. Barbosa, A. Ferreira, and J. Karlsson. Implementation of a flexible membership protocol on a real-time ethernet prototype. In *PRDC*, 2007.

[156] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst.*, 13, 1995.

[157] L.E. Moser and P.M. Melliar-Smith. Probabilistic bounds on message delivery for the totem single-ring protocol. In *RTSS*, 1994.

[158] M. Clegg and K. Marzullo. A low-cost processor group membership protocol for a hard real-time distributed system. In *RTSS*, 1997.

[159] C. Almeida. Handling qos in a dynamic real-time environment. In *WORDS*, 2003.

[160] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Comput. Surv.*, 33, 2001.

[161] R. Friedman and R. van Renesse. Strong and weak virtual synchrony in horus. In *SRDS*, 1996.

[162] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIXATC*, 2010.

[163] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Taming uncertainty in distributed systems with help from the network. In *EuroSys*, 2015.

[164] José Rufino, Paulo Veríssimo, and Guilherme Arroz. Node failure detection and membership in canely. In *DSN*, 2003.

[165] Iratxe Soraluze, Roberto Cortiñas, Alberto Lafuente, Mikel Larrea, and Felix Freiling. Communication-efficient failure detection and consensus in omission environments. *Inf. Process. Lett.*, 111, 2011.

[166] Christof Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Trans. Comput.*, 52(2), 2003.

[167] Dacfey Dzung, Rachid Guerraoui, David Kozhaya, and Yvonne-Anne Pignolet. Never say never - probabilistic and temporal failure detectors. In *IPDPS*, 2016.

[168] Ieee standard profile for use of ieee 1588 precision time protocol in power system applications. *IEEE Std C37.238-2011*, July 2011.

[169] M. Felser. Real-time ethernet - industry prospective. *Proceedings of the IEEE*, 93, 2005.

[170] Roy Friedman, Achour Mostéfaoui, and Michel Raynal. A weakest failure detector-based asynchronous consensus protocol for f<n. *Inf. Process. Lett.*, 90, 2004.

[171] G. Khanna, I. Laguna, F. A. Arshad, and S. Bagchi. Stateful detection in high throughput distributed systems. In *SRDS*, 2007.

[172] J. Balasubramanian, S. Tambe, C. Lu, A. Gokhale, C. Gill, and D. C. Schmidt. Adaptive failover for real-time middleware with passive replication. In *RTAS*, 2009.

[173] M. Serafini, P. Bokor, N. Suri, J. Vinter, A. Ademaj, W. Brandstatter, F. Tagliabo, and J. Koch. Application-level diagnostic and membership protocols for generic time-triggered systems. *TDSC*, 8, 2011.

[174] S. Varadarajan and T. Chiueh. Automatic fault detection and recovery in real time switched ethernet networks. In *INFOCOM*, volume 1, 1999.

[175] Bin Rong, I. Khalil, and Z. Tari. An adaptive membership algorithm for application layer multicast. In *ICNS*, 2006.

[176] J. K. Muppala, S. P. Woolet, and K. S. Trivedi. Real-time systems performance in the presence of failures. *Computer*, 24, 1991.

[177] D. M. Moraes and E. Duarte. A failure detection service for internet-based multi-as distributed systems. In *ICPADS*, 2011.

[178] Fabíola Greve, Pierre Sens, Luciana Arantes, and Véronique Simon. A failure detector for wireless networks with unknown membership. In *Euro-Par*, 2011.

[179] Antonio Fernandez Anta, Sergio Rajsbaum, and Corentin Travers. Brief announcement: Weakest failure detectors via an egg-laying simulation. In *PODC*, 2009.

[180] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[181] M. Vukolic. *Quorum Systems:With Applications to Storage and Consensus*. Morgan and Claypool, 2012.

[182] Tao Qian, F. Mueller, and Yufeng Xin. A real-time distributed hash table. In *RTCSA*, 2014.

[183] Yuan Wei, S. H. Son, J. A. Stankovic, and K. D. Kang. Qos management in replicated real-time databases. In *RTSS*, 2003.

[184] Dacfey Dzung, Rachid Guerraoui, David Kozhaya, and Yvonne-Anne Pignolet. To transmit now or not to transmit now. In *SRDS*, 2015.

[185] Rachid Guerraoui, David Kozhaya, Manuel Oriol, and Yvonne-Anne Pignolet. Who's on board? probabilistic membership for real-time distributed control systems. In *SRDS*, 2016.

[186] A. Timbus, A. Oudalov, and C. N. M. Ho. Islanding detection in smart grids. In *ECCE*, 2010.

[187] Fred B. Schneider, David Gries, and Richard D. Schlichting. Fault-tolerant broadcasts. *Sci. Comput. Program.*, 4, 1984.

[188] B. Galloway and G. P. Hancke. Introduction to industrial control networks. *IEEE Communications Surveys Tutorials*, 15(2), 2013.

[189] M. Dev Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens. A generic, scalable and globally arbitrated memory tree for shared dram access in real-time systems. In *DATE*, 2015.

[190] A. BanaiyanMofrad, N. Dutt, and G. Girão. Modeling and analysis of fault-tolerant distributed memories for networks-on-chip. In *DATE*, 2013.

[191] Simon Schliecker and Rolf Ernst. Real-time performance analysis of multiprocessor systems with shared memory. *ACM Trans. Embed. Comput. Syst.*, 10, 2011.

[192] B. B. Brandenburg and J. H. Anderson. Reader-writer synchronization for shared-memory multiprocessor real-time systems. In *ECRTS*, 2009.

[193] Ming Xiong, K. Ramamritham, J. Haritsa, and J. A. Stankovic. Mirror: a state-conscious concurrency control protocol for replicated real-time databases. In *RTAS*, 1999.

[194] J. Skodzik, P. Danielis, V. Altmann, and D. Timmermann. Hartkad: A hard real-time kademlia approach. In *CCNC*, 2014.

David Kozhaya

Chemin des Clochetons 1
1004 Lausanne, Switzerland
Email address: david.kozhaya@epfl.ch
Mobile: +41(0)789540705

Education

Fall 2011 - Present **Assistant Doctorate in Computer, Communication and Information Sciences, École Polytechnique Fédérale de Lausanne, Switzerland**

- Awarded an EPFL doctoral fellowship
- Expecting to graduate December 2016.
- Thesis title: "Reliable and Real-Time Distributed Abstractions".
- Worked under the supervision of Professor Rachid Guerraoui, distributed programming lab, EPFL and in collaboration with ABB Corporate Research, supervisor Dr. Yvonne-Anne Pignolet.

Fall 2009 - Spring 2011 **M.S. in Computer Engineering, Lebanese American University, Byblos**

- Cumulative GPA 4.00/4.00.
- Thesis: A Parallel Ant Colony Optimization to Globally Optimize Area in High-Level Synthesis.
- Covered topics: *High performance computer architecture, heuristics* and *queuing theory.*

Fall 2004 - Spring 2009 **B.E. in Computer Engineering, Lebanese American University, Byblos**

- Graduated with high distinction, cumulative GPA 3.84/4.00.
- Covered technical topics: *Embedded systems* and *VLSI design.*

Publications

- "Right On Time Distributed Shared Memory", R. Guerraoui, D. Kozhaya, Y. Pignolet (alphabetical order), *RTSS 2016 (To appear)*.
- "Who's On Board? Probabilistic Membership for Real-Time Distributed Control Systems", R. Guerraoui, D. Kozhaya, M. Oriol, Y. Pignolet (alphabetical order), *SRDS 2016 (To appear)*.
- "Never Say Never - Probabilistic and Temporal Failure Detectors", D. Dzung, R. Guerraoui, D. Kozhaya, Y. Pignolet (alphabetical order), *IPDPS 2016*.
- "To Transmit Now Or Not To Transmit Now", D. Dzung, R. Guerraoui, D. Kozhaya, Y. Pignolet (alphabetical order), *SRDS 2015*.
- "Source Routing in Time-Varying Lossy Networks", D. Dzung, R. Guerraoui, D. Kozhaya, Y. Pignolet (alphabetical order), *NETYS 2015*.
- "An Ant Colony Optimization Heuristic To Optimize Prediction of Stability of Object-Oriented Components", H. Harmanani, D. Azar, G. Zgheib, D. Kozhaya, *IRI 2015*.

Professional Experience

September 2012 - Present **Research Collaboration, ABB Corporate Research, Switzerland.**

- Managed 5 research projects, collaborating with 3 ABB scientists and an EPFL professor.
- Identified problems and requirements for futuristic technological advancements in ABB automated control systems for areas such as power and smart grids.
- Planned strategies and devised solutions allowing industrial automation to be 9 times more reliable.
- Implemented and deployed solutions in production systems meeting ABB needs and constraints.
- Established and devised methods allowing celebrated theoretical achievements to meet real network requirements in areas for sensor networks, smart grids and controls systems.

David Kozhaya

Chemin des Clochetons 1
1004 Lausanne, Switzerland
Email address: david.kozhaya@epfl.ch
Mobile: +41(0)789540705

September 2012 - Present **Teaching Assistant, EPFL, Switzerland.**

Managed 15 assistants, guiding them to coach 320 students ensuring a better learning experience.

September 2010 **Financial Analyst, Procter and Gamble *(Internship)***

- Lead a group of 3 people, establishing common ground among people from interdisciplinary fields.
- Managed team discussions and market analysis for solving a real-world business case.
- Devised financial strategies and innovative solutions based on analyzing market needs, allowing the team to win best solution award.

Fall 2009 - Spring 2011 **Part Time Instructor, Electrical and Computer Engineering Department, Lebanese American University, Byblos**

Instructor of Microprocessors lab, Computer Programming lab and Computer Proficiency Course

- Taught 6 classes of 40 students to read, write, and modify code, enhancing their software engineering skills.
- Designed and explained experiments improving students' ability to design systems meeting realistic constraints such as environmental, ethical, safety and sustainability constraints.
- Prepared exams evaluating students' analytic thinking and understanding of engineering topics.
- Coached students to develop good communication skills when presenting engineering topics.

September 2010 - June 2011 **Part Time Instructor, Mounsif National School**

- Instructed the computer literacy subject to 150 students of various ages (grade 5 - grade 12).
- Familiarized students with different computer hardware improving their knowledge of technology.
- Introduced students to working with spreadsheets formulation, databases access and modification, and report/document preparation and formatting.
- Maintained the school computer lab for maximal learning experience, respecting school regulations.

Summer 2008 **Design and Layout Engineer, Fidus - Jounieh, Lebanon**

- Designed and worked on 3 customer based projects involving processes, such as placement and routing, prior to achieving a final printed board layout.
- Analyzed datasheets of components for measurement and manufacturing specifications.
- Assisted full time engineers in creating board footprints and layouts.

2004 - Spring 2009 **Part Time Office Assistant, Lebanese American University, Byblos**

Assisted in orientations and presentations of the financial aid system and regulations.

Technical Skills

- Programming languages: *Java, Assembly 68000, VHDL*
- CAD tools: *Altera Quartus II, CodeWarrier, Magic, Irsim, AutoCAD, Allegro, PCB editor, OrCAD CIS*

Activities

Fall 2014 – Spring 2016 **Les Ateliers du Coeur, Switzerland**

- Handled and monitored a group of 10 children between the ages of 5 and 12.
- Organized and gave English initiation animated courses for children with chronic diseases.

David Kozhaya
Chemin des Clochetons 1
1004 Lausanne, Switzerland
Email address: david.kozhaya@epfl.ch
Mobile: +41(0)789540705

November 2009 **Representing the School of Engineering, Lebanese American University in Third Festival of World Thinkers, Abu Dhabi/Dubai**

- Engaged in multicultural exchange with 500 attendees via discussions of global world problems.
- Enriched from the life testimonies of Nobel laureates from engineering, chemistry, biology, sports.

June 2001 - June 2011 **Scouts Association, Lebanon**

- Lead 200 people of diverse age groups: kids (6-12), teens (14-19), adults (20-27).
- Motivated, initiated and engaged these groups in charitable activities helping unfortunate individuals in their community.
- Managed and organized events for funding charity work.
- Directed lectures and camping events involving awareness programs regarding social issues.

Personal Profile

Languages: English and Arabic spoken and written fluently, French spoken and written fairly.

Interests and Hobbies: Volleyball, camping, biking, swimming, skiing and painting.

References

References are available upon request.