UNIVERSITY OF CALIFORNIA, SAN DIEGO

# Coping with Dependent Failures in Distributed Systems

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in
Computer Science

by

Flavio Junqueira

Committee in charge:

Professor Keith Marzullo, Chair
Professor Geoffrey M. Voelker, Co-chair
Professor Rene Cruz
Professor P. Michael Melliar-Smith
Professor Stefan Savage

2006

TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# ACKNOWLEDGMENTS

I'd like to express my gratitude to a few people, who made all the work presented here possible.

To Marília and Luíz Sérgio de Castro, who supported me at all times and saved no resources to make my life easier despite the long distance that unfortunately separated us all these years.

To my wife Marina Raya for being by my side on the best and on the worst.

To my advisor Keith Marzullo for pushing me through my limits and for always believing in my ideas. I have learned a lot more than simply distributed computing from you.

To my advisor Geoff Voelker for always being there for me. Your dedication to your students is unique, and I certainly would have not survived grad school without your support. I'll never forget that "it's all part of the fun!"

To my committee members Stefan Savage, Rene Cruz, and P. Michael Melliar-Smith for dedicating a bit of their time to evaluate my research work.

To my dear friends Ryan Schmidit, Dmitrii Zagorodnov, Henri Casanova, Renata Teixeira, Graziano Obertelli, Alan Su, Holly Dail, Marvin McNett, David Hutches, Michel Abdalla, Stacy Abdalla, Márcio Faerman, Célio Albuquerque, Juan Camino, Conceição Soares, Marius Rodriguez, Marta Ibanez, Andreu Alibes, Inês Ribeiro, Carlos Garcia, and Carol Soler who made these years in San Diego very pleasant.

To Ranjita Bhagwan, Alejandro Hevia, Xianan Zhang, Luca Telloli, Yanhua Mao, and David Oppenheimer for working with me. Karen Bhatia and Sandeep Chandra for all the support with the Geon measurement study.

To the GTA folks Otto, Rezende, Luish, Rubi, and Baiano for all the support from the homeland.

To my labmates John Bellardo, Alper Mizrak, Jeannie Albrecht, Kiran Tati, and Ramana Kompella for all the existential discussions, which happened to be the most useful ones.

To the CSE professors. In particular to Amin Vahdat, Alex Snoeren, Joe Pasquale, Mihir Bellare, and Russell Implagliazzo for helpful discussions.

To the CSE Staff. In particular to Julie Conner, for making my relationship with the UCSD bureaucracy a lot easier, to Michelle Panik for being in general so helpful, and to Dave Wargo, for solving all technical problems presented to him as fast as one can.

To CAPES for funding my research.

## VITA

| | |
|---|---|
| 1997 | B.S. in Electrical Engineering, Federal University of Rio de Janeiro (UFRJ) |
| 1999 | M.S. in Electrical Engineering, Federal University of Rio de Janeiro (UFRJ) |
| 1999-2006 | Teaching Assistant and Research Assistant, Department of Computer Science and Engineering |
| 2003 | Intern at Microsoft Research, Cambridge, UK |
| 2006 | Ph.D., University of California, San Diego |

## PUBLICATIONS

F. Junqueira, and K. Marzullo, "*Coterie availability in sites*", in the Proceedings of the 19th International Conference in Distributed Computing (DISC'05), LNCS 3724, pp. 3–17, September 2005.

F. Junqueira, and K. Marzullo, "*Replication predicates for dependent-failure algorithms*", in the Proceedings of the Euro-Par'05 Conference, LNCS 3648, pp. 617–632, August 2005.

F. Junqueira, and K. Marzullo, "*The virtue of dependent failures in multi-site systems*", in the Proceedings of the 1st Workshop on Hot Topics in System Dependability (HotDep'05), pp. 242–247, June 2005.

F. Junqueira, R. Bhagwan, A. Hevia, K. Marzullo, and G. M. Voelker, "*Surviving Internet Catastrophes*", in the Proceedings of the USENIX Technical Conference, pp. 45–60, April 2005.

F. Junqueira, R. Bhagwan, S. Savage, K. Marzullo, and G. M. Voelker, "*The Phoenix Recovery System: Recovering from the ashes of an Internet catastrophe*", in the Proceedings of the IX Workshop on Hot Topics in Operating Systems (HotOS-IX), pp. 73–78, May 2003.

F. Junqueira, and K. Marzullo, "*Synchronous consensus for dependent process failures*", in the Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03), pp. 274–283, May 2003.

F. Junqueira, and K. Marzullo, "*Designing algorithms for dependent process failures*", in the Proceedings of the 1st International Workshop on Future Directions in Distributed Computing (FuDiCo), LNCS 2584, pp. 24–28, June 2002.

ABSTRACT OF THE DISSERTATION

# Coping with Dependent Failures in Distributed Systems

by

Flavio Junqueira

Doctor of Philosophy in Computer Science

University of California, San Diego, 2006

Professor Keith Marzullo, Chair

Professor Geoffrey M. Voelker, Co-chair

Traditionally, fault-tolerant systems assume that failures are independent, often expressed as a threshold on the number of failures. This assumption, however, is increasingly unrealistic for current distributed systems. This dissertation presents a model of dependent failures based on two abstractions: *cores* and *survivor sets*. Cores are minimal sets of processes such that at least one process is correct in every execution. Survivor sets are minimal sets of processes such that in every execution at least one survivor set contains only correct processes. This model has both theoretical and practical use. Theoretically, this model enables more flexible designs of distributed algorithms. This flexibility often improves efficiency; for example, it enables systems to meet reliability goals with fewer processes. Practically, I apply these replication techniques for multi-site systems that tolerate site failures, and for cooperative backup systems that tolerate Internet attacks.

Multi-site systems, such as grid systems, experience failures that render all the processes in a site unavailable. These failures are often caused by failures of shared resources. Under such site failures, previous availability results on replication techniques, such as quorum systems, no longer necessarily hold. Using a failure model that explicitly captures dependent failures, I develop techniques for selecting replicas and forming quorums that do have optimal availability in multi-site systems.

Internet attacks by worms and viruses can infect a large number of hosts, resulting in catastrophic failures. To cope with such attacks, I propose a new approach for designing distributed systems called *informed replication*. Informed replication uses a model of correlated failures to exploit software diversity. To demonstrate this approach, I design and evaluate a cooperative backup service called *The Phoenix Recovery System*. Phoenix uses heuristics to select replicas that provide excellent reliability guarantees, result in low degree of replication, limit the storage burden on each host in the system, and lend themselves to a distributed implementation.

Incorporating dependent failures into the design of systems and algorithms therefore results in important advantages such as more efficient algorithms, higher availability and efficient utilization of resources. As systems increase in size and extent, these advantages will become increasingly more effective.

# Chapter 1

# Introduction

Incorporating dependent failures into the design of distributed algorithms has both theoretical and practical benefits. Theoretically, it enables algorithms that use fewer replicas and complete executions in fewer rounds of message exchange. In practice, it enables higher available fault tolerant systems and replicated systems that use resources more efficiently. This dissertation presents a new model of dependent failures for the design of fault-tolerant distributed systems and demonstrates the theoretical benefits of this model by developing new algorithms that improve upon previous fundamental results, as well as its practical benefits by applying it to wide-area distributed systems.

Computer systems are ubiquitous. Individually, people use computers at home to accomplish a variety of tasks, ranging from text editing through bank account access. Users also largely rely on computers at work to maintain and generate information, such as e-mail messages and electronic documents. At the enterprise level, corporations depend upon large data centers to store sensitive data and accomplish mission-critical tasks. Banks require a secure computing infrastructure to register transactions and maintain customer information. Internet companies rely upon front-end servers connected to the Internet to deliver the best user experience and on back-end servers to accomplish business tasks, such as keeping the inventory up to date and charging customers for purchases. In science, researchers rely upon distributed resources to conduct experiments that require a vast amount of computational power, such as large-scale simulations or visualization of large data sets. Traditionally, mission-critical applications, such as aircraft control, control of industrial processes, and control of nuclear plants, have also depended upon computer systems as they enable faster and more reliable responses in critical situations.

Perhaps most important is that this reliance upon computer systems has been increasing over time. Figures from the U.S. Census Bureau show how the use of computers has been increasing. First, 56% of the households in the United States had a computer in 2001, compared to 61% in 2003 [Bura]. Second, still in 2003, 56% of the workforce use a computer in the work environment, and approximately 42% have access to the Internet [Bura]. Although these numbers do not show the overall use of computer systems in companies, it is clear that a significant fraction of these users depend on a computing infrastructure composed of servers and network interconnection to accomplish their job-related tasks. Third, *e*-commerce sales increased 24% in 2005 compared to the previous year [Burc], and the overall number of computers manufacturers shipped has increased by 1.5 millions between 2003 and 2004 [Burb].

Because we increasingly rely upon computer systems, a computer failure can wreak havoc: it can cause a small disruption of our daily routines when our desktop is not working, or it can have a more serious impact that can be either life threatening or business threatening in the case of mission-critical systems. A survey conducted by a market research company in 2005 shows that downtime due to failures costs millions of dollars yearly to large companies across the United States [Res], consequently adding to the cost of ownership [Pat02]. Building systems at all levels that are able to operate correctly despite failures is therefore an important mission for designs of current and future computer systems.

Fault tolerance in computer systems is a field that has been studied continuously for the past three decades. Its applications, however, have been so far mostly to mission-critical, highly-specialized systems (*e.g.*, air traffic control and stock exchange [Bir99]). There are several techniques that have been used to cope with failures in computer systems. At a high level, these techniques can be classified as building more robust hardware components, detecting/recovering from failures, and masking failures.

Computer systems can achieve higher reliability by using more reliable components, *i.e.*, components that have longer time between failures. These components can be more robust by using more reliable parts or more suitable materials. For example, computers can be built with more reliable disks and memory, or even be placed inside steel cases for physical protection as well as protection against natural disasters such as fires and floods.

When using more reliable components is not an option, it is still possible to implement highly-reliable systems using less reliable components. At the heart of these techniques is a common approach called *replication*. Replication consists of using a set of replicas of a particular component. For example, instead of using one server for a particular service, a system can use multiple copies of the same service. A detection/recovery approach, such as primary-backup, assumes that at most one replica at a time has the authority to reply to client requests. For example, in a replicated file system implemented with this technique, the primary replica handles reads and writes issued to the file system. If the primary fails, then there is a detection mechanism responsible for indicating this failure, and for triggering the election of a new primary replica. Once a new replica is elected, this replica takes over the role of a primary. Transferring the primary responsibilities from a faulty primary to a working one constitutes the recovery step.

A different approach, called masking, consists in making failures transparent to clients. Using the same replicated file system example, a masking technique provides seamless access to files and directories despite failures, whereas in the primary-backup approach there is a detection step and a recovery step when the primary fails. A typical masking technique is *quorum update*. Given a set of replicas, operations such as reads and writes to the file system execute concurrently in a *quorum* of replicas. Thus, a system is available as long as at least one quorum of replicas is available for a particular operation, and the overall availability of the system depends upon the choices for quorums. There are constraints to these choices, though. Quorums, for example, must intersect in order to preserve the consistency of the application state.

Replication, however, is not necessary only when coping with failures of a single computer. Several of the computing infrastructures mentioned previously are in fact distributed. Companies have offices spread across the globe, each of these offices comprising a small part of the computational capacity owned by the company. Banks have several branches in one or more countries. Grid systems used by scientists to conduct experiments have resources such as CPUs and storage nodes spread across a number of sites in different geographic locations [FK99]. In all these cases, it is often necessary to coordinate the actions of two or more computers to guarantee the integrity and the consistency of application state (collection of data objects used by the application).

There is a number of well-known, fundamental problems in distributed computing related to the coordination of multiple computers: agreement upon one single value (Consensus [Mul95a]), reliable dissemination of messages (Reliable Broadcast [Mul95a]), and election of one single replica to perform a particular task on behalf of the system (Leader Election [AW98b]). Depending on the particular problem we are trying to solve, there exists some constraint on the amount of replication of a particular component. In distributed systems, a computing unit is modeled as a process, and a process hence is the abstraction used for a computer. When considering process replication, there are often constraints on the number of processes of the form $n > k \cdot t$, where $n$ is the number of processes, $t$ is a threshold on the number of process failures and $k$ is a multiplying factor. There are many examples in the literature of problems that have such requirements. One example of particular importance is the consensus problem [Mul95a]. This problem is important, for example, in the replicated file system example discussed before. To guarantee eventual consistency across the state of all replicas, it is necessary and sufficient to have all correct replicas applying the same set of operations and in the same order, assuming the replicas execute only deterministic operations [Sch90]. Consensus has an important role in the decision of the sequence of operations to execute on replicas. Informally, consensus consists of having a set of processes, which communicate by exchanging messages, deciding upon the same value, where each process proposes one possible decision value. Assuming that processes can fail arbitrarily, there is no solution to this problem if $n \leq 3 \cdot t$. Thus, the problem requires $n > 3 \cdot t$. We discuss the consensus problem further in Chapter 3.

Modeling failures of processes using a single threshold $t$ is attractive as it condenses all possibilities for subsets of faulty processes in a single value. We call this model the *threshold model*. The threshold model captures well process failures that are independent and identically distributed. That is, failures have the same probability distribution and have joint probabilities equal to the multiplication of the individual probabilities. In such cases, for any group of processes of size $s$, the probability of having all $s$ processes simultaneously faulty is some constant. To determine the threshold in a particular scenario, it is simply a matter of selecting a value, perhaps using probabilities, that is more appropriate for a particular application.

Failures in practice, however, are often not independent. Multiple racks within a data center fail due to shared resources (*e.g.*, storage devices [BWWG02]), operational errors, and physical problems such overheating [AWL05], thus bringing all of its resources down and rendering them unavailable. Large-scale Internet attacks rely upon the existence of vulnerabilities shared by a large number of hosts. Internet pathogens that exploit such vulnerabilities may compromise a large number of hosts, perhaps causing them to behave arbitrarily [MSB02].

Reliability engineers have approached dependent failures in the past when designing safety-critical systems such as the ones for nuclear plants [Mos91]. They have used techniques such as fault trees to model the dependencies among components of a system [DBB93]. Fault trees model in detail the components of systems and their relationships.

In distributed systems, however, coping with dependent failures is not a well-studied problem. There are three main reasons for this state of affairs. First, both algorithm and system designers often believe that dealing with dependent failures is a daunting task, and using a threshold not only simplifies substantially the design but it is also a conservative assumption. Although techniques such as fault trees can be used, they are often not sufficiently abstract when designing algorithms. We address this concern by proposing a new model that can be used both in the design of algorithms and systems, and show how to use it. Second, sources of failure correlation are not well abstracted, and consequently integrating dependent failures into designs has been overlooked. We discuss examples of real systems in which we identify the sources of failure correlation and incorporate this information into a failure model that enables the development of mechanisms that achieve the goals for the system. Third, there is a common belief that, if failures are correlated, then the failure of one component implies the failure of all the other components. This is true when, for example, all processes run the same software, as a software fault may cause the failure of all the processes. In general, faults that affect the whole system are not covered in our work. Although important, reducing the risk of software faults in deployed software is a study area of software engineering, and it is out of the scope of this work.

**Outline and Summary of Contributions.** When discussing fault tolerance, it is often important to distinguish faults, errors, and failures. According to the IEEE Standard Glossary of Software Engineering Terminology [Ins90], a *fault* is either a software or a hardware defect. An *error* is an incorrect step, process, or data definition. A *failure* is a deviation from the expected correct behavior. As an example, if a programmer introduces an invalid set of instructions, and the execution of these instructions causes a computer to crash, then the introduction of these instructions into the program is the fault, executing them is the error, and crashing the computer is the failure. When testing software, for example, it is very important to distinguish between failures and faults as the failures are the observable events, but to fix software problems it is necessary to map them to faults. In this work, however, we only consider the consequences of faults, *i.e.*, we only consider failures.

We first introduce a new abstract model of dependent failures that enables the design of algorithms and systems in which failures are dependent (Chapter 2). This model is based on two abstractions that correspond to more general versions of two common objects found in the design of algorithms for distributed systems: subsets of processes of size $t+1$ and subsets of processes of size $n-t$. Cores generalize subsets of processes of size $t+1$, subsets in which at least one process is correct, and survivor sets generalize subsets of processes of size $n-t$, subsets that contain only correct processes in some execution. This model is simple, yet it is more expressive to describe subsets of faulty processes, thus enabling the design of a number of interesting algorithms for dependent failures.

Using this model, we derive a number of new results related to consensus in Chapter 3. We consider synchronous systems as well as asynchronous systems. Synchronous systems are systems in which the clocks of processes are synchronized within some bound. This enables non-faulty processes to proceed in rounds of message exchange. For synchronous systems, this chapter discusses the lower bound on the number of rounds to solve consensus, and presents properties that are necessary and sufficient to solve consensus in our new model along with optimal algorithms with respect to process replication. An interesting result out of the lower bound on the number of rounds is the difference on the minimum number of rounds when different failure modes are considered (*e.g.*, crash vs. arbitrary) for some systems. This result is surprising as this is not the case for the threshold model. Thus, considering dependent failures enables the design of algorithms that are theoretically more efficient. Asynchronous systems are systems in which message delay, processor speed, and clock drift are unbounded. These assumptions preclude the use of synchronous rounds of message exchange as for synchronous systems. For asynchronous systems, this chapter explores the difficulties of reaching consensus and solutions. More specifically, we present properties on process replication that are necessary and sufficient to solve consensus with dependent failures in asynchronous systems as well as algorithms that solve this problem and are optimal with respect to process replication.

In studying algorithms for consensus in both types of systems, we also introduce new forms of requirements on process replication based on set properties: *Crash Partition*, *Crash Intersection*, *Byzantine Partition*, and *Byzantine Intersection*. Partition properties express constraints on how the set of processes can be partitioned to enable solutions. Similarly, intersection properties express constraints on the intersection of survivor sets. These partition and intersection properties constitute more general forms of requirements of the forms: $n > t$, $n > 2 \cdot t$, and $n > 3 \cdot t$.

In Chapter 4, we study partition and intersection properties that generalize requirements under the threshold model of the form $n > k \cdot t$, for integer $k > 1$. We also study a set of unusual partition and intersection properties: ($k$,$k-1$)-Partition and ($k$,$k-1$)-Intersection, $k > 1$. These properties have more general forms of $n > \lfloor k \cdot t/(k-1) \rfloor$. This requirement appears, for example, in the context of primary-backup protocols for receive-omission failures of processes, $k = 3$. Informally, in the primary-backup approach to replication, it is necessary to elect a new primary every time the current primary fails. The lower bound on process replication for this problem in the receive-omission failure model has been known for many years to be $n > \lfloor 3 \cdot t/2 \rfloor$. In Chapter 4, we review this proof and we present for the first time an algorithm for this problem that uses (3,2)-Intersection. A proof of correctness for this algorithm is in Appendix A.

In the last two chapters of this dissertation, we present techniques that take dependent failures into account not only to improve fault-tolerance in practical applications, but also to use resources more efficiently by:

1. Achieving higher availability with the same set of resources (Chapter 5);
2. Reducing the storage overhead when replicating data (Chapter 6).

With the proliferation of projects related to Grid Systems and large companies relying upon distributed infrastructures, there is an increasing interest in systems composed of multiple sites, or *multi-site systems*. In addition to process failures, multi-site systems experience dependent failures in the form of *site failures*: failures that bring all the nodes in the faulty site down. In Chapter 5, we show that in scenarios in which site failures can happen, quorums formed of majorities are not necessarily optimal with respect to availability, a result that has been known for two decades to hold when failures are independent and identically distributed. We then provide a few constructions that are optimal with respect to one metric that consists in counting the number of covered survivor sets, and present results of a small experiment consisting of running Paxos on PlanetLab (Paxos uses quorums implicitly). The results of the PlanetLab experiment show that our construction is more available than a majority system that uses the same number of hosts when there are failures. This conclusion holds because there are more cases in which no quorums are available for the majority system we considered.

In Chapter 6, we propose a technique called *informed replication*. Informed replication uses attributes of processes and cores to determine how to choose replica sets in a system. Heuristics based on this technique enable hosts to survive large-scale attacks, such as the outbreak of a worm or a virus, with a modest amount of replication compared to "uninformed" techniques that do not take the heterogeneity of hosts into consideration. These heuristics are interesting because they provide excellent reliability guarantees, result in low degree of replication, limit the storage burden on each host in the system, and lend themselves to a fully distributed implementation. We used one of these heuristics to build The Phoenix Recovery System, a system designed to tolerate large-scale attacks of worms and viruses. Our evaluation on PlanetLab (a distributed testbed) shows that our results hold in practice: hosts have a high probability of recovering data after a large-scale Internet attack, using a small number of replicas (on average less than two), and committing a modest amount of resources to the system (each host holding replicas for at most three other hosts).

In summary, this dissertation presents several results that demonstrate the benefits, both theoretical and practical, of incorporating dependent failure information into the failure model when designing fault-tolerant systems. Chapter 7 discuss in more detail the conclusions on results of this work.

# Chapter 2

# A new model of dependent failures

In this chapter, we present a new model for systems in which failures are not necessarily independent and identically distributed. This model is useful when designing algorithms for dependent failures, as we illustrate in the next chapter with algorithms for the consensus problem. We first introduce two common abstractions in models of distributed systems: processes and channels. Processes and channels abstract real world components such as computers and networks. We then present the two abstractions that compose our proposed model, which we call *cores* and *survivor sets*. Cores and survivor sets are subsets of processes that characterize possible failures in a system. Finally, in the last section of this chapter, we discuss related work on failure characterization.

## 2.1 Processes and channels

Distributed systems are often comprised of multiple computers interconnected by a message-passing network. To abstract the details of real world computers and networks, computers are often modeled as processes (or processors) and networks as channels that interconnect the processes. A process models a unit capable of computing, and it proceeds in atomic steps. Channels simply store and forward messages.

Henceforth, we assume that a system is a set $\Pi$ of processes pairwise interconnected by channels. We consider systems in which processes can be faulty. We will leave the possible failure modes unspecified, however, as they depend on the particular problem we want to solve and they are not required for the exposition of our model. Throughout the following chapters, we assume that all of the channels connecting processes are *quasi-reliable* [ACT97]:

- If a correct process $p_i$ sends a message $m$ to a correct process $p_j$, then $p_j$ eventually receives $m$;
- Messages are not duplicated;
- A process $p_j$ receives a message $m$ from a process $p_i$ only if $p_i$ has sent $m$ to $p_j$.

In the remainder of this chapter, we only consider faulty processes.

## 2.2 Cores and survivor sets

Typically, system models use a single threshold $t$ to characterize failure scenarios. Under such a failure model, the value $t$ constitutes an upper limit on the number of processes that can fail in any execution. That is, it makes no distinction among subsets of processes, and any subset of $t$ processes is subject to failure. We call this model the *threshold model*. This model is appropriate when failures are independent and identically distributed because in such cases all subsets of processes of the same size have the same failure probability. In practice, different subsets of processes of the same size might have different probabilities of failure. This implies that using just a threshold may cause algorithms either to use more processes than necessary, or perhaps to violate the assumption on the number of faulty processes if failures are assumed to be independent but they are not. Violating the assumption on the number of processes may cause an algorithm to execute incorrectly.

Instead of using a simple threshold, we characterize failure scenarios with *cores* and *survivor sets* [JM03a, JM03b, JM05a]. A core is a minimal subset of processes such that, in every execution, there is at least one process in the core that is not faulty. A core generalizes the idea of a subset of processes of size $t+1$ in the threshold model. A *survivor set* is a minimal set of processes such that there is an execution in which none of the processes in the set are faulty. A survivor set generalizes the idea of a subset of processes of size $n-t$ in the threshold model, where $n$ is the total number of processes. Cores and survivor sets are duals of each other: from the set of cores, the set of survivor sets is the set of all minimal subsets of processes that intersect every core. Similarly, from the set of survivor sets, the set of cores is the set of all minimal sets of processes that intersect every survivor set.

To define cores and survivor sets more formally, we first define executions. Informally, an execution encompasses all the operations executed by a distributed algorithm as well as environmental behavior such as process failures and message delays. An execution $E$ is a tuple $\langle \mathit{Init}, \mathit{Steps}, \mathit{Time}, \mathit{Faulty} \rangle$, where *Init* is a mapping from process to initial state, *Steps* is a set of steps of processes, *Time* is a mapping from steps in *Steps* to positive integers, and *Faulty* is a mapping from step to the subset of processes. The initial state of a process depends upon the problem we are solving. For example, for the consensus problem, the initial state is a single value in a set of possible initial values. The second component *Steps* is a set of steps of processes. The definition of a step includes sending one or more messages, receiving one or more messages, and executing local operations. We refine the definition of a step throughout the following chapters, and for now we leave it undefined. We also use $\mathit{StepsOf}(i)$ to denote the subset of *Steps* containing the steps of process $p_i$. The mapping *Time* maps steps to integers that represent global time generated by some external device. Although processes do not have access to such a device, we introduce this feature to enable discussions on the order of steps. Finally, *Faulty* is a mapping from step to subset of processes, where the subset comprises the processes that are faulty in the step. That is, given a step $s$ of some execution $E$, $\mathit{Faulty}(s)$ evaluates to the set of processes that are faulty in step $s$. In most cases, we assume that processes do not recover, and consequently *Faulty* is monotonically increasing. In Chapter 5, however, we assume that processes can recover.

We are now ready to define cores and survivor sets more formally. Consider a system with a set $\Pi$ of processes. Let $\mathcal{E}$ be all executions of a distributed algorithm. We then have that:

**Definition 2.2.1** A subset $C \subseteq \Pi$ is a core if and only if:

1. $\forall \langle \mathit{Init}, \mathit{Steps}, \mathit{Time}, \mathit{Faulty} \rangle \in \mathcal{E}$: $\forall s \in \mathit{Steps}$: $\Pi \setminus \mathit{Faulty}(s) \cap C \neq \emptyset$;

2. $\forall p_i \in C$: $\exists \langle \mathit{Init}, \mathit{Steps}, \mathit{Time}, \mathit{Faulty} \rangle \in \mathcal{E}$: $\exists s \in \mathit{Steps}$: $C \setminus \{p_i\} \cap \Pi \setminus \mathit{Faulty}(s) = \emptyset$.

**Definition 2.2.2** A subset $S \subseteq \Pi$ is a survivor set if and only if:

- $\exists \langle \mathit{Init}, \mathit{Steps}, \mathit{Time}, \mathit{Faulty} \rangle \in \mathcal{E}$: $\forall s \in \mathit{Steps}$: $\Pi \setminus \mathit{Faulty}(s) = S$;

- $\forall p_i \in S$: $\forall \langle \mathit{Init}, \mathit{Steps}, \mathit{Time}, \mathit{Faulty} \rangle \in \mathcal{E}$: $\forall s \in \mathit{Steps}$: $\Pi \setminus \mathit{Faulty}(s) \not\subseteq S \setminus \{p_i\}$.

If *Faulty* is monotonically increasing in all executions, then we can define cores and survivor sets more concisely [JM05a]. Let *Correct*$(E)$ be the set of processes that are not faulty in an execution $E = \langle \mathit{Init}, \mathit{Steps}, \mathit{Time}, \mathit{Faulty} \rangle \in \mathcal{E}$. That is, *Correct*$(E)$= $\bigcup_{s \in \mathit{Steps}} \mathit{Faulty}(s)$, $E = \langle \mathit{Init}, \mathit{Steps}, \mathit{Time}, \mathit{Faulty} \rangle \in \mathcal{E}$. We then have the following definitions:

**Definition 2.2.3** A subset $C \subseteq \Pi$ is a core if and only if:

1. $\forall E \in \mathcal{E}$: $\mathit{Correct}(E) \cap C \neq \emptyset$;

2. $\forall p_i \in C$: $\exists E \in \mathcal{E}$: $C \setminus \{p_i\} \cap \mathit{Correct}(E) = \emptyset$.

**Definition 2.2.4** A subset $S \subseteq \Pi$ is a survivor set if and only if:

- $\exists E \in \mathcal{E}$: $\mathit{Correct}(E) = S$;

- $\forall p_i \in S$: $\forall E \in \mathcal{E}$: $\mathit{Correct}(E) \not\subseteq S \setminus \{p_i\}$.

In [JM03a, JM03b], we defined cores and survivor sets using probabilities. The alternative definition based on executions given above is more convenient when discussing algorithms. In practice, one can use failure probabilities and a target reliability (or availability) to compute the sets of faulty processes that can be tolerated, and these sets determine the possible failures of an execution. However, one does not have to determine sets of faulty processes to be tolerated on probabilities. As our example in Section 2.3 shows, determining sets of faulty processes can be based on a combination of quantitative and qualitative information.

We use the term *system profile* to denote a description of the tolerated failure scenarios. In the threshold model, a system profile is a pair $\langle \Pi, t \rangle$, and means that any subset of $t$ processes in $\Pi$ can be faulty. In our dependent failure model, the system profile is a triple $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$, where $\mathcal{C}_\Pi$ is the set of cores and $\mathcal{S}_\Pi$ is the set of survivor sets.[1] We assume that each process is a member of at least one survivor set (otherwise, that process can be faulty in each execution, and thus may best be ignored by the other processes), and that no process is a member of every survivor set (otherwise, that process is never faulty, which enables trivial solutions to many problems). The threshold system profile $\langle \Pi, t \rangle$ is equivalent to the profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ where $\mathcal{C}_\Pi$ is all subsets of $\Pi$ of size $t + 1$ and $\mathcal{S}_\Pi$ is all subsets of $\Pi$ of size $|\Pi| - t$.

We treat the *kind* of failure—crash, omission, arbitrary, etc.—as a separate part of the failure model. The kind of failure is important both in the design of algorithms and in the derivation of lower bounds. In some situations, such as with hybrid failure models (for example, the model in [Chr90]), separating the kind of failures from the system profile would be complex. In general, we do not assume any particular kind of failure, but we do so when discussing specific problems.

Determining the system profile requires one to consider the possible causes of process failures. For example, a process running on a particular processor fails if the processor hardware fails (crash failure). As another example, if one has as a concern software faults (bugs), then a process can fail if there is an error in one of the software packages it depends upon and the system executes the erroneous instructions (which can

[1]Since $\mathcal{C}_\Pi$ and $\mathcal{S}_\Pi$ can be computed from each other, in fact the system profile could contain only one of these two sets. We include both for convenience.

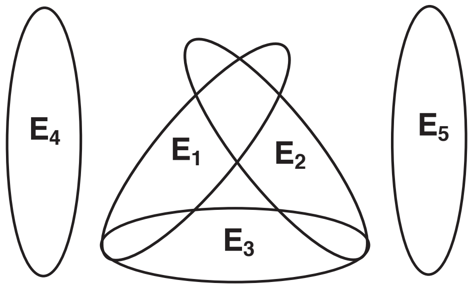

**Figure 2.1:** Sets of executions. $E_i$ is the set of executions that have at least one faulty replica running version $v_i$. The executions in an intersection between two sets are executions in which a software fault causes processes running two different versions to fail.

result in an arbitrary failure) [GS91].

## 2.3 Determining a system profile

We now give an example of a system profile that we derive using qualitative information. In the work by Castro *et al.* [CRL03], the authors observe that independent software development ideally produces disjoint sets of software faults. This observation is the basic idea of $n$-version programming, whose goal is to render software failures independent. Of course, there is still a marginal probability that two or more replicas fail in the same execution, but the assumption is that this probability is small enough that it can be ignored.

Suppose we want to implement a fault-tolerant service using the state machine approach [CL02, Sch90], and we are concerned about arbitrary failures arising from software faults. Moreover, we want to leverage the existence of multiple standalone implementations of this service we are interested in, as in BASE [CRL03]. In [CRL03], Castro *et al.* propose a replicated NFS server using existing standalone implementations.

Using this approach, each of our replicas has two components: a standalone implementation and a replica coordination component that implements a distributed consensus algorithm. Now suppose that there are five standalone versions available for use: $v_1$ through $v_5$. Looking more carefully at the history of these versions, we discover that two of them reuse code from previous versions: $v_2$ reuses a set $X$ of modules from $v_1$, and $v_3$ reuses modules $Y$ from $v_1$ and $Z$ from $v_2$ (a module is a collection of functions and data structures). We also assume that $X$, $Y$, and $Z$ are disjoint sets, and that $v_4$ and $v_5$ were developed independently.

Assuming that every software module potentially has software faults, we have: 1) faults in the modules in $X$ can affect both $v_1$ and $v_2$; 2) faults in the modules in $Y$ can affect both $v_1$ and $v_3$; 3) faults in the modules in $Z$ can affect both $v_2$ and $v_3$.

Consider a system in which there is at least one replica running each of the five versions. Let $E_i$ be the set of executions in which at least one replica is faulty because of a fault in the version $v_i$. These sets of executions are related to each other as shown in Figure 2.1.

Assuming one replica for each version, and assuming that at most one software fault can be exercised in an execution, we have the profile of Example 2.3.1. This system has sufficient replication to solve the consensus problem in a synchronous system with arbitrarily faulty processes and no digital signatures (Section 3.3), and asynchronous consensus assuming a mute failure detector (Section 3.4). The amount of replication is also sufficient to implement a fault-tolerant state machine for arbitrarily faulty processes using PBFT (Practical Byzantine Fault Tolerance) [CL02].

**Example 2.3.1**

$$\begin{aligned}
\Pi =\ & \{p_1, p_2, p_3, p_4, p_5\} \\
\mathcal{C}_\Pi =& \{\{p_1, p_2, p_3\}, \{p_4, p_5\}\} \cup \{\{p_i, p_j\} : i \in \{1,2,3\} \wedge j \in \{4,5\}\} \\
\mathcal{S}_\Pi =& \{\{p_i, p_4, p_5\} : i \in \{1,2,3\}\} \cup \{\{p_1, p_2, p_3, p_i\} : i \in \{4,5\}\}
\end{aligned}$$

PBFT is an attractive protocol because it assumes a weak failure model. It was designed, however, assuming a threshold failure model. In the system profile above, the smallest survivor set has three processes, which means that there are executions in which two processes are faulty. Hence, there is *not* enough replication to run PBFT: seven processes are required to tolerate two faulty processes.

Using our model instead, we are able to have a working implementation of PBFT by having some processes running one copy of a version and others running simultaneously two copies of the same version, and each copy voting as a separate process. More specifically, if we have five processes, then we distribute copies of the versions $v_1$ through $v_5$ as follows: one process executes one copy of $v_1$, one process executes one copy of $v_2$, one process executes one copy of $v_3$, one process executes two copies of $v_4$, and one process executes two copies of $v_5$. It is easy to check that there is no more than two faulty copies in any execution of this configuration under the failure assumptions we previously described for these versions.

As an alternative to having multiple copies in each process, we can implement a replica coordination component with a modified version of PBFT that can be run in the five-process system of Example 2.3.1. In this case, the PBFT implementation needs to know the system profile in the same way that an unmodified PBFT (one assuming a threshold) needs to know the maximum number of faulty processes in an execution.[2]

This example illustrates an important point about dependent failures. Since IID failures can be represented as a particular system profile, lower bound proofs that hold for IID failures also hold in our model. However, if one has a system in which failures are not IID, then one should use an algorithm that explicitly uses a system profile. By using such an algorithm, it is often possible to use less replication than required when using an algorithm developed using the threshold model. Example 2.3.1 illustrates this point. Because the maximum number of failures is two (the smallest survivor set has size three and there are five processes), an algorithm that requires $n > 3 \cdot t$ in the threshold model cannot be executed in such a system as it requires at least seven processes. According to our previous discussion, the amount of replication in the system of Example 2.3.1 is sufficient for the algorithms in Chapter 3.

[2]The original PBFT algorithm assumes a threshold on the number of failures, but it is possible to modify it to work with cores and survivor sets. Although we do not show for the general case, we show for one instance of the PBFT algorithm, which is the algorithm of Section 3.4.2.

## 2.4 Survivor sets, fail-prone systems, and adversary structures

We are not the first to consider non-IID behaviors: quorum systems have addressed the issues of non-IID behavior for some time. In [MR97a], the idea of *fail-prone systems* was introduced. This paper gives the following definition for a set of servers $U$:

> A *fail-prone system* $\mathcal{B} \subseteq 2^U$ is a non-empty set of subsets of $U$, none of which is contained in another, such that some $B \in \mathcal{B}$ contains all the faulty servers.

This paper then observes that a fail-prone system can be used to generalize to less uniform assumptions than a typical threshold assumption. Their definition does not give a name to the elements of $\mathcal{B}$; we call each one a *fail-prone set*. As fail-prone sets are maximal, a fail-prone set is the complement of a survivor set and $\mathcal{B} = \{\Pi \backslash S : S \in \mathcal{S}_\Pi\}$. Although both survivor sets and fail-prone sets characterize failure scenarios, survivor sets have a fundamental use: if a process is collecting messages from the other processes, it can be fruitless to wait for messages from a set larger than a survivor set. Of course, there are times when fail-prone sets are more useful. For example, if $B_{\text{max}}$ is the largest fail-prone set, then $|B_{\text{max}}|$ is the value of $t$ if one wishes to use a threshold-based protocol.

Non-threshold protocols have also been considered in the context of secure multi-party computation with adversary structures [AFM99, HM97, KF05]. Adversary structures are similar to fail-prone systems; yet differ in two ways. First, adversary structures can represent more than one failure mode, *e.g.*, crash failures and arbitrary failures. Each failure mode is described with sets of possibly faulty processes (processes are referred to as *players* in this literature). Second, the sets of possibly faulty players given in an adversary structure are not necessarily maximal; all sets of possibly faulty players are given. Using all possible sets of players that can deviate from the correct protocol behavior as opposed to only maximal sets (or minimal sets of correct processes, as with survivor sets) gives one more expressiveness in modeling system failures. Using fail-prone systems or survivor sets, however, is sufficient for establishing the properties on process replication in the following chapters. Moreover, these bounds hold even for a more expressive model such as adversary structures because we use properties about the intersections of sets of correct processes. If the intersection property holds for some sets of processes $A_1$, $A_2$, ..., $A_m$ then it holds for the sets of processes $A'_1 \supset A_1$, $A'_2 \supset A_2$, ..., $A'_m \supset A_m$. Hence, one only has to consider the minimal sets of correct processes in these intersection properties.

## 2.5 Discussion

The model of this chapter enables the design of fault-tolerant algorithms for systems in which failures are dependent. As we show in the following chapters, there are several benefits in using a more expressive model.

The model has three limitations, however. First, there can be an exponential number cores and survivor sets. Representing and selecting such subsets in system profiles with an exponential number of subsets may not be practical. Our experience with this model, however, shows that often the number of subsets is not exponential and that it is possible to provide a compact representation for the set of survivor sets. Also, it may not be necessary to know all the cores or all the survivor sets. As we show in the next chapter, in the case of synchronous consensus for crash failures, processes need to know only of a single core. Second, the model does not describe a particular technique for extracting the cores and survivor sets of a system. That is, the model assumes these sets are given in the same way that the threshold model assumes that there is a way of computing a threshold $t$. To obtain cores and survivor sets, different techniques can be used: we presented an example in Section 2.3, and we provide more examples in the following chapters. Third, cores and survivor sets are in principle static sets. If these sets change over time, then a service to update these sets and guarantees consistency across correct processes is necessary. A potential candidate for such a mechanism is the RAMBO memory service [LS02].

The following chapters discuss the design of algorithms using cores and survivor sets (Chapters 3 and 4), and how to obtain cores and survivor sets in real systems (Chapters 6 and 5).

# Chapter 3

# Consensus for dependent process failures

Consensus is an important primitive in fault-tolerant distributed computing because it enables solutions for several problems that involve distributed coordination. Using a consensus primitive, it is possible, for example, to build fault-tolerant services using state-machine replication [Sch90]. To implement a replicated state machine, it is necessary to guarantee that replicas agree on the commands they execute and the order they execute these commands. The utility of consensus comes from this necessity of agreement among the replicas. A different and useful form of the consensus problem is atomic broadcast [CT96]. Atomically broadcasting messages consists in guaranteeing that messages sent by correct processes are eventually delivered, that correct processes deliver the same set of messages, and that processes deliver messages in the same order.

The consensus problem in a fault-tolerant message-passing distributed system consists, informally, in having a set of processes reaching agreement upon a value. Each process starts with a proposed value and the goal is to have all non-faulty processes eventually deciding upon the same value.

In the remainder of this chapter, we first discuss related work in Section 3.1. We then present the consensus specification (3.2). In Section 3.3, we show lower bounds on process replication and algorithms for synchronous systems, considering both crash and arbitrary process failures. We also show in Section 3.3.2 the lower bound on the number of rounds to solve consensus. In Section 3.4, we show lower bounds on process replication and provide algorithms for asynchronous systems extended with unreliable failure detectors, also considering both crash and arbitrary failures. The algorithms we present are modified versions of existing ones, and the general goal for presenting them is not only to obtain consensus algorithms in our model, but also to illustrate how to design algorithms.

## 3.1 Background

Agreement problems such as consensus have been broadly studied since the early 80's. Initially, the focus was on a related problem called interactive consistency [LSP82]. Different from consensus, interactive consistency assumes a single proposer, and a solution to this problem guarantees that all correct processes eventually decide upon the same value and that the decision value is the value proposed by the proposer if the proposer is not faulty. It is straightforward to generalize an interactive consistency solution to a consensus solution.

A system model that many algorithms assume is the one of a synchronous system. In a synchronous system, message latency, clock drift, and process speed are all bounded, thereby permitting executions to proceed in synchronous rounds. Lamport *et al.* showed that in synchronous systems at least $3t + 1$ processes are necessary and sufficient to solve interactive consistency without digital signatures, if processes fail arbitrarily, and $t + 1$ are necessary and sufficient with digital signatures [LSP82]. In [DS83], Dolev and Strong present synchronous algorithms for interactive consistency that improve on message complexity compared to previous algorithms. As these algorithms are efficient with respect to message complexity, the techniques they introduce are commonly used in synchronous algorithms for consensus assuming benign process failures, such as crash failures [AW98c]. Dolev and Strong also show that even when using authentication, the lower bound on number of rounds is $t + 1$. This lower bound was previously shown for unauthenticated channels by Fischer and Lynch [FL82].

As implementing synchronous systems is difficult in practice, a question that followed this initial effort was how to move away from the synchronous model. In [FLP85], Fischer *et al.* show the impossibility of consensus in asynchronous systems, even if a single process can crash. This is commonly known as the *FLP impossibility result*. To overcome this impossibility result, Chandra and Toueg proposed to extend the asynchronous model with failure detectors [CT96]. The FLP impossibility result derives from the impossibility of distinguishing slow processes from faulty processes in asynchronous systems. Unreliable failure detectors do not enable this distinction, but they provide weak properties that enable processes to eventually make progress. With the failure detector approach, processes use failure detector modules that provide failure information about other processes. Such failure detectors can make mistakes, however, although not infinitely often. In [CHT96], Chandra *et al.* showed the weakest properties that a failure detector has to satisfy to solve consensus.

Failure detectors guarantee that non-faulty processes eventually decide. In a more recent trend, proposed asynchronous algorithms ensure safety, but not liveness. This implies that processes may never decide, although under realistic assumptions such executions are highly unlikely to happen. To guarantee progress all these algorithms assume a leader oracle. Castro and Liskov describe such an algorithm that is practical and tolerates Byzantine failures [CL02]. Lamport proposes the Paxos algorithm [Lam98]. An interesting characteristic of the Paxos algorithm is its division of roles. In Paxos, processes can be proposers, acceptors, or learners, and these roles are not mutually exclusive. Informally, proposers propose values that are accepted by acceptors. Once an acceptor accepts a value, it informs the learners. Learners learn a value $v$ once they receive $v$ from a quorum of acceptors. Assuming that client processes present values to proposers, learning takes three message rounds in the best case, where a message round corresponds to the delay for some process $q$ to receive a message sent by some process $p$. A variant of the Paxos algorithm, called Fast Paxos, enables learning in fewer message rounds in the absence of conflicting proposals [Lam05]. Both flavors of Paxos tolerate crash failures. Due to the similarity between the algorithm by Castro and Liskov and Paxos, the former is often referred to as Byzantine Paxos. Fast learning is also possible for Byzantine failures, as illustrated with the algorithm by Martin and Alvisi [MA05].

Solutions to the consensus problem are usually developed assuming no more than $t$ of the $n$ processes are faulty, where "being faulty" is specialized by a failure model. Recall that this assumption corresponds to the threshold model, discussed in Chapter 2. This is a convenient assumption to make. For example, bounds are easily expressed as a function of $t$: if processes can fail only by crashing, then the consensus problem is solvable when $n > t$ if the system is synchronous [AW98c] and when $n > 2t$ if the system is asynchronous extended with a failure detector of the class $\diamond W$ [KR01, CT96].

## 3.2 Consensus specification

In this section, we present a formal specification of consensus. This specification comprises three properties that constrain the set of admissible executions of a consensus algorithm. As we discussed before, every process proposes its initial value. Throughout the remainder of this chapter, we use $V$ to denote the set of possible decision values. We assume that the set $V$ is $n$-ary, $n \geq 2$. Considering binary decision sets is important when showing lower bounds, as this is the strongest case: if there is an algorithm that solves for $V$, $|V| = k > 2$, then we can use this same algorithm to solve for $V'$, $2 \leq |V'| < k$. On the other hand, algorithms that assume a set $V$ of arbitrary size are more general. We state clearly when we use a binary set or an $n$-ary set for some arbitrary value of $n$ greater than one. We also assume a default value $\perp$ used in the algorithms that is not in $V$. When a value $v$ is either a decision value in $V$ or the default, we use $v \in V \cup \{\perp\}$ to denote all the possibilities.

Assuming that processes only fail by crashing, consensus has the following three properties [DS98]:

**Validity:** If some non-faulty process $p_i \in \Pi$ decides on value $v$, then some process

$p_j \in \Pi$ proposes $v$;

**Agreement:** If two non-faulty processes $p_i, p_j \in \Pi$ decide on values $v_i$ and $v_j$, respectively, then $v_i = v_j$;

**Termination:** Every correct process eventually decides.

When failures are arbitrary, a stronger validity property may be necessary as algorithms satisfying the previous validity property can prevent progress. For example, suppose that the only possible decision values are write and abort. With the above validity property a faulty process may prevent correct processes from writing if they are all ready to do so, and consequently from making progress.

Thus, assuming arbitrary failures for processes, a stronger version of validity, called strong validity, is more appropriate [DS98, KMMS97]:

**Strong Validity:** If all non-faulty processes propose $v$, then all non-faulty processes eventually decide $v$.

Compared to validity, strong validity restricts the case in which all non-faulty processes have the same initial value. Intuitively, this is sufficient to prevent a Byzantine process from disrupting the normal behavior of a system when all non-faulty processes are able to make progress.

Doudou and Schiper propose yet another variant of the validity property, called *vector validity* [DS98]. The vector validity property states that correct processes decide upon a vector of proposed values, such that the vector has one entry for each process in $\Pi$. Upon decision, for every correct process $p$, the value in the vector corresponding to $p$ is either the initial value of $p$ or some default value. The vector must also contain at least $f + 1$ initial values of correct processes. Observe that under these assumptions, agreement refers to vectors, and not to a single value. A modification of agreement to comply with this new requirement is straightforward. If the processes have to decide upon a single value instead of a vector of values, we can use a solution to vector consensus by using a deterministic function to evaluate to some value based on the decided vector.

Although vector consensus (consensus with the vector validity property) is more general than strong consensus (consensus with the strong validity property), we have chosen to use the strong version when assuming that processes can fail arbitrarily, as often processes need to reach agreement upon a single value.

## 3.3 Synchronous systems

In this section, we present results for consensus in synchronous systems using our new model. First, we constrain the set of executions to contain only executions that can be divided into rounds. Second, we present a lower bound on the number of rounds to solve consensus in our model of dependent failures. Surprisingly, a conclusion of this result is that the lower bound differs for crash and arbitrary failures. Finally, we describe and prove correct two consensus algorithms: one assuming crash process failures and another considering arbitrary failures. These algorithms are modified versions of algorithms previously proposed in the literature.

### 3.3.1 Synchronous executions

In synchronous systems, there are bounds on message delay, process speed, and clock drift. These bounds, however, are not necessarily based on absolute time. As in the model of Dolev *et al.* [DDS87], steps of processes define these bounds. In a step a process accomplishes one of the following: 1) receive a message; 2) undergo a state transition; 3) send a message to a single process.

Following this model, the timing assumptions for a synchronous system are given by two integer parameters corresponding to steps of processes: $\Phi \geq 1$ and $\Delta \geq 1$. Any execution of an algorithm $\alpha$ in such a system satisfies the following properties:

**Process synchrony:** for any finite subsequence $w$ of consecutive steps, if some process $p_i$ takes $\Phi + 1$ consecutive steps in $w$, then any process that is not faulty at the end of $w$ has taken at least one step in $w$;

**Message synchrony:** for any pair of indices $k, l$, with $l \geq k + \Delta$, if message $m$ is sent to $p_i$ during the *k-th* step, then $m$ is received by the end of the *l-th* step.

An execution is further organized in rounds, each round composed of steps of processes. In a round, a process $p_i$ executes $n + k$ steps. The first $n$ steps are used by $p_i$ to send real messages, whereas in the last $k$ steps it sends *null* messages. These $k$ last steps are necessary to guarantee that all messages sent to $p_i$ in a round $r$ are received before $p_i$ proceeds to round $r + 1$. The number $k$ of steps is a function of $\Delta$, $\Phi$, $n$, and $r$.

We make some additional assumptions regarding the executions of the algorithms we present in this section and define a few terms we use throughout this section. First, we assume that processes do not recover. Thus, the function *Faulty*$(s)$ is monotonically increasing in every execution. Second, if $E = \langle \mathit{Init}, \mathit{Steps}, \mathit{Time}, \mathit{Faulty} \rangle \in \mathcal{E}$ is an execution, then we use *Faulty*$(E)$ to denote the set of faulty processes in E. That is, $\mathit{Faulty}(E) = |\bigcup_{s \in \mathit{Steps}} \mathit{Faulty}(s)|$. Third, a process is correct in an execution if it does not fail in this execution. Finally, we say that a process is non-faulty if either it is correct, it has not crashed, or it has not executed any incorrect step.

### 3.3.2 Lower bound on the number of rounds

To show the lower bound result on the number of rounds, we use the technique of layering proposed by Keidar and Rajsbaum [KR01]. The general idea is to show by construction that the application of environment actions to some initial state still results in states in which non-faulty processes cannot decide. An example of an environment action is a process failure.

A layering is defined as a set of environment actions. The set of possible actions depends upon the failure modes assumed. In our case, we assume that a layering consists of crashing at most one process in a round[1]. Before crashing in a round, a

[1] Of course, there are executions in which more than one process crash in a round, but it is not necessary to consider these to show the lower bound.

process is allowed to send messages to a number of processes. Recall that from our system model, a process $p_i$ sends at most one message to another process $p_j$ in each step. We split processes into two groups: *active* processes and *passive* processes. In a round, only active processes send messages to other processes, whereas passive processes do not send any messages (a correct process can be passive due to the algorithm). We then use $(i, [j])$ to denote a layer applicable to a round $r$ in which process $p_i$ fails in $r$, and the messages $p_i$ sent to processes $\{p_1 \cdots p_j\} \subseteq \Pi$ are not received. Note that layer $(i, [j])$ is applicable if and only if $p_i$ is an active process.

We apply a layer to a state. If $x$ is a state, then we denote the application of a layer $\ell$ to $x$ as $x \cdot \ell$. We define a state as a string of entries, one for each process. Each entry comprises the local state of a process at a given round. If some process $p_i$ is crashed at round $r$, then the state of $p_i$ is represented by a special symbol denoting that it has crashed. We assume that no process crashes before the first round.

For the consensus problem, every process has an initial value, and we assume without loss of generality that the set of values is binary. Thus, for every binary string $w$ of length $|\Pi| = n$, there is an initial state $x_w$, as we assume that no process crashes before the first round, and *Init* is the set of all possible initial states. Note that a layer $(i, [j])$ is only applicable to some state $x$ if $p_i$ is not crashed in $x$.

The application of a sequence of layers to some initial state $x$ partially defines an execution. That is, if $x$ is the initial state of the processes and $\ell_1\ell_2 \cdots \ell_k$ is a sequence of $k$ distinct layers, then $((\cdots((x \cdot \ell_1) \cdot \ell_2) \cdots) \cdot \ell_k)$ partially characterizes the steps that processes take in execution. Observe that it does not say what the steps after round $k$ are, and it does not say in which order the steps of a round occur.

Let *sys* = $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ be a system profile, and *Active* be the set of active processes in $\Pi$. By the system profile, we have that the maximum number of failures $f_a$ in *Active* is given by:

$$f_a = \max_x \{x : x = |\mathit{Active} \cap (\Pi \setminus S)| \wedge S \in \mathcal{S}_\Pi\}$$

We then define the following layering for our model:

$$\mathbf{L} = \{(p, [q]) \mid p \in \mathit{Active},\ [q] = \{1 \cdots q\} \subseteq \Pi\}$$

We use $\mathbf{L}(x) = \{x \cdot \ell | \ell \in L\}$ to denote the application of layering $\mathbf{L}$ to state $x$ and $\mathbf{L}(X) = \{\mathbf{L}(x) | x \in X\}$ to express the application of layering $\mathbf{L}$ to the set of states $X$. In addition, we define $\mathbf{L}^i$ as the application of $\mathbf{L}$ for $i$ consecutive times. More specifically, we define $\mathbf{L}^i$ as follows:

$$\begin{aligned} \mathbf{L}^0(X) &= X \\ \mathbf{L}^k(X) &= \mathbf{L}(\mathbf{L}^{k-1}(X)) \end{aligned}$$

We observe, however, that we can have no more than $f_a$ layering applications. Recall that $f_a$ is the maximum number of failures among active processes. Thus, the system profile constrains the number of consecutive applications of $\mathbf{L}$.

Another important definition is the one of similar states. Similarity of states captures the notion of states in which a correct process cannot make a decision because there is not sufficient information for it to do so. We use this notion in the proofs we present below. Similar states and similarity connected sets of states are defined as follows:

**Definition 3.3.1** States $x$ and $y$ are similar, denoted $x \sim y$, if there is a process $p_j$ that is not faulty in these states, such that (a) $x$ and $y$ are identical except in the local state of $p_j$, and (b) there exists $p_i \neq p_j$ that is not faulty in both $x$ and $y$. A set of states is similarity connected if for every $x, y \in X$ there are states $x = x_0, x_1, \cdots, x_m = y$ such that $x_i \sim x_{i+1}, 0 \leq i \leq m$.

Note that a set of states for a given system $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ can only be similarity connected if the set of processes contains at least two processes, *i.e.*, $|\Pi| > 1$. However, recall from Chapter 2 that the system profiles we assume do not have any single process in all the survivor sets and every process is in at least one survivor set. These two assumptions constrains system profiles to have at least two processes.

We now show that *Init* is similarity connected with the following lemma.

**Lemma 3.3.2** Init *is similarity connected.*

**Proof:**

Given a state $z$, we denote by $z_j$ the local state of process $p_j$ in the state $z$. Let $y, y'$ be two states in *Init*. For every $0 \leq m \leq n$, define $x^m$ by setting $x^m_j = y_j$ for all $j > m$ and $x^m_j = y'_j$ for all $j \leq m$. We get: $x^0 = y$ and $x^n = y'$. Note that $x^m$ and $x^{m+1}$ differ exactly in the local state of process $p_m$. Since all the processes are non-faulty in every state in *Init*, these states are similar, that is, $x^m \sim x^{m+1}$.

□

Now, we show that any $k \leq f_a$ consecutive applications of layering $\mathbf{L}$ on a similarity connected set of states generates another similarity connected set of states. With the following lemma, we show that after $f_a$ layering applications on a similarity set of states we still have a similarity connected set of states.

**Lemma 3.3.3** *Let X be a similarity connected set of states in which no process is faulty and there are at least two correct processes.* $L^k(X)$ *is similarity connected for all* $k \leq f_a$.

**Proof:**

We prove by induction. The base case is $k = 0$. By definition, we have that $\mathbf{L}^0(X) = X$. Consequently, $\mathbf{L}^0(X)$ is similarity connected. The induction hypothesis is that $\mathbf{L}^{k-1}(X)$ is similarity connected and we want to show that $\mathbf{L}(\mathbf{L}^{k-1}(X))$ is also similarity connected. To show this, we need to demonstrate that the two following properties hold:

1. if $x \in \mathbf{L}^{k-1}(X)$ then $\mathbf{L}(x)$ is similarity connected;

2. if $y, y' \in \mathbf{L}^{k-1}(X)$, $y \sim y'$, then $\mathbf{L}(y) \cup \mathbf{L}(y')$ is similarity connected;

1: Suppose we apply layers $(i, [0])$ and $(j, [0])$ to $x$. Because no process is faulty in these layers, we have that $x \cdot (i, [0])$ and $x \cdot (j, [0])$ are identical. Now let us apply layers $(i, [l-1])$ and $(i, [l])$ to $x$. We have that the state of processes in $x \cdot (i, [l-1])$ and $x \cdot (i, [l])$ differ only in the state of $p_l$, and therefore they are similar.

2: $y$ and $y'$ differ in the state of one process, say $p_i$. If we apply layer $(i, [n])$ to both states, we get $y \cdot (i, [n])$ and $y' \cdot (i, [n])$. Notice that in this round, no process receives a message from $p_i$. Moreover, all processes besides $p_i$ have identical state in $y$ and $y'$ and consequently the messages they send have to be the same. Therefore, we have that $y \cdot (i, [n]) = y' \cdot (i, [n])$. Along with Property 1, this proves our claim that $\mathbf{L}(y) \cup \mathbf{L}(y')$ is similarity connected.

□

Using the two previous lemmas, we show that there is no consensus algorithm such that the size of the set of active processes is smaller than a smallest core. Recall that a core is a minimal set of processes such that in every execution at least one of the processes in the set is correct. First, we show a preliminary lemma.

**Lemma 3.3.4** *Let $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ be a system profile, and $C_{\text{min}}$ be an element of $\{C : C \in \mathcal{C}_\Pi \wedge \forall C' \in \mathcal{C}_\Pi : |C_{\text{min}}| \leq |C'|\}$. If $\Pi' \subseteq \Pi$ is such that $|\Pi'| < |C_{\text{min}}|$, then there is an execution in which all processes in $\Pi'$ are faulty.*

**Proof:**

Proof by contradiction. Suppose a subset that $\Pi' \subseteq \Pi$ is such that $|\Pi'| < |C_{min}|$ and there is no execution in which all the processes in $\Pi'$ are faulty. If $\Pi'$ contains at least one correct process in every execution, then $\Pi'$ contains a core. Consequently, $C_{min}$ is not in $\{C : C \in \mathcal{C}_\Pi \wedge \forall C' \in \mathcal{C}_\Pi : |C_{min}| \leq |C'|\}$, a contradiction.

□

We are now ready to state a theorem that establishes the minimum number of active processes.

**Theorem 3.3.5** *Let $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ be a system profile and $C_{\text{min}}$ be a smallest core in $\mathcal{C}_\Pi$. There is no consensus algorithm such that $|\text{Active}| < |C_{\text{min}}|$.*

**Proof:**

By Lemma 3.3.4, every subset $\Pi'$ of processes with fewer than $|C_{min}|$ elements is such that there is some execution in which all processes in $\Pi'$ are faulty. By Lemma 3.3.2, the set of initial values is similarity connected. By Lemma 3.3.3, assuming that $|Active| < |C_{min}|$, the set of states obtained after $|Active|$ layer applications is also similarity connected. Having a set of states that is similarity connected after $x$ layer applications, for some value of $x$, implies that there is some execution in which some correct process cannot decide after $x$ rounds. Consequently, there is some execution $E$ and some process $p_i$ correct in $E$ such that $p_i$ cannot decide after $|Active|$ rounds. As all active processes fail after $|Active|$ rounds of such execution, the state of $p_i$ does not change in subsequent rounds, and consequently $p_i$ never decides, thus violating termination. We conclude that there cannot be a consensus algorithm such that $|Active| < |C_{min}|$.

□

We now use the previous lemmas to show a theorem that provides the lower bound on the number of rounds. The theorem is as follows:

**Theorem 3.3.6** *Let sys = $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ be a system profile, $\mathcal{A}$ be a consensus algorithm, and $f_a$ be the maximum number of faulty active processes. If $|\Pi| - f_a > 1$, then there is an execution of $\mathcal{A}$ in which $f \leq f_a$ processes are faulty and some correct process takes at least $f + 1$ rounds to decide.*

**Proof:**

By Lemma 3.3.2, the set of initial states is similarity connected. According to Lemma 3.3.3, the $f$-th application of layering $\mathbf{L}$ on the set of initial states *Init* results in another similarity connected set of states. Thus, there is some execution in which after $f$ rounds there is at least one correct process that has not yet decided. We conclude that at least $f + 1$ rounds are required for all correct processes to decide.

□

Now we show a theorem that determines the lower bound on the number of rounds in the case that there are executions with a single correct process.

**Theorem 3.3.7** *Let $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ be a system profile, $\mathcal{A}$ be a consensus algorithm, and $f_a$ be the maximum number of faulty active processes. If $|\Pi| - f_a = 1$, then there is an* *execution of $\mathcal{A}$ in which $f \leq f_a$ processes are faulty and some correct process takes at least $min(f_a, f+1)$ rounds to decide.*

**Proof:**

Suppose that $0 \leq f \leq |\Pi| - 2$. By Theorem 3.3.6, if there are at least two correct processes, then there is at least one execution in which some correct process requires at least $f + 1 \leq f_a$ rounds to decide.

Lemma 3.3.3 does not include the case in which $f = |\Pi| - 1$. We hence show this case separately. After $|\Pi| - 2$ layer applications on the set of initial states, by using a similar proof as the one of Theorem 3.3.6, we can show that we obtain a similarity connected set of states. Consequently, there is an execution with $|\Pi| - 1$ faulty processes in which the single correct process cannot decide before $f_a = |\Pi| - 1$ rounds.

For every consensus algorithm $\mathcal{A}$ assuming a system such that $|\Pi| - f_a = 1$, there is therefore some execution of $\mathcal{A}$ in which some correct process does not decide earlier than $\min(f+1, |\Pi| - 1)$.

□

### 3.3.3 Synchronous consensus for crash failures

The consensus problem in synchronous systems with crash process failures is solvable for any number of failures [CBS00]. Executions in which every process is faulty, however, are not interesting because it may not be possible to know the outcome of the computation, unless we use recovery techniques. As faulty processes do not recover in our model, we assume for the following algorithm that in every execution there is at least one correct process. Using our model, this implies that, for a given a system profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$, there is at least one core in $\mathcal{C}_\Pi$:

**Property 3.3.8** $C_\Pi \neq \emptyset$. □

There is another important reason for assuming systems that have at least one core. In doing so, we will see below that we can make the processes of a core active and the remaining ones passive. The main advantage is reducing the number of rounds necessary for non-faulty processes to decide.

We now move on to discuss a consensus algorithm for synchronous systems. This algorithm assumes a system profile $\langle \Pi, \mathcal{C}_{\Pi}, \mathcal{S}_{\Pi} \rangle$ such that Property 3.3.8 holds for $\langle \Pi, \mathcal{C}_{\Pi}, \mathcal{S}_{\Pi} \rangle$. The algorithm is based on the early-deciding ones in [CBS00] and [LF82]. Early-deciding algorithms use the actual number of failures $f$ in an execution to determine in which round a non-faulty process decides. Algorithms that consider the actual number of failures $f$ are important because they reduce the latency when there are no failures, which is often the common case. Charron-Bost and Schiper show that, for every early-deciding algorithm, there is some execution with $f$ failures in which no correct process decides before $f + 1$ rounds [CBS00].

Our algorithm, which we call SyncCrash, differentiates the processes of a core *d-core* $\in \mathcal{C}_{\Pi}$, selected arbitrarily, and the processes in $\Pi \setminus$ *d-core*. In a round, every process in *d-core* broadcasts its knowledge of proposed values to all the other processes, whereas processes in $\Pi \setminus$ *d-core* listen to these messages. Note that according to our terminology when showing the lower bound on the number of rounds in Section 3.3.2, the processes in *d-core* are active and the others are passive.

Because we assume synchronous systems, if a process $p_i$ does not receive a message in a round from some process $p_j \in$ *d-core*, then $p_i$ knows $p_j$ is faulty. We use this observation to detect rounds in which no process in *d-core* crashes. A process $p_i \in \Pi$ hence keeps track of the processes in *d-core* that crash in a round, and as soon as $p_i$ detects a round with no crashes, $p_i$ decides. Once a process $p_i$ in *d-core* decides, it broadcasts a decision message announcing its decision value $dec_i$. All processes receiving this message decide on $dec_i$ as well. Thus, only two types of messages are necessary in the algorithm: messages containing the array of proposed values and decide messages.

As every process in *d-core* broadcasts at most one message in every round to all the other processes in $|\Pi|$, message complexity is $O(|\textit{d-core}| * |\Pi|)$. Note that the protocols in [CBS00, LF82] designed with the $t$ of $n$ assumption have message complexity $O(|\Pi|^2)$. If we select a smallest core for *d-core*, then message complexity is the smallest possible under this strategy of using a core of active processes.

In SyncCrash, correct processes decide within $f + 1$ rounds in a given execution $E$, where $f$ is the number of processes in *d-core* that crash in $E$. As at most $|$*d-core*$| - 1$ processes of core *d-core* can fail in some execution, by assumption, we have that, by Theorem 3.3.6, there is some execution in which $|$*d-core*$|$ rounds are necessary for all the correct processes to decide. As our algorithm matches this bound, we thus prove that the bound is tight. Note that we require at least one core of active processes. Otherwise there are executions in which a passive correct process never decides, thus violating the Termination property of consensus. Equally important is the observation that if there is one passive process and the set of active processes is smaller than the one of any core, then there is no algorithm that solves the consensus problem with such a set of active process.

The consensus service by Guerraoui and Schiper uses a similar approach of having a set of processes reaching consensus on behalf of a group of processes [GS96]. In their model, however, processes are either clients or servers. Servers processes take inputs from the clients and use them as their initial values. The decision is later propagated back to the clients. In our algorithm, there is no such a distinction between clients and servers, but there is a distinction between active and passive processes. Also, their failure model is still based upon the $t$-out- of-$n$ failure assumption for processes.

A consequence of having passive processes, as opposed to having all processes active, is that correct processes cannot decide upon the initial values of passive processes. This does not violate any of the consensus properties, however. We have decided to use just a subset of processes in a single core for lower bound purposes, and we can use the same algorithm having all processes active by replacing *d-core* with $\Pi$. Moreover, using just a subset has the advantage of reducing message complexity, as we discussed before.

Before presenting pseudocode for the algorithm, we show a table describing the variables used in the protocol. Table 3.1 describes the variables, and Figure 3.1

presents the pseudocode of SyncCrash. A specification of SyncCrash in TLA+ appears in Appendix B.

**Table 3.1:** Variables used in the algorithm SyncCrash

| | |
|---|---|
| $\mathit{d\text{-}core} \in C_\Pi$ | Set of active processes. |
| $p_i.\mathit{dec} \in V \cup \{\perp\}$ | A non-faulty process $p_i$ decides once it sets $p_i.\mathit{dec}$ to a value different than $\perp$. |
| $p_i.M(r)$ | Messages received in round $r$ by process $p_i$. |
| $p_i.\mathit{pv}$, $p_i.\mathit{pv}[j] \in V$, $j \in \mathit{d\text{-}core}$ | Vector of proposed values. |
| $p_i.e(r)$, $p_i.e(r) \subset \mathit{d\text{-}core}$, $r \in \{1 \ldots |\mathit{d\text{-}core}|\}$ | Mapping from round numbers to subsets of crashed processes. $p_i.e(r)$ evaluates to the subset of processes that $p_i$ detects to have crashed at round $r$. |

We now present a proof of correctness for SyncCrash. Before proving the theorems showing that our algorithm satisfies the three consensus properties, we show a few preliminary lemmas that we use later to show that the algorithm satisfies the consensus properties.

**Lemma 3.3.9** *Let $E \in \mathcal{E}$ be an execution of SyncCrash, $p_i$ be a process in* d-core*, and $p_j$ be a process in $\Pi$. Let $r$ be the first round in which $p_j$ receives a message $m$ from $p_i$ such that $m.\mathit{pv}[l] \neq \perp$, $p_l \in$* d-core*. For every round $r' < r$, $p_j.\mathit{pv}[l] = \perp$, and for every $r'' \geq r$, $p_j.\mathit{pv}[l] = v_l$.*

**Proof:**

We prove the lemma by induction on the round numbers $r$. The base case is $r = 1$. If $p_j$ receives a message $m$ from $p_i$ such that $m.\mathit{pv}[l] \neq \perp$, then $i = l$, and $m.\mathit{pv}[l] = v_l$ by the algorithm. Again by the algorithm, $p_j$ sets the value of $p_j.\mathit{pv}[l]$ to $v_l$ at round 1, and it does not change it in future rounds.

Now we assume that the lemma is valid for $r$ and we prove it for $r+1$. Suppose that $p_j$ receives a message $m$ from $p_i$ at round $r+1$ such that $m.\mathit{pv}[l] \neq \perp$, and $r+1$

```
**Algorithm** SyncCrash **for process** $p_i$:
**Input:** set $\Pi$ of processes; set $C_\Pi$ of cores; *d-core* $\in \mathcal{C}_\Pi$; initial value $v_i \in V$

**Initialization:**
    $p_i.dec \leftarrow \perp$
    $p_i.pv[j] \leftarrow \perp$, if $p_j \in$ d-core $\wedge\ j \neq i$. $p_i.pv[i] \leftarrow v_i$
    $p_i.e : 0 \cdots (|\text{d-core}|) \mapsto$ d-core

**Round** 0, $\forall p_i \in$ d-core:
    **send**$(i, p_i.pv)$ **to** all process in d-core
    **send**$(i, p_i.pv)$ **to** all process in $\Pi -$ d-core

**Round** $1 \leq r < |\text{d-core}|$, $\forall p_i \in$ d-core:
    **if** there is $m \in p_i.M(r)$ such that $m = (Decide, dec)$ **then** $p_i.dec \leftarrow dec$
    **else for** every $m = (j, pv)$ **do** $p_i.e(r) \leftarrow p_i.e(r) \setminus \{p_j\}$
      **for** $k = 1$ to $|\Pi|$ **do**
          **if** $(m.pv[k] \neq \perp \wedge p_i.pv[k] = \perp)$ **then** $p_i.pv[k] \leftarrow m.pv[k]$
    **if** $(p_i.e(r-1) = p_i.e(r))$ **then** $p_i.dec \leftarrow p_i.pv[k], k = min_x\{x : p_i.pv[x] \neq \perp\}$
    **if** $(p_i.dec = \perp)$ **then**
      **send**$(i, p_i.pv)$ **to** all process in d-core
      **send**$(i, p_i.pv)$ **to** all process in $\Pi -$ d-core
    **else**
      **send**$(Decide, p_i.dec)$ **to** all processes in d-core
      **send**$(Decide, p_i.dec)$ **to** all processes in $\Pi -$ d-core
      **halt**

**Round** $|\text{d-core}|$, $\forall p_i \in$ d-core:
    **if** there is $m \in p_i.M(r)$ such that $m = (Decide, dec)$ **then** $p_i.dec \leftarrow dec$
    **else** for every $m = (j, pv)$ **do** $p_i.e(r) \leftarrow p_i.e(r) \setminus \{p_j\}$
      **for** $k = 1$ to $|\Pi|$ **do**
          **if** $(m.pv[k] \neq \perp \wedge p_i.pv[k] = \perp)$ **then** $p_i.pv[k] \leftarrow m.pv[k]$
    $p_i.dec \leftarrow p_i.pv[k], k = min_x\{x : p_i.pv[x] \neq \perp\}$
    **halt**

**Round** $1 \leq r \leq |\text{d-core}|$, $\forall p_i \in \Pi -$ d-core:
    **if** there is $m \in p_i.M(r)$ such that $m = (Decide, dec))$ **then**
      $p_i.dec \leftarrow dec$
      **halt**
    **else**  for every $m = (j, pv)$ **do** $p_i.e(r) \leftarrow p_i.e(r) \cup \{j\}$
        **for** $k = 1$ to $|\Pi|$ **do**
            **if** $(m.pv[k] \neq \perp)$ **then** $p_i.pv[k] \leftarrow m.pv[k]$
    **if** $((p_i.e(r-1) = p_i.e(r)))$ **then**
        $p_i.dec \leftarrow p_i.pv[k], k = min_x\{x : p_i.pv[x] \neq \perp\}$
        **halt**
```

**Figure 3.1:** Algorithm SyncCrash

is the first round in which $p_j$ receives such a message. By the induction hypothesis, we have that $p_i.pv[l] = v_l$. By the algorithm, $p_j$ sets the value of $p_j.pv[l]$ to $v_l$ in round $r+1$, and it does not change it in future rounds. This completes the proof of the lemma.

□

From Lemma 3.3.9 we can extract the following corollary.

**Corollary 3.3.10** *Let $E \in \mathcal{E}$ be an execution of* SyncCrash*. For every $p_i \in$* d-core $\setminus$ Faulty$(E), p_j \in \Pi \setminus$ Faulty$(E)$ *and for every round $r \in \{1 \cdots |$d-core$|\}$, we have that $p_j.pv[i] = v_i$.*

**Proof:**

If $p_i \in$ *d-core* is correct, then for every non-faulty process $p_j$ in round 1, we have that $p_j$ receives a message $m$ from $p_i$ such that $m.pv[i] = v_i$. From Lemma 3.3.9, for every round $r$, $r \geq 1$, we have that $p_j.pv[i] = v_i$.

□

In the following lemmas, when we say that the vectors of proposed values of two processes $p_i, p_j$ are identical in a round $r$, we denote that, for every entry $l$, $p_i.pv[l] = p_j.pv[l]$ after all possible updates to the vector due to the reception of messages in round $r$. More formally, let $E = \langle Init, Steps, Time, Faulty\rangle$ be an execution of SyncCrash. Suppose that there are steps $s^i_r, s^j_r \in Steps$ of round $r$ of processes $p_i$ and $p_j$, respectively, such that $p_i.pv$ does not change in $r$ after $p_i$ executes $s^i_r$ and $p_j.pv$ does not change in $r$ after $p_j$ executes $s^j_r$. We then have that $p_i.pv$ and $p_j.pv$ are identical in round $r$ of $E$ if and only if $p_i.pv$ at $T(s^i_r) + 1$ is identical to $p_j.pv$ at $T(s^j_r) + 1$. We also say that a process $p_i$ is *alive* in round $r$ if either $p_i$ sends at least one message or $p_i$ decides in $r$.

**Lemma 3.3.11** *Let $E \in \mathcal{E}$ be an execution of* SyncCrash*, $r$ be a round of $E$, and $p_i, p_j$ be two alive processes in $r$, $r > 0$. If $p_i.e(r) = p_i.e(r-1)$ and $p_j.e(r) = p_j.e(r-1)$, then $p_i.e(r) = p_j.e(r)$ and $p_i.e(r-1) = p_j.e(r-1)$.*

**Proof:**

If $r = 1$, then it must be true because $p_i.e(r-1) = p_j.e(r-1) = \Pi$. Now we show for $r > 1$. We prove this case by contradiction.

First, suppose that $p_i.e(r) \neq p_j.e(r)$. This implies that there is some $p_l$ such that:

$$\bigvee p_l \notin p_i.e(r) \wedge p_l \in p_j.e(r)$$
$$\bigvee p_l \notin p_j.e(r) \wedge p_l \in p_i.e(r)$$

Suppose without loss of generality that $p_l \notin p_i.e(r)$ and $p_l \in p_j.e(r)$. Because $p_i.e(r) = p_i.e(r-1)$ by assumption, we have that $p_l$ is faulty in round $r-2$. Consequently, as a process that crashes in round $r$ does not send more messages in future rounds, we have that $p_l \notin p_j.e(r)$, a contradiction.

From the previous argument and by assumption, we have that: $p_i.e(r) = p_i.e(r-1)$, $p_j.e(r) = p_j.e(r-1)$, and $p_i.e(r) = p_j.e(r)$. This implies that $p_i.e(r-1) = p_j.e(r-1)$.

□

**Lemma 3.3.12** *Let $E \in \mathcal{E}$ be an execution of* SyncCrash*, $r$ be a round of $E$, $r > 0$, and $p_i, p_j$ be two alive processes in $r$. If both $p_i$ and $p_j$ decide in $r$ and none of $p_i, p_j$ receive a decide message, then $p_i.\mathrm{dec} = p_j.\mathrm{dec}$ at the end of round $r$.*

**Proof:**

By Lemma 3.3.11, both $p_i$ and $p_j$ receive the same set of messages. As none of these messages is a decide message by assumption, we have that $p_i$.*pv* is identical to $p_j$.*pv* in round $r$. By assumption, $p_i$ and $p_j$ decide in $r$. By the algorithm we have that $p_i$.*dec* must be equal to $p_j$.*dec*.

□

**Lemma 3.3.13** *Let $E \in \mathcal{E}$ be an execution of* SyncCrash*, $r$ be a round of $E$, $r > 0$, and $p_i, p_j$ be two alive processes in $r$. If both $p_i$ and $p_j$ decide in $r$, then $p_i$ decides in $r$ due to the reception of a decide message if and only if $p_j$ decides in $r$ due to the reception of a decide message.*

**Proof:**

Proof by contradiction. Suppose without loss of generality that $p_i$ decides in $r$, but it receives no decide message. This implies that, by the algorithm, $p_i.e(r) = p_i.e(r-1)$. Now suppose that $p_j$ receives a decide message. If $p_j$ receives a decide message from some process $p_l$ in round $r$, but $p_i$ does not receive such message, then $p_l \notin p_i.e(r)$. Because $p_j$ receives a message from $p_l$ in round $r$, we have that $p_l$ must be in $p_i.e(r-1)$. Consequently, $p_i.e(r) \neq p_i.e(r-1)$, a contradiction.

□

**Lemma 3.3.14** *Let $E \in \mathcal{E}$ be an execution of* SyncCrash*, $r$ be a round of $E$, and $p_i, p_j$ be two alive processes in $r$. If both $p_i$ and $p_j$ send decide messages* (Decide, $p_i$.dec) *and* (Decide, $p_j$.dec)*, respectively, in $r$, then* $p_i$.dec $= p_j$.dec.

**Proof:**

By the algorithm, if processes $p_i$ and $p_j$ decide in $r$, then, by Lemma 3.3.13, they either both receive a decide message or both detect a round without failures. If it is the latter, then they decide upon the same value by Lemma 3.3.12.

Suppose that both $p_i$ and $p_j$ decide due to the reception of decide messages, (*Decide*, *dec*$_{l_1^i}$) and (*Decide*, *dec*$_{l_1^j}$) respectively. For $p_i$, there is a sequence of processes $p_{l_k^i}, p_{l_{k-1}^i}, \ldots, p_{l_1^i}$ such that $p_{l_x^i}$ sends at least one decide message (*Decide*, $p_{l_x^i}$*.dec*) in round $r-x$, and process $p_{l_{x-1}^i}$ decides in round $r-x+1$ due to the reception of such a decide message from $p_{l_x^i}$. Similarly for $p_j$, there is a sequence of processes $p_{l_k^j}, p_{l_{k-1}^j}, \ldots, p_{l_1^j}$ such that $p_{l_x^j}$ sends at least one decide message (*Decide*, $p_{l_x^j}$*.dec*) in round $r-x$, and process $p_{l_{x-1}^j}$ decides in round $r-x+1$ due to the reception of such a decide message from $p_{l_x^j}$. Note that these two sequences of processes must have the same length by Lemma 3.3.13. By the algorithm, $p_{l_k^i}$ and $p_{l_k^j}$ are the first in their respective sequences to decide by detecting a round without crashes. By Lemma 3.3.12, $p_{l_k^i}$ and $p_{l_k^j}$ must decide upon the same value. Consequently, $p_i$*.dec* $= p_j$*.dec*.

□

Now we show that the algorithm satisfies all three consensus properties.

**Theorem 3.3.15** SyncCrash *satisfies Validity.*

**Proof:**

By the algorithm, a process decides either when it detects a round with no crashes or when it receives a decide message. In the first case, a non-faulty process $p_i$ decides upon the first value in its vector of proposed values that is different than $\perp$. Such a value exists by Corollary 3.3.10, and it is the initial value of some process in *d-core*.

In the second case, by the algorithm, a non-faulty process sends a decide message in a round that it either receives a decide message or detects a round with no crashes. If a non-faulty process receives a decide message in some round $r$, there must be an earlier round $r' < r$ such that some process alive in round $r'$ detects a round with no crashes, and consequently sends at least one decide message. By the same argument of the first case, such a decide message contains the initial value of some process in *d-core*.

□

**Theorem 3.3.16** *SyncCrash satisfies Agreement.*

**Proof:**

We have to show that in every execution, two correct processes $p_i, p_j$ decide upon the same value. There are two case to consider. First, $p_i$ and $p_j$ decide in the same round $r$. We then have the following:

1. $p_i$ and $p_j$ both receive a decide message in round $r$. By Lemma 3.3.14, we have that $p_i.dec = p_j.dec$;

2. $p_i$ and $p_j$ do not receive decide messages in round $r$. By Lemma 3.3.12, they must decide upon the same value;

3. $p_i$ does not receive a decide message in round $r$ and $p_j$ receives a message in round $r$. By Lemma 3.3.13, this case is not possible.

The second case to consider is the one in which $p_i$ and $p_j$ decide in different rounds. Suppose without loss of generality that $p_i$ decides in round $r$ and $p_j$ decides in round $r'$, $r' > r$. By the algorithm, $p_i$ sends a decide message to all the other processes, and therefore $r'$ must be $r + 1$.

□

**Theorem 3.3.17** *SyncCrash satisfies Termination.*

**Proof:**

By the algorithm, a correct process decides in at most $|$*d-core*$|$ rounds.

□

The following theorem shows that SyncCrash is early deciding. Recall from the beginning of this section that an early-deciding algorithm is one that differentiates executions based on the number of failures, and the number of rounds for processes to decide depends upon this number.

**Theorem 3.3.18** *Let $E \in \mathcal{E}$ be an execution of SyncCrash. If $p_i$ is a correct process in $E$, then it decides in at most $f + 1$ rounds, where $f =$ Faulty$(E)$.*

**Proof:**

By the algorithm, a process decides either when it detects a round without failures or when it receives a decide message. If a non-faulty process $p_i$ does not decide by round $f$, it implies that in every round $0 < r \leq f$, there is some process $p_l$ such that $p_l \in p_i.e(r-1)$ and $p_l \notin p_i.e(r)$. Because an execution can have at most $f$ failures, we have that $p_i.e(f + 1)$ must be equal to $p_i.e(f)$. This implies that if $p_i$ does not decide before round $f + 1$, it does so in round $r + 1$. Thus, every correct process decides at most in $f + 1$ rounds.

□

Before closing this section, we present an example of the benefit of using our model along with algorithm SyncCrash. Suppose a system with six processes, such that two processes are very reliable and the remaining processes are less reliable. One possibility for modeling the failures of such a system is as follows:

$$\Pi \quad = \quad \{p_{r_1}, p_{r_2}, p_{u_1}, p_{u_2}, p_{u_3}, p_{u_4}\}$$

$$\mathcal{C}_\Pi \;=\; \{\{p_{r_1}, p_{r_2}\}, \{p_{u_1}, p_{u_2}, p_{u_3}, p_{u_4}\}\}$$

That is, in every execution, at least one of the more reliable process ($p_{r_i}, i \in \{1, 2\}$) is not faulty, and at least one of the less reliable processes ($p_{u_i}, i \in \{1, 2, 3, 4\}$) is not faulty. Considering a threshold $t$ on the maximum number of failures, we have that the value of $t$ is four as every execution has at most four failures. As an algorithm that uses a threshold does not differentiates processes (assumes that processes are alike), the minimum number of active processes such an algorithm must use is five. Assuming such an algorithm selects exactly five processes, any choice of active processes will be such that there is some execution in which five active processes are faulty. To see why this statement holds, we just have to observe that no subset of five processes cannot include both cores. Thus, for any algorithm that uses a threshold, there are executions with four process failures such this algorithm requires at least five rounds. Using SyncCrash, we can reach agreement in two rounds, if we use *d-core* $= \{p_{r_1}, p_{r_2}\}$.

### 3.3.4 Synchronous consensus for Byzantine failures

In this section, we first present two properties that a system has to satisfy to enable a solution of strong consensus. Then we show that these properties are in fact equivalent. The reason for having two equivalent properties is that one is useful when showing necessity and the other is useful when designing algorithms.

A partition of a set $S$ is a collection of disjoint subsets of $S$, called *blocks*, such that the union of these blocks is equal to $S$. Let $\mathcal{P}_b(\Pi)$ be the set of all partitions of $\Pi$ into $b$ blocks. Given a system profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$, suppose the following properties:

**Property 3.3.19** (**Byzantine Partition**) $\forall (B_1, B_2, B_3) \in \mathcal{P}_3(\Pi) : \exists C \in \mathcal{C}_\Pi : (C \subseteq B_1) \vee (C \subseteq B_2) \vee (C \subseteq B_3)$

**Property 3.3.20** (**Byzantine Intersection**) $\forall S_{i_1}, S_{i_2}, S_{i_3} \in \mathcal{S}_\Pi : S_{i_1} \cap S_{i_2} \cap S_{i_3} \neq \emptyset$.

We now show that these two properties are equivalent.

**Claim 3.3.21** Byzantine Partition $\equiv$ Byzantine Intersection.

**Proof:**

**Byzantine Partition → Byzantine Intersection** Proof by contrapositive. Suppose that Byzantine Intersection does not hold for some system $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$. This implies that there are three survivor sets $S_{i_1}, S_{i_2}, S_{i_3} \in \mathcal{S}_\Pi$, such that $S_{i_1} \cap S_{i_2} \cap S_{i_3} = \emptyset$. We can then build a partition $(B_1, B_2, B_3)$ as follows:

$$\begin{aligned} B_1 &= \Pi \setminus S_{i_1} \\ B_2 &= \Pi \setminus (S_{i_2} \cup B_1) \\ B_3 &= \Pi \setminus (S_{i_3} \cup B_1 \cup B_2) \end{aligned}$$

We now show that $B_1 \cup B_2 \cup B_3 = \Pi$, followed by a detailed explanation:

$$\begin{aligned} \bigcup_i B_i &= (\Pi \setminus S_{i_1}) \cup (\Pi \setminus (S_{i_2} \cup B_1)) \cup (\Pi \setminus (S_{i_3} \cup B_1 \cup B_2)) && (3.1) \\ &= (\Pi \setminus (S_{i_1} \cap (S_{i_2} \cup B_1))) \cup (\Pi \setminus (S_{i_3} \cup B_1 \cup B_2)) && (3.2) \\ &= (\Pi \setminus (S_{i_1} \cap S_{i_2})) \cup (\Pi \setminus (S_{i_3} \cup B_1 \cup B_2)) && (3.3) \\ &= \Pi \setminus (S_{i_1} \cap S_{i_2} \cap (S_{i_3} \cup B_1 \cup B_2)) && (3.4) \\ &= \Pi \setminus (S_{i_1} \cap S_{i_2} \cap S_{i_3}) && (3.5) \\ &= \Pi && (3.6) \end{aligned}$$

- Line 3.1 to line 3.2: For arbitrary sets $X \setminus Y$ and $X \setminus Z$, $X \setminus Y \cup X \setminus Z = X \setminus (Y \cap Z)$;
- Line 3.2 to line 3.3: $S_{i_1} \cap B_1 = \emptyset$;
- Line 3.3 to line 3.4: For arbitrary sets $X \setminus Y$ and $X \setminus Z$, $X \setminus Y \cup X \setminus Z = X \setminus (Y \cap Z)$;
- Line 3.4 to line 3.5: $S_{i_1} \cap S_{i_2} \cap B_1 = S_{i_1} \cap S_{i_2} \cap B_2 = \emptyset$;
- Line 3.5 to line 3.6: By assumption, $S_{i_1} \cap S_{i_2} \cap S_{i_3} = \emptyset$.

By the construction of the partition, there is no block such that it contains elements from every survivor set. This implies that no block contains a core.

**Byzantine Intersection $\rightarrow$ Byzantine Partition** Proof by contrapositive. Suppose a system profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ such that Byzantine Partition does not hold for $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$. This implies that there is a partition of $\Pi$ into three blocks $(B_1, B_2, B_3)$ such that none of these blocks contains a core. We then have the following:

$$\begin{aligned}
\forall C \in \mathcal{C}_\Pi : C \not\subseteq B_1 &\Rightarrow \exists S_1 \in \mathcal{S}_\Pi : S_1 \subseteq B_2 \cup B_3 \\
\forall C \in \mathcal{C}_\Pi : C \not\subseteq B_2 &\Rightarrow \exists S_2 \in \mathcal{S}_\Pi : S_2 \subseteq B_1 \cup B_3 \\
\forall C \in \mathcal{C}_\Pi : C \not\subseteq B_3 &\Rightarrow \exists S_3 \in \mathcal{S}_\Pi : S_3 \subseteq B_1 \cup B_2
\end{aligned}$$

From the statements above, we have that $S_1 \cap S_2 \cap S_3 = \emptyset$, otherwise $(B_1, B_2, B_3)$ is not a partition violating our previous assumption.
$\square$

### 3.3.5 Lower bound on process replication

The Byzantine Intersection (Partition) property is necessary and sufficient for solving strong consensus in a synchronous system with Byzantine failures. First, we prove that this property is necessary. The proof we provide is based upon the one by Lamport [LSP82, PSL80]. We show that if there is a partition of the processes in three non-empty subsets, such that none of them contains a core, then there is at least one run in which Agreement is violated, for any algorithm $\mathcal{A}$. Figure 3.2 illustrates this violation, where we have three executions: $E_\alpha$, $E_\beta$, and $E_\gamma$.

Suppose that we have a system profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ and a partition of $\Pi$ into three blocks $(B_1, B_2, B_3)$ such that none of them contains a core. In addition, suppose by way of contradiction that we have an algorithm $\mathcal{A}$ that solves strong consensus in such a system.

In execution $E_\alpha$, the initial value of every the processes is the same, say $v$. Moreover, all the processes in subset $B_2$ are faulty, and they all lie to the processes in subset $B_3$ about their initial values and the values received from processes in $B_1$. Thus, running algorithm $\mathcal{A}$ in such an execution results in all the processes in subset $B_3$ deciding $v$, by the strong validity property. Execution $E_\beta$ is analogous to execution $E_\alpha$, but instead of every process beginning with an initial value $v$, they all have initial value $v' \neq v$. Consequently, by the strong validity property, all processes in $B_2$ decide $v'$ in this execution. Lastly, in execution $E_\gamma$, the processes in subset $B_3$ have initial value $v$, whereas processes in subset $B_2$ have initial value $v'$. The processes in subset $B_1$ are all faulty and behave for processes in $B_3$ as in execution $E_\alpha$. For processes in $B_3$, however, processes in $B_2$ behave as in execution $E_\beta$. Because processes in $B_3$ cannot distinguish execution $E_\alpha$ from execution $E_\gamma$, processes in $B_3$ have to decide $v$. At the same time, processes in $B_2$ cannot distinguish executions $E_\beta$ from $E_\gamma$, and therefore they decide $v'$. Consequently, there are correct processes that decide differently in execution $E_\gamma$, violating the Agreement property of strong consensus.

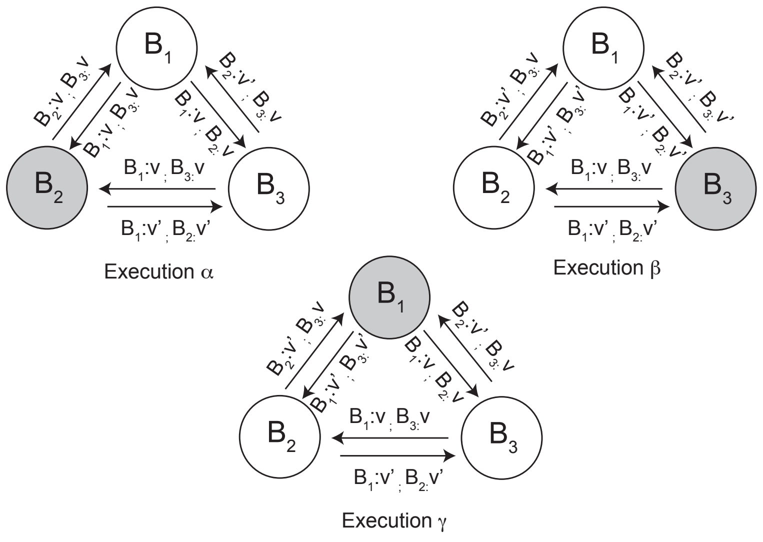

**Figure 3.2:** Violation of strong consensus. The processes in shaded subsets are all faulty in the given execution.

We now provide a more formal argument by proving Theorem 3.3.22. Before proceeding into the statement and proof of the theorem, we introduce some useful notation. Let $E_\alpha$ be an execution. We use $E_\alpha(\omega)$, $\omega = \langle i_1, i_2, \ldots, i_k \rangle$, to denote the value that process $p_{i_1}$ receives from process $p_{i_2}$, which claims that this value is the initial value of $p_k$ passed by process $p_i$ to process $p_{i-1}$ in this sequence of $k$ processes.

For example, $E_\alpha(\langle i, j, k\rangle)$ is the value that process $p_i$ receives from process $p_j$, which is the value that supposedly $p_k$ has sent to $p_j$ as its initial value. If the $k$-process chain contains only correct process, $k \geq 1$, then the value $E_\alpha(\langle i_1, i_2, \ldots, i_k\rangle)$ is the initial value of $p_k$. Otherwise, this property is not guaranteed. In the case that $k = 1$, we have that $E_\alpha(\langle i\rangle)$ is the initial value of process $p_i$. We use *S-Pid*$(\Pi)$ to denote the set of all possible sequences $\langle i, \ldots, j\rangle$ of process ids in $\Pi$, and "$\circ$" to denote the concatenation of two sequences (*e.g.*, $\langle i\rangle \circ \langle j\rangle = \langle i, j\rangle$). We also use *Len*$(\omega)$ to denote the length of sequence $\omega$.

**Theorem 3.3.22** *Let $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi\rangle$ be a system profile. If there is a partition $(B_1, B_2, B_3)$ of $\Pi$ such that none of $B_1$, $B_2$, or $B_3$ contains a core, then there is no algorithm which solves strong consensus in such a system.*

**Proof:**

We assume without loss of generality that none of $B_1$, $B_2$, or $B_3$ is empty.

Suppose there is an algorithm $\mathcal{A}$ which solves strong consensus in $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi\rangle$. We construct recursively an execution in which two correct processes decide differently. Moreover, the violation of agreement in this execution is independent of the number of rounds. Even if the algorithm runs for an infinite number of rounds, it cannot prevent a violation of agreement.

By assumption, there is a partition $(B_1, B_2, B_3)$ of $\Pi$ such that *none* of $B_1$, $B_2$, or $B_3$ contains a core. We start by describing two preliminary executions that are used to construct the one in which Agreement is violated. We construct executions $E_\alpha$ and $E_\beta$ as follows:

Let $p_{b_1} \in B_1, p_{b_2} \in B_2, p_{b_3} \in B_3, v \in V, v' \in V, v' \neq v$

$$E_\alpha(\langle b_1\rangle) = E_\alpha(\langle b_2\rangle) = E_\alpha(\langle b_3\rangle) = v$$

$$E_\beta(\langle b_1\rangle) = E_\beta(\langle b_2\rangle) = E_\beta(\langle b_3\rangle) = v'$$

Let $\omega \in$ *S-Pid*$(\Pi)$ and $p_i \in \Pi$

$$E_\alpha(\langle i, b_1\rangle \circ \omega) = E_\alpha(\langle b_1\rangle \circ \omega)$$

$$E_\alpha(\langle b_1, b_2\rangle \circ \omega) = E_\alpha(\langle b_2\rangle \circ \omega)$$

$$E_\alpha(\langle b_3, b_2\rangle \circ \omega) = E_\beta(\langle b_2\rangle \circ \omega)$$

$$E_\alpha(\langle i, b_3\rangle \circ \omega) = E_\alpha(\langle b_3\rangle \circ \omega)$$

$$E_\beta(\langle i, b_1\rangle \circ \omega) = E_\beta(\langle b_1\rangle \circ \omega)$$

$$E_\beta(\langle i, b_2\rangle \circ \omega) = E_\beta(\langle b_2\rangle \circ \omega)$$

$$E_\beta(\langle b_1, b_3\rangle \circ \omega) = E_\beta(\langle b_3\rangle \circ \omega)$$

$$E_\beta(\langle b_2, b_3\rangle \circ \omega) = E_\alpha(\langle b_3\rangle \circ \omega)$$

Based on executions $E_\alpha$ and $E_\beta$, we construct execution $E_\gamma$ as follows:

Let $p_{b_1}$, $p_{b_2}$, $p_{b_3}$, $v$, $v'$, $p_i$, and $\omega$ be as in definition of executions $E_\alpha$ and $E_\beta$

$$E_\gamma(\langle b_1\rangle) = v$$

$$E_\gamma(\langle b_2\rangle) = v'$$

$$E_\gamma(\langle b_3\rangle) = v$$

$$E_\gamma(\langle b_2, b_1\rangle \circ \omega) = E_\beta(\langle b_1\rangle \circ \omega)$$

$$E_\gamma(\langle b_3, b_1\rangle \circ \omega) = E_\alpha(\langle b_1\rangle \circ \omega)$$

$$E_\gamma(\langle i, b_2\rangle \circ \omega) = E_\gamma(\langle b_2\rangle \circ \omega)$$

$$E_\gamma(\langle i, b_3\rangle \circ \omega) = E_\gamma(\langle b_3\rangle \circ \omega)$$

It remains to show that $E_\alpha(\langle b_3\rangle \circ \omega) = E_\gamma(\langle b_3\rangle \circ \omega)$ and $E_\beta(\langle b_2\rangle \circ \omega) = E_\gamma(\langle b_2\rangle \circ \omega)$, for $p_{b_2} \in B_2$, $p_{b_3} \in B_3$, and $\omega \in$ *S-Pid*$(\Pi)$. We prove these equivalences with a simple induction on the length of $\omega$.

- Base case: *Len*$(\omega) = 0$

  For *Len*$(\omega) = 0$, we have that $E_\alpha(\langle b_3\rangle) = v = E_\gamma(\langle b_3\rangle)$ and that $E_\beta(\langle b_2\rangle) = v' =$

$E_\gamma(\langle b_2 \rangle)$.

- Induction step: the induction hypothesis is that the proposition is valid for all $\omega$ such that *Len*$(\omega) \leq i$. We need to prove that the proposition is true for all $\omega$ of length $i+1$. That is, we need to show that $E_\alpha(\langle b_3, i\rangle \circ \omega) = E_\gamma(\langle b_3, i\rangle \circ \omega)$ and $E_\beta(\langle b_2, i\rangle \circ \omega) = E_\gamma(\langle b_2, i\rangle \circ \omega)$ for every $p_i \in \Pi$. There are three cases to analyze: $p_i \in B_1$, $p_i \in B_2$, and $p_i \in B_3$. We show below these three cases separately:

1. $p_i \in B_1$: by the definitions of $E_\alpha$, $E_\beta$, and $E_\gamma$:

$$E_\alpha(\langle b_3, i\rangle \circ \omega) = E_\alpha(\langle i\rangle \circ \omega) = E_\gamma(\langle b_3, i\rangle \circ \omega)$$
$$E_\beta(\langle b_2, i\rangle \circ \omega) = E_\beta(\langle i\rangle \circ \omega) = E_\gamma(\langle b_2, i\rangle \circ \omega)$$

2. $p_i \in B_2$: by the definitions of $E_\alpha$, $E_\beta$, and $E_\gamma$ and the induction hypothesis:

$$E_\alpha(\langle b_3, i\rangle \circ \omega) = E_\beta(\langle i\rangle \circ \omega) = E_\gamma(\langle i\rangle \circ \omega) = E_\gamma(\langle b_3, i\rangle \circ \omega)$$
$$E_\beta(\langle b_2, i\rangle \circ \omega) = E_\beta(\langle i\rangle \circ \omega) = E_\gamma(\langle i\rangle \circ \omega) = E_\gamma(\langle b_2, i\rangle \circ \omega)$$

3. $p_i \in B_3$: by the definitions of $E_\alpha$, $E_\beta$, and $E_\gamma$ and the induction hypothesis:

$$E_\alpha(\langle b_3, i\rangle \circ \omega) = E_\alpha(\langle i\rangle \circ \omega) = E_\gamma(\langle i\rangle \circ \omega) = E_\gamma(\langle b_3, i\rangle \circ \omega)$$
$$E_\beta(\langle b_2, i\rangle \circ \omega) = E_\alpha(\langle i\rangle \circ \omega) = E_\gamma(\langle i\rangle \circ \omega) = E_\gamma(\langle b_2, i\rangle \circ \omega)$$

Because processes in $B_3$ cannot distinguish between executions $E_\alpha$ and $E_\gamma$, these processes have to decide $v$ in $E_\gamma$. On the other hand, processes in $B_2$ cannot distinguish execution $E_\beta$ from execution $E_\gamma$, and consequently they have to decide $v'$ in $E_\gamma$. By assumption, in execution $E_\gamma$, the processes in both subset $B_2$ and subset $B_3$ are correct. This implies that $E_\gamma$ violates Agreement.

$\square$

### 3.3.6 An algorithm to solve strong consensus

We describe an algorithm that solves strong consensus in a system with profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$. This algorithm is based on the one described by Lamport *et al.* to show that $3t + 1$ processes ($t$ is the maximum number of faulty processes) is sufficient to solve Interactive Consistency for arbitrarily faulty processes [LSP82]. Different from the one proposed by Lamport *et al.*, we assume a system profile that satisfies Byzantine Intersection. As a consequence, our algorithm does not use a threshold on the number of faulty processes.

Because the formal description of the algorithm is fairly complex, we first present an overview of the algorithm, and then we present a more detailed description.

#### Overview

The algorithm proceeds in rounds of message exchange. In each round, a process receives messages from other processes, and prepare new messages to send to the other processes. Each message a process receives contains a mapping from sequence of process ids to value. A sequence of process ids in such a mapping does not contain repeated ids, and a process $p_i$ interprets an entry $[i_1 \circ i_2 \circ i_3 \mapsto v]$ in a mapping received from process $p_{i_1}$ as the value that $p_{i_3}$ sent to $p_{i_2}$ and $p_{i_2}$ sent to $p_{i_1}$.

Once all the entries a correct process $p_i$ receives are such that the sequence of process ids contains a core, it does the equivalent of mapping the sequences of process ids to nodes of a tree, where the value of a node is the value the sequence maps to. The process then traverses this tree in postorder. When visiting each of the nodes, it determines the value of the node by observing if there is a pair of survivor sets such that the children corresponding to the intersection of this pair contains the same value (leaves maintain their original values). By Byzantine Intersection, such an intersection must contain at least one correct process. A process decides upon the value in the root of this tree.

As we show in the following detailed characterization of the algorithm, in the end of this postorder traversal, every node that contains a sequence such that the last id is the one of a correct process has the same value across all correct processes. This property implies that the root must contain the same value across all the correct processes because the immediate level under the root corresponds to the sequences containing a single process and there is at least one survivor set containing only correct processes by assumption.

**Detailed description**

We call our algorithm SyncByz. In SyncByz, all the processes run the same code, and the algorithm proceeds in rounds of message exchange. A message contains a mapping from sequence of process ids to value. Using $\mathit{Processes}(\omega) = \{p_i : i \in \omega\}$ to denote the set of processes that have their ids in $\omega \in \mathit{S\text{-}Pid}(\Pi)$, such a sequence $\omega$ must also satisfy the following two constraints:

- For every $p_i \in \Pi$, $i$ appears at most once in $\omega$;
- Either $\omega = \langle\rangle$ or $\omega = \langle i \rangle \circ \omega'$, $\omega' \in \mathit{S\text{-}Pid}(\Pi)$, and there exists $S_\omega \in \mathcal{S}_\Pi$ such that $S_\omega \subseteq \Pi \setminus \mathit{Processes}(\omega')$.

We use $\mathit{D\text{-}Pid}(\Pi)$ to denote the subset of $\mathit{S\text{-}Pid}(\Pi)$ that satisfy these two constraints.

In each round a correct process $p_i$ receives a set of messages, each one containing a mapping, and it forwards to process $p_j$ a new mapping containing all the sequences that do not contain $j$. In this new mapping, $p_i$ adds its own id to every sequence that is an element of the domain of the mapping. For example, if $\omega$ and $\omega'$ are two sequences that do not contain $j$, then the mapping $(\mathit{VSeq} : \mathit{seq} \in \mathit{D\text{-}Pid}(\Pi) \mapsto V)$ that $p_i$ sends to

$p_j$ contains $\langle i \rangle \circ \omega \mapsto v$ and $\langle i \rangle \circ \omega' \mapsto v'$, where $\omega \mapsto v$ and $\omega' \mapsto v'$ are in mappings that $p_i$ receives in messages from other processes and $v, v' \in V$.

To illustrate further, consider a system with four processes: $p_{i_1}, p_{i_2}, p_{i_3}, p_{i_4}$. Suppose that process $p_{i_2}$ is correct and it receives in round 2 a message $m$ from process $p_{i_3}$ such that (DOMAIN $m.\mathit{VSet}$) contains the sequence $\langle i_3, i_4 \rangle$.[2] Process $p_{i_2}$ sends a message $m'$ to $p_{i_1}$ in round 2 such that $\langle i_2, i_3, i_4 \rangle \in$ (DOMAIN $m'.\mathit{VSet}$) and $m.\mathit{VSeq}(\langle i_3, i_4 \rangle) = m'.\mathit{VSeq}(\langle i_2, i_3, i_4 \rangle)$.

In the last round, correct processes use a recursive procedure to decide upon a value. This recursive procedure, which we call **DecisionValue**, uses the mappings from sequences of process ids to values received throughout the rounds to compute a decision value. **DecisionValue** computes a final mapping from sequences of process ids to values. To describe in a more abstract fashion how this procedure computes a decision value, we can use a tree, where the nodes are labeled with sequences of process ids. The label of the root is the empty sequence. For each leaf of the tree, assuming $\omega = j\omega'$ is its label, process $p_i$ assigns *VSeq*$(\omega)$ to this leaf if at round *Len*$(\omega)$ $p_i$ receives a message $m$ from $p_j$ such that $\omega \in$ (DOMAIN $m.\mathit{VSet}$).

Once a process assigns values to the leaves, it traverses the tree in post-order. This means that we first visit the children of a node before visiting the node itself, and visiting a node consists in assigning a value to this node. A leaf, when visited, just keeps the value previously assigned as we explained before. When visiting a parent node, if there is a set of ids of its children such that they are ids of processes in the intersection of two survivor sets and the value assigned to these children is the same value, say $v$, then $v$ is also the value the process assigns to the parent node. After visiting all the nodes of the tree, the decision value is the value assigned to the root.

The total number of rounds is $|\Pi| - \min\{x | (x = |S|) \wedge (S \in \mathcal{S}_\Pi)\} + 1$. An important observation is that this algorithm is optimal with respect to the number of rounds, as it matches the lower bound of Section 3.3.2. Comparing with **SyncCrash**, note that we cannot use a single core or a single survivor set as a set of active processes

[2]DOMAIN *f* denotes the domain of an arbitrary mapping *f*.

because the system profile has to satisfy Byzantine Intersection, according to our result in Section 3.3.5.

**Definitions and conventions.** Before presenting the pseudocode and the proof of correctness, we review a few definitions and present new ones. We use $m.\mathit{VSeq}$ to denote the mapping from sequences of process ids to values in message $m$, and $m.\mathit{VSeq}(\omega)$ to denote the value that $m.\mathit{VSeq}$ evaluates to for $\omega$. Similarly, we use $p_i.\mathit{VSeq}$ to denote the mapping from process ids to values of process $p_i$, and $p_i.\mathit{VSeq}(\omega)$ to denote the value that $p_i.\mathit{VSeq}$ evaluates to. In the proofs that follow the algorithm, we also use $p_i.\mathit{VSeqR}(r)$ to denote the state of the *VSeq* mapping of process $p_i$ in the last step of round $r$. Thus, $p_i.\mathit{VSeqR}(\omega, r)$ evaluates to the value of $p_i.\mathit{VSeq}(\omega)$ in the last step of round $r$. If process $p_i$ does not receive any message $m$ such that $\omega \in \mathit{D\text{-}Pid}(\Pi)$ is in $(\text{DOMAIN } m.\mathit{VSeq})$ or it does not assign a value to $\omega$ when executing **DecisionValue** in the last round of an execution, then $p_i.\mathit{VSeq}(\omega)$ is undefined throughout this execution.

Note that $p_i.\mathit{VSeq}(\omega)$ can have different values at different stages of an execution. If $p_i$ receives a message $m$ at round $r$ such that $\omega \in \text{DOMAIN } m.\mathit{VSeq}$, then $p_i.\mathit{VSeq}(\omega)$ evaluates to $m.\mathit{VSeq}(\omega)$ until the execution of the recursive procedure **DecisionValue**. After the execution of **DecisionValue** in the last round, $p_i.\mathit{VSeq}(\omega)$ can evaluate to a value different than the one previously assigned.

As for **SyncCrash**, we use $p_i.M(r)$ to denote the set of messages that process $p_i$ receives by the beginning of round $r$. Because processes can fail in an arbitrary fashion, messages can be malformed. A malformed message is one that satisfies one of the following:

1. It does not follow the message format;
2. Its mapping contains at least one invalid element. An invalid element either maps a sequence to $v \notin V$ or maps a sequence $\omega$ to a value $v \in V$, but $\mathit{Len}(\omega) > r$, $\mathit{Len}(\omega) < r$, or $\omega \notin \mathit{D\text{-}Pid}(\Pi)$.

Henceforth, we assume that $p_i.M(r)$ does not include such malformed messages, as it is straightforward to detect and to discard them.

Figure 3.3 presents the pseudocode for SyncByz. Appendix C contains a specification of SyncByz in TLA+. In the pseudocode, we use the operator CHOOSE of TLA+. The expression $(\text{CHOOSE } x \in X : e)$ evaluates to an element $x$ of the set $X$, chosen deterministically, such that $x$ satisfies the boolean expression $e$.

We now prove that the algorithm SyncByz satisfies the properties of strong consensus. First, we state and prove preliminary lemmas that we use when showing agreement and strong validity SyncByz. For the following lemmas, we assume a system profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$. We use $S_{min}$ to denote a smallest survivor set in $\mathcal{S}_\Pi$. That is, there is no survivor set in $\mathcal{S}_\Pi$ with fewer elements than $S_{min}$.

**Lemma 3.3.23** *Let $E$ be an execution of SyncByz, $p_i$ be a process in $\Pi \setminus \text{Faulty}(E)$, and $\omega \in \text{D-Pid}(\Pi)$ be a sequence such that $i \notin \text{Processes}(\omega)$ and there is at least one survivor set in $\mathcal{S}_\Pi$ contained in $\Pi \setminus \text{Processes}(\omega)$. For every process $p_j \in \Pi \setminus \text{Faulty}(E)$, $j \notin \text{Processes}(\omega)$, $p_j.\mathit{VSeqR}(\langle i \rangle \circ \omega, |\Pi| - |S_{\text{min}}| + 1) = p_i.\mathit{VSeqR}(\omega, \text{Len}(\omega))$.*

**Proof:**

There are two cases to analyze: $j = i$ and $j \neq i$. If $i = j$, then by the algorithm, process $p_i$ adds $[\langle i \rangle \circ \omega \mapsto p_i.\mathit{VSeqR}(\omega, \mathit{Len}(\omega))]$ to $p_i.\mathit{VSeq}$ in round $\mathit{Len}(\omega) + 1$ and it does not change this value when executing recursively procedure **DecisionValue**. This implies that $p_i.\mathit{VSeqR}(\langle i \rangle \circ \omega, |\Pi| - |S_{min}| + 1) = p_i.\mathit{VSeqR}(\omega, \mathit{Len}(\omega))$.

For the case $i \neq j$, let $S_\omega$ be the smallest survivor set in $\mathcal{S}_\Pi$ such that $S_\omega \subseteq \Pi \setminus \mathit{Processes}(\omega)$. We prove this case by induction on the values of $\rho$, where $0 \leq \rho \leq |\Pi \setminus S_\omega|$ and $\mathit{Len}(\omega) = |\Pi \setminus S_\omega| - \rho$.

**Base case.** $\rho = 0$. Because $p_i$ is correct by assumption, it sends a message $m$ to $p_j$ in round $\mathit{Len}(\omega)$ such that $\langle i \rangle \circ \omega \in \text{DOMAIN } m.\mathit{VSeq}$ and $m.\mathit{VSeq}(\langle i \rangle \circ \omega) = p_i.\mathit{VSeqR}(\omega, \mathit{Len}(\omega))$. According to the construction of the mapping $\mathit{VSeq}$ in the last round, process $p_j$ does not change the value of $p_j.\mathit{VSeq}(\langle i \rangle \circ \omega)$ when executing recursively **DecisionValue**. Thus, we have that $p_j.\mathit{VSeqR}(\langle i \rangle \circ \omega, r) = p_i.\mathit{VSeqR}(\omega, \mathit{Len}(\omega))$,

```
**Algorithm** SyncByz **for process** $p_i$:
**Input:** a set of processes $\Pi$; a set of survivor sets $S_\Pi$; an input value $v_i \in V$

**Initialization:**
    $S_{min} = \min\{S : S \in \mathcal{S}_\Pi\}$
    $p_i.VSeq \leftarrow [\langle i \rangle \mapsto v_i]$

**Round** $r = 0$:
    **for** every $p_j \in \Pi \setminus \{p_i\}$, **send**$(p_i.VSeq, p_j)$

**Rounds** $1 \leq r < (|\Pi| - |S_{min}| + 1)$:
    **for** every message $m \in p_i.M(r)$
        $p_i.VSeq \leftarrow p_i.VSeq \cup m.VSeq$
    **for** every $p_j \in \Pi$
        $ToSendVSeq \leftarrow \emptyset$
        **for** every message $m \in p_i.M(r)$
            **for** every $\omega \in (D\text{-}Pid(\Pi) \cap \text{DOMAIN } m.VSeq)$
            such that $(p_j \notin Processes(\omega)) \wedge (\exists S \in \mathcal{S}_\Pi : S \subseteq \Pi \setminus Processes(\omega))$
                $ToSendVSeq \leftarrow ToSendVSeq \cup [\langle i \rangle \circ \omega \mapsto m.VSeq(\omega)]$
        **send** $(ToSendVSeq, p_j)$

**Round** $r = (|\Pi| - |S_{min}| + 1)$:
    **for** every message $m \in p_i.M(r)$
        $p_i.VSeq \leftarrow p_i.VSeq \cup m.VSeq$
    $TmpVSeq \leftarrow p_i.VSeq$
    **DecisionValue**$(TmpVSeq)$
    $p_i.dec \leftarrow p_i.VSeq(\langle\rangle)$

**DecisionValue**$(VSeq \in (D\text{-}Pid(\Pi) \times V \cup \{\perp\}))$
    **if** $\forall \omega \in \text{DOMAIN } VSeq : Len(\omega) = 1$
        **if** $\exists v \in V : S, S' \in \mathcal{S}_\Pi : \forall p_j \in S \cap S' : VSeq(\langle j \rangle) = v$
            $v \leftarrow \text{CHOOSE } v \in V : S, S' \in \mathcal{S}_\Pi : \forall p_j \in S \cap S' : VSeq(\langle j \rangle) = v$
        **else** $v \leftarrow \text{CHOOSE } v \in V : \exists S \in \mathcal{S}_\Pi : \exists p_j \in S : VSeq(\langle j \rangle) = v$
        $p_i.VSeq \leftarrow p_i.VSeq \cup [\langle\rangle \mapsto v]$
        **return**
    **else**
        $(\langle j \rangle \circ \omega) \leftarrow \text{CHOOSE } Sq \in \text{DOMAIN } VSeq : \forall Sq' \in \text{DOMAIN } VSeq : Len(Sq) \geq Len(Sq')$
        $v \leftarrow \perp$
        **if** $\exists v' \in V : S, S' \in \mathcal{S}_\Pi : \forall p_l \in S \cap S' : VSeq(\langle l \rangle \circ \omega) = v'$
            $v \leftarrow \text{CHOOSE } v' \in V : S, S' \in \mathcal{S}_\Pi : \forall p_l \in S \cap S' : VSeq(\langle l \rangle \circ \omega) = v'$
        $p_i.VSeq \leftarrow (p_i.VSeq \setminus [\omega \mapsto p_i.VSeq(\omega) : \omega \in \text{DOMAIN } p_i.VSeq]) \cup [\omega \mapsto v]$
        $IntSet \leftarrow [\langle l \rangle \circ \omega \mapsto VSeq(\langle j \rangle \circ \omega) : \langle l \rangle \circ \omega \in \text{DOMAIN } VSeq] \cup$
                    $[\omega \mapsto VSeq(\omega) : \omega \in \text{DOMAIN } VSeq]$
        $VSeq \leftarrow (VSeq \setminus IntSet) \cup [\omega \mapsto v]$
        **DecisionValue**$(VSeq)$
        **return**
```

**Figure 3.3:** Algorithm SyncByz

$r \geq \textit{Len}(\omega) + 1$.

**Induction step.** We now assume that the proposition holds for every $\rho < |\Pi \setminus S_\omega|$ and we show for $\rho + 1$. If the proposition holds for $\rho$, then for every correct process $p_l$, such that $p_l \notin \textit{Processes}(\langle i \rangle \circ \omega)$, we have that $p_j.\textit{VSeqR}(\langle l, i \rangle \circ \omega, \textit{Len}(\langle l, i \rangle \circ \omega))$ evaluates to $p_i.\textit{VSeqR}(\omega, \textit{Len}(\omega))$. Let $S_c \in \mathcal{S}_\Pi$ be a survivor set containing only correct processes in $E$. Such a survivor set exists by assumption. By the algorithm, we have that $S_c \cap S_\omega \subseteq \Pi \setminus \textit{Processes}(\langle i \rangle \circ \omega)$. Consequently, for every $p_l \in S_c \cap S_\omega$, the value of $p_j.\textit{VSeqR}(\langle l, i \rangle \circ \omega, \textit{Len}(\langle l, i \rangle \circ \omega))$ evaluates to $p_i.\textit{VSeqR}(\omega, \textit{Len}(\omega))$.

By assumption, $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ satisfies Byzantine Intersection. This implies that the intersection of every two survivor sets contains a core. By definition, a core contains at least one correct process in every execution. Thus, there are no two survivor sets $S, S' \in \mathcal{S}_\Pi$, $S \cap S' \cap S_c = \emptyset$, and $v \in V \cup \{\perp\}$, $v \neq p_i.\textit{VSeqR}(\omega, \textit{Len}(\omega))$, such that for every $p_l \in S \cap S'$, the value of $p_j.\textit{VSeqR}(\langle l, i \rangle \circ \omega, \textit{Len}(\langle l, i \rangle \circ \omega))$ evaluates to $v$.

By the algorithm, process $p_j$ constructs the mapping $p_j.\textit{VSeq}$ in the last round such that $p_j.\textit{VSeq}(\langle i \rangle \circ \omega)$ evaluates to $p_i.\textit{VSeqR}(\omega, \textit{Len}(\omega))$ after executing **Decision-Value**. Thus, $p_j.\textit{VSeqR}(\langle i \rangle \circ \omega, |\Pi| - |S_{\textit{min}}| + 1)$ evaluates to $p_i.\textit{VSeqR}(\omega, \textit{Len}(\omega))$.
$\square$

**Lemma 3.3.24** *Let $E$ be an execution of* **SyncByz** *and $(\langle i \rangle \circ \omega)$ be an element of* D-Pid($\Pi$) *such that there is some survivor set in $\mathcal{S}_\Pi$ contained in* $\Pi \setminus$ Processes($\omega$). *If we have that:*

$$\bigwedge \quad \forall \omega' \circ \langle i \rangle \circ \omega \in \text{D-Pid}(\Pi) : \text{Processes}(\langle \omega' \rangle \circ i) \cap (\Pi \setminus \text{Faulty}(E)) \neq \emptyset$$
$$\bigwedge \quad \exists p_{c_1} \in \Pi \setminus (\text{Faulty}(E) \cup \text{Processes}(\omega)) : \exists v \in V : p_{c_1}.\textit{VSeqR}(\langle i \rangle \circ \omega, |\Pi| - |S_{\text{min}}| + 1) = v$$

*then for every correct process $p_{c_2} \in \Pi \setminus$ (Faulty($E$) $\cup$ Processes($\omega$)), $p_{c_2}.\textit{VSeqR}(\langle i \rangle \circ \omega, |\Pi| - |S_{\text{min}}| + 1) = v$.*

**Proof:**

Let $S_\omega \in \mathcal{S}_\Pi$ be a survivor set such that $S_\omega \subseteq \Pi \setminus \textit{Processes}(\omega)$. We prove this lemma by induction on the values of $\rho$, where $0 \leq \rho \leq |\Pi \setminus S_\omega|$ and $\textit{Len}(\omega) = |\Pi \setminus S_\omega| - \rho$.

**Base case.** $\rho = 0$. In this case, $\Pi \setminus \mathit{Processes}(\omega) = S_\omega$. We then have that by assumption $p_i$ is correct, and that $p_{c_1}.\mathit{VSeqR}(\langle i \rangle \circ \omega, |\Pi| - |S_{min}| + 1) = p_{c_2}.\mathit{VSeqR}(\langle i \rangle \circ \omega, |\Pi| - |S_{min}| + 1)$, from Lemma 3.3.23, for every $p_{c_1}, p_{c_2} \in \Pi \setminus (\mathit{Faulty}(E) \cup \mathit{Processes}(\omega))$.

**Induction step.** Suppose that the proposition holds for every $\rho < |\Pi \setminus S_\omega|$. We show for $\rho + 1$. There are two cases to analyze: $p_i$ is correct and $p_i$ is faulty. If $p_i$ is correct, then the proof follows from Lemma 3.3.23. If $p_i$ is faulty, then $\mathit{Processes}(\omega') \cap (\Pi \setminus \mathit{Faulty}(E)) \neq \emptyset$. We then have that for every $p_{c_1}, p_{c_2} \in \Pi \setminus (\mathit{Faulty}(E) \cup \mathit{Processes}(\omega))$, $p_{c_1}.\mathit{VSeqR}(\langle j, i \rangle \circ \omega, |\Pi| - |S_{min}| + 1) = p_{c_2}.\mathit{VSeqR}(\langle j, i \rangle \circ \omega, |\Pi| - |S_{min}| + 1)$ by the induction hypothesis, where $p_j \in \Pi \setminus \mathit{Processes}(\langle i \rangle \circ \omega)$. Because the assignment of values to sequences in the recursive procedure **DecisionValue** is deterministic, we then have that if $p_{c_1}.\mathit{VSeqR}(\langle i \rangle \circ \omega, |\Pi| - |S_{min}| + 1) \in V$, $p_{c_1} \in \Pi \setminus (\mathit{Faulty}(E) \cup \mathit{Processes}(\omega))$, then $p_{c_1}.\mathit{VSeqR}(\langle i \rangle \circ \omega, |\Pi| - |S_{min}| + 1) = p_{c_2}.\mathit{VSeqR}(\langle i \rangle \circ \omega, |\Pi| - |S_{min}| + 1)$, for every $p_{c_2} \in \Pi \setminus (\mathit{Faulty}(E) \cup \mathit{Processes}(\omega))$.

□

**Lemma 3.3.25** *SyncByz satisfies Strong Validity.*

**Proof:**

Let $S_c \in \mathcal{S}_\Pi$ be a survivor set containing only correct processes. By assumption, such a survivor set exists in every execution. By Lemma 3.3.23, for every process $p_c \in S_c$, we have that $p_c.\mathit{VSeqR}(\langle i \rangle, |\Pi| - |S_{min}| + 1)$ is the initial value of $p_i$, assuming $p_i$ is also correct. By the algorithm, to determine the value of $p_c.\mathit{VSeqR}(\langle \rangle, |\Pi| - |S_{min}| + 1)$, process $p_c$ first checks if there is a subset $\Pi'$ of processes in the intersection of two survivor sets, such that for every $p_i \in \Pi'$, $p_c.\mathit{VSeq}(\langle i \rangle) = v$, for some $v \in V$. Otherwise, it chooses deterministically some $v \in V$ such that $p_c.\mathit{VSeq}(\langle i \rangle) = v$, for some $p_i$.

In the first case, by Lemma 3.3.23 and Byzantine Intersection, every correct process decides upon the same value $v \in V$. In the second case, every correct process chooses the same value by Lemma 3.3.24. Thus, if every correct process $p_c$ has the same initial value $v \in V$, then every correct process $p_c$ decides upon the same value $v \in V$.

□

**Lemma 3.3.26** *SyncByz satisfies Agreement.*

**Proof:**

Let $E$ be an execution of SyncByz. We need to show that for every process $p_c \in \Pi \backslash \textit{Faulty}(E)$, $p_c.\textbf{\textit{VSeqR}}(\langle\rangle, |\Pi| - |S_{\textit{min}}| + 1) = v$, for some $v \in V$. By Lemma 3.3.24, for every $p_{i_1} \in \Pi$, we have that if $p_{c_1}.\textbf{\textit{VSeqR}}(\langle i \rangle, |\Pi| - |S_{\textit{min}}| + 1) \in V$, $p_{c_1} \in \Pi \backslash \textit{Faulty}(E)$, then $p_{c_1}.\textbf{\textit{VSeqR}}(\langle i \rangle, |\Pi| - |S_{\textit{min}}| + 1) = p_{c_2}.\textbf{\textit{VSeqR}}(\langle i \rangle, |\Pi| - |S_{\textit{min}}| + 1)$, for every $p_{c_2} \in \Pi \setminus \textit{Faulty}(E)$. By the procedure that assigns a value to $p_c.\textbf{\textit{VSeqR}}(\langle\rangle, |\Pi| - |S_{\textit{min}}| + 1)$, $p_c \in \Pi \setminus \textit{Faulty}(E)$, we have that $p_{c_1}.\textbf{\textit{VSeqR}}(\langle\rangle, |\Pi| - |S_{\textit{min}}| + 1) = p_{c_2}.\textbf{\textit{VSeqR}}(\langle\rangle, |\Pi| - |S_{\textit{min}}| + 1)$, for every $p_{c_1}, p_{c_2} \in \Pi \setminus \textit{Faulty}(E)$. This implies that every correct process decides upon the same value $v$ in $E$.

□

**Lemma 3.3.27** *SyncByz satisfies Termination.*

**Proof:**

In every execution, every correct process decides in a finite number of rounds. We conclude that correct processes eventually decide, thus satisfying Termination.

□

### 3.3.7 Revisiting the lower bound on the number of rounds

For crash failures, a single core is sufficient to solve consensus, as we discussed in Section 3.3.3. Consider a system *sys* = $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ that contains at least one core. Assuming arbitrary failures, a system profile has to satisfy Byzantine Intersection to enable a solution to strong consensus. Because of these different requirements, we have that the lower bound on the number of rounds can be different for the same system profile. Suppose the following profile:

**Example 3.3.28**

$$\Pi = \{p_1, p_2, p_3, p_4, p_5\}$$

$$\begin{aligned}\mathcal{C}_\Pi &= \{\{p_1,p_2,p_3\},\{p_1,p_4\},\{p_1,p_5\},\{p_2,p_4\},\{p_2,p_5\},\{p_3,p_4\},\{p_3,p_5\},\{p_4,p_5\}\} \\ \mathcal{S}_\Pi &= \{\{p_1,p_2,p_3,p_4\},\{p_1,p_2,p_3,p_5\},\{p_1,p_4,p_5\},\{p_2,p_4,p_5\},\{p_3,p_4,p_5\}\}\end{aligned}$$

The smallest cores in this profile have size two. Thus, according to our lower bound result, there are executions with two failures that require at least two rounds, assuming crash failures. This bound is actually tight, as we have shown in Section 3.3.3. For arbitrary failures, we have that the lower bound on the number of rounds is $\Pi - S_{min} + 1 = 5 - 3 + 1 = 3$, and this is also tight from Section 3.3.4. We conclude that for different failure models, the number of rounds necessary and sufficient to solve consensus considering the worst case (maximum number of faulty processes by the system profile and one failure in each round) can be different. This is in contrast with the previous result for consensus under the $t$ of $n$ failure assumption (threshold model), where the lower bound on the number of rounds in the worst case is the same for both crash and arbitrary. It is important to observe that these results do not contradict each other. If all the cores have the same size, then our result shows that the number of rounds necessary and sufficient to enable a solution to consensus is the same.

## 3.4 Asynchronous systems

In this section, we present results for consensus in asynchronous systems using our new model. In asynchronous systems, there is no bound on message latency, clock drift, or processor speed, although message latency is finite and correct processes take steps infinitely often if enabled forever. According to the well-known FLP result, consensus cannot be solved in asynchronous systems even if a single process can crash [FLP85]. A common technique to enable a solution to consensus in such systems is to extend the system with a failure detector [CT96]. A failure detector consists of a collection of modules, one for each process, that provide to processes information about failures. Henceforth, we assume that the output of the failure detector module of a process $p_i$ is the subset of processes that $p_i$ *suspects* to be faulty.

Chandra *et al.* show that $\diamond W$ is the weakest class of failure detectors to enable a solution to consensus [CHT96]. Failure detectors of the class $\diamond W$ satisfy the following two properties:

**Weak Completeness:** Eventually every process that crashes is permanently suspected by some correct process;

**Eventual Weak Accuracy:** There is a time after which some correct process is never suspected by any correct process.

In [CT96], Chandra and Toueg show that the class $\diamond W$ is equivalent to the class $\diamond S$. Failure detectors of the class $\diamond S$ also satisfy both a completeness property and an accuracy property. The accuracy property is Eventual Weak Accuracy as for $\diamond W$, whereas the completeness property is a different one. The completeness property is as follows:

**Strong Completeness:** Eventually every process that crashes is permanently suspected by every correct process.

In Section 3.4.1, we present an algorithm based on the one by Chandra and Toueg that assumes a failure detector of the class $\diamond S$.

When processes fail arbitrarily, it is possible to use a stronger failure detector that identifies faulty processes through different failure modes. For example, if a process sends a malformed message, or if it skips messages, then it must be faulty. Doudou and Schiper propose a class of failure detectors based on the definition of mute processes [DS98]. A mute process is defined as follows:

**Mute process:** Let $p_i$ and $p_j$ be two processes. Process $p_i$ is mute to $p_j$ if there is a time $t$ after which either:

1. $p_i$ crashes;
2. $p_i$ stops forever sending messages to $p_j$.

In [DS98], Doudou and Schiper assume that messages are digitally signed using a public key scheme. If a process incorrectly signs messages it sends to other processes, then these messages are discarded by correct receivers. Thus, in the case that all messages sent by a faulty process $p_i$ to another correct process $p_j$ after some time $t$ are incorrectly signed, $p_j$ eventually declares $p_i$ mute.

Based on the notion of mute processes, the following property replaces strong completeness in [DS98]:

**Mute Completeness:** There is a time $t$ after which every process $p_i$ that is mute to a process $p_j$ is suspected forever by $p_j$.

Failure detectors of the class $\diamond M$ satisfy Eventual Weak Accuracy and Mute Completeness. Section 3.4.2 describes a consensus algorithm that assumes a failure detector of the class $\diamond M$ (processes have a failure detector module such that the collection for these modules satisfy the properties of the $\diamond M$ class).

It is important to observe at this point that other classes of failure detectors for Byzantine failures have been previously proposed. Malkhi and Reiter propose unreliable failure detectors that detect faulty processes that are *quiet* [MR97b], where a process is said to be quiet if it broadcasts only a finite number of messages in an execution. This definition is similar to the one of a mute process, and the difference relies on the assumption that processes have available a primitive for reliably broadcasting messages, and that this primitive also preserves causal order of messages. Kihlstrom *et al.* also propose new classes of failure detectors for Byzantine failures. Different from these two previous approaches, their failure detectors do not include in their output only processes that become silent during an execution, but also processes that present any form of detectable misbehavior [KMMS97]. Delegating the detection of all forms of detectable misbehavior to the failure detector is not strictly necessary, however, as the work by Doudou and Schiper show. We therefore opt for minimal properties of a failure detector that enable a solution to consensus in asynchronous systems with arbitrarily faulty processes.

Because the system model in the remainder of this section is extended with some failure detector, an execution of an algorithm has also to consider the behavior of such failure detectors. Thus, we extend our definition of an execution by adding a mapping from time and process identifiers to subsets of processes:

$$H : \Pi, \mathit{Time} \mapsto 2^{\Pi}$$

The tuple $\langle \mathit{Init}, \mathit{Steps}, \mathit{Time}, \mathit{Faulty}, H \rangle$ then represents an execution in a system model extended with a failure detector, where $H$ is the failure detector history in this execution.

In the remainder of this section, we discuss solutions to consensus assuming two different failure models. In Section 3.4.1, we assume that processes fail by crashing, and in Section 3.4.2, we assume that processes fail arbitrarily.

### 3.4.1 Asynchronous consensus for crash failures

This section discusses consensus in asynchronous systems, assuming that processes fail by crashing. Under the assumption of crash failures, a process executes no further steps once it fails, but all the steps it executes, it executes correctly. We use *crash step* to denote the last step that a faulty process executes. Because a faulty process executes no further steps once it fails in an execution, it must execute a finite number of steps in such an execution. A correct process executes an infinite number of steps if enabled forever.

The structure of this section is identical to one of Section 3.3.4. We first present two properties, Crash Partition and Crash Intersection, that are necessary and sufficient to solve consensus in this model, and show that these two properties are equivalent. Second, we show that these properties are necessary by showing that there is no algorithm that can solve consensus in a system that does not satisfy these properties. Finally, we present an algorithm that solves consensus in systems that satisfy these two properties, thus showing that Crash Partition and Crash Intersection are necessary and sufficient.

As in Section 3.3.4, let $\mathcal{P}_b(\Pi)$ be the set of all partitions of $\Pi$ into $b$ blocks. Given a system profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$, suppose the following properties for this system:

**Property 3.4.1** (**Crash Partition**) $\forall (B_1, B_2) \in \mathcal{P}_2(\Pi) : \exists C \in \mathcal{C}_\Pi : (C \subseteq B_1) \vee (C \subseteq B_2)$ $\square$

**Property 3.4.2** (**Crash Intersection**) $\forall S_{i_1}, S_{i_2} \in \mathcal{S}_\Pi : S_{i_1} \cap S_{i_2} \neq \emptyset$ $\square$

We show that Crash Partition and Crash Intersection are equivalent with the following claim.

**Claim 3.4.3** *Crash Partition $\equiv$ Crash Intersection.*

**Proof:**

**Crash Partition → Crash Intersection.** Proof by contrapositive. Suppose a system $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ such that Crash Intersection does not hold for $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$. This implies that there is a pair of survivor sets $S_{i_1}, S_{i_2} \in \mathcal{S}_\Pi$ such that $S_{i_1} \cap S_{i_2} = \emptyset$. We need to construct a partition of $\Pi$ into two blocks such that no block contains a core.

Suppose the following partition:

$$\begin{aligned} B_1 &= \Pi \setminus S_{i_1} \\ B_2 &= \Pi \setminus (S_{i_2} \cup B_1) \end{aligned}$$

It is straightforward to see that $B_1$ and $B_2$ are disjoint sets (no element of $B_1$ can be in $B_2$). We have to show now that $B_1 \cup B_2 = \Pi$.

$$\begin{aligned} B_1 \cup B_2 &= \Pi \setminus (S_{i_1} \cap (S_{i_2} \cup B_1)) \\ &= \Pi \end{aligned} \tag{3.7}$$

Note that $S_{i_1}$ does not have any process in common with $S_{i_2}$ by assumption, and no process in common with $B_1$ by construction. By definition, if a block does not contain at least one element from some survivor set, then it does not contain a core. As both of the blocks do not contain elements from some survivor set, no block contains a core.

**Crash Intersection → Crash Partition.** Proof by contrapositive. Let $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ be a system profile such that there is some partition $(B_1, B_2) \in \mathcal{P}_2(\Pi)$ such that none of $B_1$ and $B_2$ contains a core. If $B_1$ does not contain a core, then $B_2$ contains a survivor set $S_{i_2}$. Similarly, if $B_2$ does not contains a core, then $B_1$ contains a survivor set $S_{i_1}$. This implies that $S_{i_1} \cap S_{i_2} = \emptyset$, and $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ does not satisfy Crash Intersection.

$\square$

**Lower bound on process replication**

In [CT96], Chandra and Toueg show that $n > 2 \cdot t$ is necessary to solve consensus, assuming a threshold $t$ on the number of process failures and a failure detector of the class $\diamond S$ [CT96]. In our core/survivor set model, the Crash Intersection (Crash Partition) property gives a condition for a set of processes that is necessary to enable a solution to consensus. We now present a proof that this property is necessary. First, we discuss the proof at a high level.

Suppose there is an algorithm $\mathcal{A}$ that solves consensus in a system $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ such that there is a partition $(B_1, B_2)$ of the processes in $\Pi$ such that neither $B_1$ nor $B_2$ contains a core. We build an execution in which the Agreement property is violated, no matter what the algorithm does. We build two preliminary executions of $\mathcal{A}$, $E_\alpha$ and $E_\beta$, in the process of building an execution $E_\gamma$ that violates Agreement.

**Execution $E_\alpha$.** All the processes in $B_1$ are correct and the processes in $B_2$ crash before sending a single message. Also, every process in $B_1$ suspects forever every process in $B_2$, starting at time $t = 0$. From the Termination property, every process in $B_1$ eventually decides, and they all have to decide upon the same value $v$ in order to satisfy Agreement. If all the processes in $B_1$ have the same initial value $v_1$, then we have that $v = v_1$ by Validity.

**Execution $E_\beta$.** All the processes in $B_2$ are correct, all the processes in $B_1$ crash before sending a single message, and a process in $B_2$ suspects every process in $B_1$ starting at $t = 0$. We assume also that all the processes in $B_2$ have the same initial value $v_2$, and $v_2 \neq v_1$. Again from the three consensus properties, every correct process $p_i \in B$ eventually decides, and every process $p_i \in B_2$ decides upon $v' = v_2$.

**Execution $E_\gamma$.** Every process in $\Pi$ is correct. We build execution $E_\gamma$ such that it looks the same as $E_\alpha$ for the processes in $B_1$, and the same as $E_\beta$ for the processes in $B_2$. The initial value for every process in $B_1$ is $v_1$ and for every process in $B_2$ is $v_2$. Let $t_1$ be the time by which all processes in $B_1$ have decided in $E_\alpha$, and $t_2$ the time by which all processes in $B_2$ have decided in $E_\beta$. We use $t_1$ and $t_2$ to determine the schedule of messages and the failure detector history. The messages sent between every pair of processes in $B_1$ are scheduled as in $E_\alpha$, whereas the messages between every pair of processes in $B_2$ are scheduled as in $E_\beta$. The messages from processes in $B_1$ to processes in $B_2$, and from processes in $B_2$ to processes in $B_1$ are only delivered after some time $t > \max\{t_1, t_2\}$. The failure detector history follows the same pattern. For the processes in $B_1$, the failure detector history is the same as in $E_\alpha$ up to time $t_a$. Processes in $B_2$ have the same history as in $E_\beta$ up to time $t_b$.

Considering the previous definitions for executions $E_\alpha$, $E_\beta$, and $E_\gamma$, processes in $B_1$ and processes in $B_2$ cannot distinguish executions $E_\alpha$ and $E_\beta$, respectively, from execution $E_\gamma$. Hence, processes in $B_1$ decide $v_1$, and processes in $B_2$ decide $v_2$ in $E_\gamma$. Execution $E_\gamma$ therefore violates Agreement independently of what algorithm $\mathcal{A}$ does.

We now prove our proposition more formally.

**Claim 3.4.4** Let $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ be a system extended with a failure detector of the class $\diamond S$. If consensus is solvable in $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$, then $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ satisfies Crash Partition.

**Proof:**

We prove this theorem by contradiction. Suppose that there is some algorithm $\mathcal{A}$ that solves consensus in $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$, and $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ does not satisfy Crash Partition. That is, there is at least one partition $(B_1, B_2)$ of the processes in $\Pi$, such that none of $B_1$

or $B_2$ contains a core. We show that there is an execution $E_\gamma$ in which Agreement is violated.

We define first two other executions, $E_\alpha$ and $E_\beta$ that we use to build $E_\gamma$. Let $E_\alpha = \langle \mathit{Init}_\alpha, \mathit{Steps}_\alpha, \mathit{Time}_\alpha, \mathit{Faulty}_\alpha, H_\alpha \rangle$ be as follows:

$$\begin{aligned} \mathit{Faulty}_\alpha(s) &= B_2, \forall s \in \mathit{Steps}_\alpha \\ H_\alpha(t,i) &= B_2, \forall t \geq 0, \forall p_i \in B_1 \\ \mathit{Init}_\alpha(i) &= v_1, v_1 \in V, \forall p_i \in B_1 \end{aligned}$$

There is a finite time $t_1$ such that for every $p_i \in \Pi \setminus \mathit{Faulty}(E_\alpha)$, there is a step $s \in \mathit{Steps}_\alpha$ of $p_i$ in which $p_i$ decides, $\mathit{Time}_\alpha(s) \leq t_1$. By assumption, algorithm $\mathcal{A}$ solves consensus and therefore it has to satisfy Termination (every correct process eventually decides). Thus, such a $t_1$ must exist.

Now let $E_\beta = \langle \mathit{Init}_\beta, \mathit{Steps}_\beta, \mathit{Time}_\beta, \mathit{Faulty}_\beta, H_\beta \rangle$ be as follows:

$$\begin{aligned} \mathit{Faulty}_\beta(s) &= B_1, \forall s \in \mathit{Steps}_\beta \\ H_\beta(t,i) &= A, \forall t \geq 0, \forall p_i \in B_2 \\ \mathit{Init}_\beta(i) &= v_2, \forall p_i \in B_2, v_2 \in V, v_2 \neq v_1 \end{aligned}$$

By the same argument presented for execution $E_\alpha$, there must be a time $t_2$ such that, for every $p_i \in \Pi \setminus \mathit{Faulty}(E_\beta)$, there is a step $s \in \mathit{Steps}_\beta$ of $p_i$ in $\mathit{Steps}_\beta$ in which $p_i$ decides, $\mathit{Time}_\beta(s) \leq t_2$.

Let $t'$ be equal to $\max\{t_1, t_2\}$. We then define execution $E_\gamma = \langle \mathit{Init}_\gamma, \mathit{Steps}_\gamma, \mathit{Time}_\gamma, \mathit{Faulty}_\gamma, H_\gamma \rangle$ as follows:

$$\begin{aligned} \mathit{Faulty}_\gamma(s) &= \emptyset, \forall s \in \mathit{Steps}_\gamma \\ H_\gamma(t,i) &= \begin{cases} H_\beta(t,i) & \forall t \leq t', p_i \in B_2 \\ H_\alpha(t,i) & \forall t \leq t', p_i \in B_1 \\ \emptyset & \forall t > t', \forall p_i \in \Pi \end{cases} \\ \mathit{Init}_\gamma(i) &= \begin{cases} v_1 & , \forall p_i \in B_1 \\ v_2 & , \forall p_i \in B_2 \end{cases} \end{aligned}$$

$\mathit{Steps}_\gamma$ and $\mathit{Time}_\gamma$ are as follows:

- For every $s_a \in \textit{Steps}_\alpha$ such that $\textit{Time}_\alpha(s_a) \leq \max\{t_1, t_2\}$, we have that $s_a \in \textit{Steps}_\gamma$ and $\textit{Time}_\gamma(s_a) = \textit{Time}_\alpha(s_a)$;

- For every $s_b \in \textit{Steps}_\beta$ such that $\textit{Time}_\beta(s_b) \leq \max\{t_1, t_2\}$, we have that $s_b \in \textit{Steps}_\gamma$ and $\textit{Time}_\gamma(s_b) = \textit{Time}_\beta(s_b)$;

- If $s \in \textit{Steps}_\gamma$ and $\textit{Time}_\gamma(s) < \max\{t_1, t_2\}$, then either $s \in \textit{Steps}_\alpha$ or $s \in \textit{Steps}_\beta$. If $s \in \textit{Steps}_\alpha$, then $\textit{Time}_\alpha(s) = \textit{Time}_\gamma(s)$, otherwise $\textit{Time}_\beta(s) = \textit{Time}_\gamma(s)$;

- Let $s \in \textit{Steps}_\gamma$ be a step in which a process $p_i \in B_1$ receives a message from a process $p_j \in B_2$. We have that for every such a step, $\textit{Time}_\gamma(s) > \max\{t_1, t_2\}$;

- Let $s \in \textit{Steps}_\gamma$ be a step in which a process $p_i \in B_2$ receives a message from a process $p_j \in B_1$. We have that for every such a step, $\textit{Time}_\gamma(s) > \max\{t_1, t_2\}$;

A process $p_i \in B_1$ cannot distinguish execution $E_\alpha$ from execution $E_\gamma$, whereas process $p_j \in B_2$ cannot distinguish execution $E_\beta$ from execution $E_\gamma$. Thus, $p_i$ and $p_j$ have to decide upon $v_1$ and $v_2$, respectively, therefore violating Agreement. We conclude that $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ must satisfy Crash Partition to enable a solution to consensus.

$\square$

**An algorithm to solve consensus**

As discussed before, consensus is not solvable in a purely asynchronous system. An approach to overcome this impossibility is to extend the asynchronous model with a failure detector. Here we assume a failure detector $\mathcal{D}$ of the class $\diamond S$, which satisfies the strong completeness and eventual weak accuracy properties. The algorithm we describe, called AsyncCrash, uses this failure detector to guarantee liveness (correct processes eventually decide).

As the algorithm proposed by Chandra and Toueg [CT96], our algorithm AsyncCrash is based on the rotating coordinator paradigm and proceeds in asynchronous rounds. In every asynchronous round, one process is assigned as the coordinator of that round. This assignment of processes is deterministic and hardcoded, and the only requirement on it is that processes must be assigned as coordinators infinitely often. In AsyncCrash, we use the round number modulo the number of processes to determine which process is the coordinator of a particular round.

The coordinator of a round gathers the estimates of some survivor set $S \in \mathcal{S}_\Pi$ and chooses a value out of the ones received from these processes. In the algorithm, the coordinator chooses the value from the process that updated it in the latest round among all the estimates received from the processes in $S$. Once the coordinator chooses a value, it sends a message to inform all the processes of its estimate. A process that receives this message from the coordinator "echoes" the coordinator estimate to all the other processes. A process decides as soon as it receives an echo from all the processes in some survivor set $S' \in \mathcal{S}_\Pi$, not necessarily the same as $S$.

So far in this informal discussion, we assumed that the coordinator is correct. If the coordinator crashes and no correct process receives an estimate from the coordinator, then eventually all the processes in some survivor set containing only correct processes suspect that the coordinator has crashed (strong completeness). Once a process $p_i$ suspects that the coordinator of its current round has failed, $p_i$ sends a message to all the other processes suggesting the others to proceed to the next round. If a process receives a message to move on from all the processes in some survivor set, then it re-initializes its variables and proceeds to the next round.

The use of echo messages is not really necessary, but it may anticipate decision when the coordinator $c_r$ of round $r$ crashes in $r$ and at least one correct process, say $p_i$, receives either a message from the coordinator or an echo message from some other process $p_j$. The echo messages from $p_i$ cause other processes to send echo messages as well, and eventually non-crashed processes executing round $r$ decide. Without the echo messages, every non-crashed process would need to wait until all the processes in some survivor set containing only correct processes suspect the coordinator and send messages requesting them to move on to the next round. Furthermore, decision would be postponed, thereby delaying termination. Because the time to suspect the coordinator may be arbitrarily long, this mechanism prevents unnecessary wait in making a decision. Therefore, the argument in favor of echo messages is not correctness, since it is not hard to modify the algorithm to work without it. Its use, however, may reduce the latency in reaching agreement among the correct processes in a real implementation. Schiper proposed originally the utilization of echo messages as an optimization to have a coordinator-based algorithm less dependent on the coordinator in an asynchronous round [DS98, Sch97].

Figure 3.4 shows the pseudocode of AsyncCrash and Table 3.2 presents a brief description of the variables used in the algorithm. We also present an specification of AsyncCrash in TLA+ in Appendix D.

Every process executes the same code in a run of the algorithm, although processes have different roles in a round. The algorithm is structured in stages, and every process initiates an execution in stage *StartRound*. In the first round, round $1$, $p_1$ is the coordinator. After sending an **Estimate** message to itself, it transitions to a different stage, from *StartRound* to *WaitEstimates*. Once it receives an **Estimate** message from every process in some survivor set, then it sends a **CoordEstimate** message with its proposed value to all the processes. After sending **CoordEstimate** messages, the coordinator transitions to stage *WaitCoordEstimate* and behaves as the other processes for the rest of this round. Note that there is a trivial optimization that can be implemented in this case: the coordinator can process the estimate and move directly to stage *WaitEchoes*. For exposition purposes, the pseudocode does not incorporate this optimization.

The other processes proceed to stage *WaitCoodEstimate* after sending an **Estimate** message in stage *StartRound*. In stage *WaitCoordEstimate*, a process waits for either an estimate from the coordinator or an **Echo** message from some other process. Once it receives such a message, it transitions to stage *WaitEchoes*. In stage *WaitEchoes*, a process waits until it receives **Echo** messages from every process in some survivor set $S \in \mathcal{S}_\Pi$. By receiving **Echo** messages from the processes in $S$, a process $p_i$ decides (process $p_i$ can also decide by receiving a decide message). If enough processes suspect the coordinator and send **MoveOn** messages, then $p_i$ proceeds to the next round upon reception of these **MoveOn** messages. It is important to observe that, by strong completeness, a correct process eventually suspects a faulty coordinator, thus preventing cases in which this correct process neither decides nor proceeds to the next round. By assumption, every process has a failure detector module that it consults to determine whether it suspects the coordinator or not. In the pseudocode, the output of the module appears in the form of a predicate which determines when a process should send **MoveOn** messages. More specifically, a process $p_i$ sends **MoveOn** messages if the predicate "**upon suspicion** of $p_c$ and (*Stage* = *WaitCoordEstimate*)" holds ($p_i$ suspects the coordinator and it is in stage *WaitCoordEstimate*, then it sends **MoveOn** messages).

**Table 3.2:** Variables used in the algorithm AsyncCrash

| | |
|---|---|
| *Stage* | Indicates the stage the process is in the current round. |
| *Echoes* | Set with **Echo** messages received in the current round. |
| *Estimate* | Current estimate of process $p_i$. |
| *EstUpdate* | Round in which *Estimate* is updated. |
| *CurEstimates* | Set with the **Estimate** messages received by the coordinator. |
| *c* | Process id of the coordinator of the current round. |
| *r* | Keeps track of the current round. |

A process sends **MoveOn** messages to other processes at most once. That is, once a non-faulty process $p_i$ suspects the coordinator and sends **MoveOn** messages to other processes in a round $r$, the predicate "**upon suspicion** of $p_c$ and (*Stage* = *WaitCoordEstimate*)" is false for the subsequent steps of $p_i$ in round $r$.

The failure detector $\mathcal{D}$ may also falsely suspect the coordinator. In this case, a process either collects enough messages to move on to the next round, or receives enough **Echo** messages to decide. Before moving on to the following round, a process increments the round number, assigns a new coordinator, and moves on the next round by transitioning back to stage *StartRound*. This procedure is repeated until all the correct processes decide.

We now provide a proof of correctness for the algorithm AsyncCrash. Before

```
Algorithm AsyncCrash for process $i$:
Input: set $\Pi$ of processes; set $\mathcal{C}_\Pi$ of cores; set $\mathcal{S}_\Pi$ of survivor sets; initial value $v_i \in V$
Variables: Stage ← StartRound; Echoes ← $\emptyset$; CurEstimates ← $\emptyset$; Estimate ← $v_i$;
        EstUpdate ← $0$; $r \leftarrow 1$; $c \leftarrow 1$
Stages: {StartRound, WaitEstimates, WaitCoordEstimate, WaitEchoes}

Transitions:
upon (Stage = StartRound)
  Send(Estimate, $i$, $r$, Estimate, EstUpdate) to the coordinator $p_c$
  if($c = i$) then Stage ← WaitEstimates
  else Stage ← WaitCoordEstimate

upon reception of (Estimate, $j$, $r$ , $v_j$, $r_j$) and (Stage = WaitEstimates)
  CurEstimates ← CurEstimates $\cup\{(j, v_j, r_j)\}$
  if($\exists S \in \mathcal{S}_\Pi$: $\forall p_k \in S$: ($k$,$v_k$, $r_k$) $\in$ CurEstimates)
  then $r_k \leftarrow \max\{r_x | (x, v_x, r_x) \in$ CurEstimates$\}$
      Estimate ← $v_k$, $(x, v_k, r_k) \in$ CurEstimates; EstUpdate ← $r$
      Send(CoordEstimate, $i$, $r$, Estimate) to all processes in $\Pi$
      Stage ← WaitCoordEstimate

upon reception of (Type, $j$, $r$, $v_j$), Type $\in$ { CoordEstimate, Echo }, and (Stage =
        WaitCoordEstimate)
  if(Type = Echo) then Echoes ← Echoes $\cup\{$(Echo, $j$, $r$, $v_j$)$\}$
  Send(Echo, $j$, $r$, $v_j$) to all processes in $\Pi$
  if ($c \neq i$) then Estimate ← $v_j$; EstUpdate ← $r$
  Stage ← WaitEchoes

upon reception of (Echo, $j$, $r$, $v_j$) and (Stage = WaitEchoes)
  Echoes ← Echoes $\cup\{$(Echo, $j$, $r$, $v_j$)$\}$
  if($\exists S \in \mathcal{S}_\Pi$ such that $\forall p_k \in S$, (Echo, $k$, $r$ , $v$) $\in$ Echoes, $v \in V$) then
    Decide upon value $v$
    Send(Decide, $i$, $v$) to all processes in $\Pi$
    halt

upon suspicion of $p_c$ and (Stage = WaitCoordEstimate)
  Send(MoveOn, $j$, $r$) to all processes in $\Pi$

upon reception of (MoveOn, $j$, $r$)
  MoveOn ← MoveOn $\cup$ (MoveOn, $j$, $r$)
  if ($\exists S \in \mathcal{S}_\Pi : \forall p_k \in S$ : (MoveOn, $k$, $r$) $\in$ MoveOn) then
    $r \leftarrow r + 1$; $c \leftarrow ((c + 1) \bmod |\Pi|) + 1$
    Echoes ← $\emptyset$; MoveOn ← $\emptyset$; CurEstimates ← $\emptyset$
    Stage ← StartRound

upon reception of (Decide, $j$, $v$) and (Stage $\neq$ Decided)
  Decide upon value $v$
  Send(Decide, $i$, $v$) to all processes in $\Pi$
  halt
```

**Figure 3.4:** Algorithm AsyncCrash

stating and proving the theorems that actually show that AsyncCrash satisfy the three consensus properties, we show some preliminary lemmas.

**Lemma 3.4.5** *Let $E$ be an execution of AsyncCrash and $p_i$ be some correct process that does not decide in round $r$, $r > 0$. Eventually $p_i$ moves on to round $r + 1$.*

**Proof:**

Proof by contradiction. Suppose a round $r$ of $E$ such that $p_i$ does not decide and $p_i$ never proceeds to round $r + 1$. If a process $p_i$ does not decide in round $r$, then it neither receives a **Decide** message nor receives an **Echo** message from all processes in some survivor set. If $p_i$ does not receive a **Decide** message, then there is no $p_j$ such that $p_j$ received an **Echo** message from all processes in some survivor set.

By assumption, at least one survivor set $S \in \mathcal{S}_\Pi$ contains only correct processes, and every message sent by a correct process to another process is eventually received. According to the algorithm, the processes in $S$ send an **Echo** message upon reception of either the first **Echo** message or a **CoordEstimate** message. If none of these messages is received by any of the processes in $S$, then the coordinator is faulty. Eventually the elements of $S$ suspect the coordinator and send **MoveOn** messages, by the strong completeness property of the failure detector. Once process $p_i$ receives a **MoveOn** message from every process $p_j \in S$, $p_i$ proceeds to round $r + 1$, a contradiction.

□

For the following lemma, we say that the coordinator of a round $r$ proposes value $v$ in round $r$, if the coordinator sends at least one **CoordEstimate** message proposing $v$ in this round.

**Lemma 3.4.6** *Let $E$ be an execution of AsyncCrash and $p_i$ be a non-faulty process in round $r$ of $E$. If $p_i$ receives an* **Echo** *message $m$ from process $p_j$ in round $r$, then the estimate $v$ of $m$ is the value proposed by the coordinator of $r$.*

**Proof:**

Proof by induction. By the algorithm, if $p_i$ receives an **Echo** message from $p_j$, then there is a sequence of processes $p_{j_1}, p_{j_2}, \ldots, p_{j_\rho}$ ($j_1 = i$, $j_2 = j$, $\rho \geq 2$) such that $p_{j_k}$ sends an **Echo** message by receiving an **Echo** message from $p_{j_{k+1}}$, $p_{j_\rho}$ sends **Echo** messages by receiving a **CoordEstimate** from the coordinator, and $j_k \neq j_{k'}$, $k \neq k'$. We show the proposition with an induction on the value of $\rho$.

**Base case:** $\rho = 2$. Process $p_j = p_{j_2}$ receives a **CoordEstimate** message from the coordinator. The **Echo** messages $p_j$ sends to other processes, including $p_i$, contain as their estimate the value $v$ received in the **CoordEstimate** message. This implies that the **Echo** message $p_i$ receives from $p_j$ contains the value $v$ proposed by the coordinator.

**Induction step.** We assume that the proposition holds for $\rho \geq 2$, and show for $\rho + 1$. By the algorithm, the **Echo** messages $p_{j_{\rho+1}}$ sends to other processes contains the estimate it receives in a **CoordEstimate** messages from the coordinator of round $r$. This implies that the **Echo** messages that $p_{j_\rho}$ send contain the value proposed by the coordinator. By the induction hypothesis, if $p_{j_\rho}$ sends **Echo** messages that contain the estimate of the coordinator of round $r$, then the estimate in the **Echo** message $p_i$ receives from $p_j$ must be the value $v$ proposed by the coordinator of round $r$. This concludes the proof of the induction step and of the lemma.

□

**Lemma 3.4.7** *Let $E$ be an execution of AsyncCrash and $r$ be the first asynchronous round in which some process $p_i$ decides. If $p_i$ decides upon value $v$ and the coordinator of round $r'$ proposes $v'$, $r' > r$, then $v' = v$.*

**Proof:**

We prove this lemma by induction on the round numbers $r'$.

**Base case:** $r' = r+1$. By assumption, we have that some process $p_i$ decides in round $r$. Suppose without loss of generality that $p_i$ decides by receiving **Echo** messages from a survivor set $S \in \mathcal{S}_\Pi$. (Process $p_i$ can also decide by receiving a **Decide** message, which implies that there is some other process $p_j$ that decides in round $r$ by receiving **Echo**

messages from some survivor set.) A non-faulty process $p_j$ sends an **Echo** message to all the processes, including itself, upon reception of either a **CoordEstimate** or an **Echo** message for the first time from some other process. Moreover, $p_j$ updates its estimate upon reception of the first **Echo** message. Thus, if $p_j$ does not crash in round $r+1$, then it sends an **Echo** message and updates its estimate. From Lemma 3.4.5, every correct process that does not decide in round $r$ eventually moves on to round $r+1$.

Suppose that the coordinator of $r+1$ proposes value $v'$. In the beginning of round $r+1$, the coordinator waits for the estimate of all the processes in some survivor set $S' \in \mathcal{S}_\Pi$. Upon reception of all the **Estimate** messages sent by processes in $S'$, the coordinator chooses the estimate updated in the latest round. By the Crash Intersection property, there is at least one process $p_j \in S'$ such that $p_j$'s estimate is $v$ and it is updated in round $r$. Consequently, the coordinator of $r+1$ chooses $v' = v$ as its estimate.

**Induction step.** Suppose that the proposition is true for every $r' > r$. We prove the proposition for $r'+1$. If the coordinator of round $r'+1$ proposes $v'$, then it received **Estimate** messages from a survivor set of processes. By the algorithm, the coordinator chooses $v'$ as the value in the estimate with highest round number. Suppose that this round number is $r''$. By the Crash Intersection property, $r'' \geq r$. If $r'' = r$, then $v = v'$ by assumption. Otherwise, $v' = v$ by the induction hypothesis.

$\square$

**Lemma 3.4.8** *Let $E$ be an execution of AsyncCrash and $r$ be the first round of $E$ such that some process $p_i$ decides in $r$. If $p_i$ decides upon $v \in V$, then $v$ is the value proposed by the coordinator of $r$.*

**Proof:**

By the algorithm, $p_i$ decides by either receiving **Echo** messages from some survivor set $S \in \mathcal{S}_\Pi$ or receiving a **Decide** message. In the first case, the estimate in each of these **Echo** messages is the value $v'$ proposed by the coordinator of $r$ (Lemma 3.4.6). By the algorithm, the value $v$ that $p_i$ decides upon must be equal to $v'$, the value proposed by the coordinator.

If $p_i$ decides by receiving a decide message from a process $p_j$, then there is a sequence of processes $p_{j_1}, p_{j_2}, \ldots, p_{j_r}$ ($j_1 = i$, $j_2 = j$, $r \geq 2$), such that $p_{j_k}$ sends a **Decide** message due to the reception of a **Decide** message from process $p_{j_{k+1}}$, $p_{j_r}$ sends **Decide** messages due to the reception of **Echo** messages from some survivor set, and $j_k \neq j_{k'}$, $k \neq k'$. By the algorithm, the **Decide** messages sent by $p_{j_k}$ contain the decision value in the **Decide** message received from process $p_{j_{k+1}}$. By Lemma 3.4.6, we conclude that $p_{j_r}$ decides upon the value $v'$ proposed by the coordinator, and consequently $p_i$ also decides upon the value $v = v'$ proposed by the coordinator.

□

**Lemma 3.4.9** *Let $E$ be an execution of AsyncCrash and $p_i$ be some correct process that decides in round $r$. Process $p_i$ decides upon the value $v \in V$ proposed by the coordinator of round $r' \leq r$.*

**Proof:**

Let $v'$ be the value proposed by the coordinator of round $r$. Process $p_i$ decides either when it receives an **Echo** message from every process in some survivor set $S \in \mathcal{S}_\Pi$ or when it receives a **Decide** message from some other process.

If $p_i$ decides upon $v$ in round $r$ by receiving **Echo** messages from a survivor set $S \in \mathcal{S}_\Pi$, then $v$ must be the value proposed by the coordinator of round $r$ (Lemma 3.4.6).

If $p_i$ decides upon $v$ in round $r$ by receiving a **Decide** message from some process $p_j$, then $v$ must be a value proposed by the coordinator of some round $r' \leq r$. We show this claim with a simple induction on the sequence of processes that send **Decide** messages and cause the reception of a **Decide** message by $p_i$. This sequence is composed of processes $p_{j_1}, p_{j_2}, \ldots, p_{j_r}$ ($j_1 = i$, $j_2 = j$, and $r \geq 2$), such that $p_{j_k}$ decides and sends **Decide** messages by receiving a **Decide** message from process $p_{j_{k+1}}$, $p_{j_r}$ decides and sends **Decide** messages by receiving in round $r'$ **Echo** messages from some survivor set $S \in \mathcal{S}_\Pi$, and $j_k \neq j_{k'}$, $k \neq k'$. By Lemma 3.4.6, we conclude that the **Echo** messages received by process $p_{j_r}$ contain the value $v'$ proposed by the coordinator of round $r$ as their estimate. This implies that $p_i$ decides upon $v = v'$.

□

**Lemma 3.4.10** *Let $E$ be an execution of* AsyncCrash*. For every process $p_i$, if $p_i$ updates its estimate in round $r$, then it does so with the initial value of some process $p_j \in \Pi$.*

**Proof:**

We prove this lemma with an induction on the round numbers $r$.

**Base case:** $r = 1$. From the algorithm, there are two possibilities for a process $p_i$ to update its estimate. First, if $i = 1$ ($p_i$ is the coordinator), then it receives an **Estimate** message from every process in some survivor set $S \in \mathcal{S}_\Pi$. Because this is the first round, all the **Estimate** messages contain initial values. More specifically, if process $p_j$ sends out an **Estimate** message, then this message contains the initial value of $p_j$. Thus, the coordinator $p_1$ chooses arbitrarily among the **Estimate** messages, since they all indicate that the last update occurred in round zero, and updates its estimate accordingly. For the second case, $p_i$ is not the coordinator. Suppose that $p_i$ updates its estimate in round 1. There are two possibilities:

1. Process $p_i$ receives a **CoordEstimate** message before receiving any **Echo** message. In this case, process $p_i$ updates its estimate with the value proposed by the coordinator $p_1$;

2. Process $p_i$ receives an **Echo** message before it receives a **CoordEstimate** message from the coordinator. In this case this **Echo** message also contains the estimate of the coordinator, by Lemma 3.4.6, and consequently $p_i$ updates its estimate with the value proposed by the round coordinator.

**Induction step.** Now suppose that $r > 1$ and that the proposition is true for every round $r' < r$. We show that the proposition is also true for $r$. There are two cases: 1)

$p_i$ is the coordinator of round $r$; 2) $p_i$ is not the coordinator of round $r$. First, suppose that $p_i$ is the coordinator of round $r$. Process $p_i$ then updates its estimate based on the values received in the **Estimate** messages sent by every process in some survivor set $S \in \mathcal{S}_\Pi$. By the induction hypothesis, the estimate of every process $p_j$ in $S$ has as its estimate the initial value of some process. For every $p_j \in S$, if $p_j$ has not updated its estimate in any previous round, then its estimate is still its initial value $v_j$. Otherwise, from the inductive assumption, $p_j$ has as its estimate the initial value of some process. Consequently, $p_i$ updates its estimate with the initial value of some process. In the case $p_i$ is not the coordinator, it updates its estimate if and only if it receives a **CoordEstimate** message or at least one **Echo** message. If $p_i$ updates its estimate using the value sent in a **CoordEstimate** message, then this value must be the initial value of some process because the coordinator updates its estimate with the initial value of some process by the previous argument (the argument for $p_i$ being the coordinator). If $p_i$ receives an **Echo** message from some other process $p_j$, then the value in this message must be the initial value of some process as it contains the estimate of the coordinator, by Lemma 3.4.6.
□

**Lemma 3.4.11** *Let $E$ be an execution of AsyncCrash. Every $p_i \in \Pi \setminus \text{Faulty}(E)$ eventually decides in $E$.*

**Proof:**

From Lemma 3.4.5, every correct process that does not decide in a round $r$, $r > 0$, proceeds to the next round. A process proceeds to the next round by receiving one **MoveOn** message from every process $p_j$ in some survivor set $S \in \mathcal{S}_\Pi$. According to the algorithm, a non-faulty process $p_i$ sends a **MoveOn** message to all the other processes when it detects that the coordinator $p_c$ has failed. From the eventual weak accuracy property of the failure detector, however, there is a time $t$ after which there is some correct process $p_c$ that is permanently not suspected by any other correct process. Therefore, there is time $t' > t$ and a round $r$ such that $p_c$ is the coordinator of $r$ and no process proceeds to round $r$ before $t'$. By assumption, no non-faulty process suspects $p_c$

in round $r$. Consequently, no non-faulty process sends **MoveOn** messages in this round, and no correct process proceeds to the next round. Eventually, every correct process receives either an **Echo** message from every process in some survivor set or a **Decide** message and decides.

□

We now show three theorems to conclude our proof of correctness for AsyncCrash. We present three theorems, each one showing that AsyncCrash satisfies one of the consensus properties.

**Theorem 3.4.12** *AsyncCrash satisfies Validity.*

**Proof:**

By Lemma 3.4.9, if a correct process decides in a round $r$, then it must decide upon the value proposed by the coordinator of some round $r' \leq r$. By the algorithm, the coordinator updates its estimate before sending **CoordEstimate** messages, and by Lemma 3.4.10, this estimate is the initial value of some process.

□

**Theorem 3.4.13** *AsyncCrash satisfies Agreement.*

**Proof:**

Let $p_i$ and $p_j$ be two correct processes that decide in rounds $r_i$ and $r_j$, respectively, of some execution of AsyncCrash, and $r$ be the first round such that some process $p_f$ decides in this execution. Let $v$ be the value that $p_f$ decides upon. By Lemma 3.4.8, the coordinator of round $r$ proposes $v$. By Lemma 3.4.7, if the coordinator of round $r' > r$ proposes a value $v'$, then $v = v'$. By Lemma 3.4.9, $p_i$ and $p_j$ decide upon the values proposed by the coordinators of rounds $r'_i$ and $r'_j$, respectively. By the algorithm and by assumption, we have that $r \leq r'_i$ and $r \leq r'_j$. We conclude that the values $v_i$ and $v_j$ that $p_i$ and $p_j$ decide upon respectively must be equal to $v$.

□

**Theorem 3.4.14** *AsyncCrash satisfies Termination.*

**Proof:**

This result follows directly from lemma 3.4.11.

□

### 3.4.2 Asynchronous consensus for Byzantine failures

Section 3.3.4 shows that Byzantine Partition (Byzantine Intersection) is a replication property that is necessary and sufficient to solve consensus in synchronous systems for Byzantine failures. Byzantine Partition is also necessary and sufficient to solve strong consensus in an asynchronous system extended with a failure detector $M \in \diamond\mathcal{M}$. To show this claim, we assume a system that does not satisfy Byzantine Partition, and we then construct an execution in which two correct processes decide differently. Such an execution violates the Agreement property of strong consensus. We then describe AsyncByz, an algorithm that solves strong consensus assuming the Byzantine Intersection property holds. This protocol has features from the protocols proposed by by Doudou and Schiper [DS98] and Kihlstrom *et al.* [KMMS97].

#### Lower bound on process replication

We claim that Byzantine Partition is necessary to solve consensus in asynchronous systems extended with failure detectors of the class $\diamond M$. Different from Section 3.3.4, we assume here that messages are digitally signed. More specifically, we assume a public key scheme that processes use to sign and verify the messages sent by other processes.

We show our claim by contradiction. Intuitively, we assume that there is an algorithm $\mathcal{A}$ that solves strong consensus for some system that does not satisfy Byzantine Partition. Thus, there is a partition of the processes into three subsets $B_1$, $B_2$, and $B_3$ such that none of these subsets contains a core, and we construct three executions as follows:

**Execution $E_\alpha$.** Every process $p_i$ has $v_1 \in V$ as its initial value, and every process

$p_j \in B_2$ crashes at time zero (0). All the other processes are correct in $E_\alpha$, and their failure detector modules suspect permanently every process in $B_2$, starting at time $t \geq 0$. In such an execution, processes in subsets $B_1$ and $B_2$ have to decide $v_1$, by the strong validity property. Suppose that $t_\alpha$ is the latest time at which some correct process decides in $E_\alpha$;

**Execution $E_\beta$.** Every process $p_i$ has $v_2 \in V$, $v_2 \neq v_1$, as its initial value, and every process $p_j \in B_1$ crashes at time zero (0). All the other processes are correct in $E_\beta$, and their failure detector modules suspect permanently every process in $B_1$, starting at time $t \geq 0$. In such an execution, processes in subsets $B_2$ and $B_3$ have to decide $v_2$, by the strong validity property. Suppose that $t_\beta$ is the latest time at which some correct process decides in $E_\beta$;

**Execution $E_\gamma$.** : Let $t_\gamma = \max(t_\alpha, t_\beta)$. Suppose that processes in subsets $B_1$ and $B_2$ are correct, whereas processes in $B_3$ are faulty. Furthermore, processes in $B_1$ have initial value $v_1$, and processes in $B_2$ have initial value $v_2$. In this execution, the failure detector modules of processes in $B_1$ suspect every process in $B_2$ from time zero to time $t_\gamma$, and the failure detector modules of processes in $B_2$ suspect every process in $B_1$ from time zero to time $t_\gamma$. Messages sent from processes in $B_3$ to processes in $B_1$ are as in $E_\alpha$, and the ones from processes in $B_3$ to processes in $B_2$ are as in $E_\beta$. Messages sent between processes in $B_1$ and $B_2$, however, are delayed until $t_\gamma$. It is not difficult to see that processes in $B_1$ cannot distinguish $E_\alpha$ from $E_\gamma$ up to time $t_\gamma$, and hence decide upon $v_1$. Processes in $B_2$, on the other hand, cannot distinguish $E_\beta$ from $E_\gamma$ up to time $t_\gamma$, and consequently decide $v_2$;

Agreement is clearly violated in Execution $E_\gamma$. Note that even if we assume digitally signed messages, the result still holds. Even assuming a full-information protocol, processes in $C$ can selectively choose the messages they want to include in their own messages to other processes.

We now provide a formal argument to support our claim.

**Claim 3.4.15** Let $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ be a system extended with a failure detector of the class

$\Diamond M$. If strong consensus is solvable in $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$, then $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ satisfies Byzantine Partition.

**Proof:**

Proof by contradiction. Suppose that there is a system $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ extended with a failure detector of the class $\Diamond M$ that does not satisfy Byzantine Partition and an algorithm $\mathcal{A}$ that solves strong consensus. If $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ does not satisfy Byzantine Partition, then there is a partition $(B_1, B_2, B_3) \in \mathcal{P}_3(\Pi)$ such that none of $B_1$, $B_2$, or $B_3$ contains a core. We construct an execution $E_\gamma$ of $\mathcal{A}$ in which Agreement is violated. First, we construct two other executions: $E_\alpha$ and $E_\beta$.

Let $E_\alpha = \langle \mathit{Init}_\alpha, \mathit{Steps}_\alpha, \mathit{Time}_\alpha, \mathit{Faulty}_\alpha, H_\alpha \rangle$ be as follows:

$$\begin{aligned}
\mathit{StepsOf}_\alpha(i) &= \emptyset, p_i \in B_2 \\
\mathit{Faulty}_\alpha(s) &= B_2, \forall s \in \mathit{Steps}_\alpha \\
H_\alpha(t,i) &= B_2, \forall t \geq 0, \forall p_i \in (B_1 \cup B_3) \\
\mathit{Init}_\alpha(i) &= v_1, v_1 \in V, \forall p_i \in (B_1 \cup B_3)
\end{aligned}$$

*Steps*$_\alpha$ and *Time*$_\alpha$ can be arbitrary, as long as the three strong consensus properties hold for $E_\alpha$.

Now let $E_\beta = \langle \mathit{Init}_\beta, \mathit{Steps}_\beta, \mathit{Time}_\beta, \mathit{Faulty}_\beta, H_\beta \rangle$ be as follows:

$$\begin{aligned}
\mathit{StepsOf}_\beta(i) &= \emptyset, p_i \in B_1 \\
\mathit{Faulty}_\beta(s) &= B_1, \forall s \in \mathit{Steps}_\alpha \\
H_\beta(t,i) &= B_1, \forall t \geq 0, \forall p_i \in (B_2 \cup B_3) \\
\mathit{Init}_\beta(i) &= v_2, v_2 \in V, \forall p_i \in (B_2 \cup B_3)
\end{aligned}$$

Similarly, *Steps*$_\beta$ and *Time*$_\beta$ can be arbitrary, as long as the three strong consensus properties hold for $E_\beta$.

Let $t_\alpha$ be a time value such that every correct process has decided at or before $t_\alpha$ in execution $E_\alpha$. Such a time exists by the termination property of strong consensus and the assumption that the algorithm is correct. Similarly, let $t_\beta$ be a time value such that every correct process decides at or before $t_\beta$ in $E_\beta$. Finally, let $t_\gamma$ be a time value that satisfies $t_\gamma > \max\{t_\alpha, t_\beta\}$. We are now ready to define execution $E_\gamma = \langle \mathit{Init}_\gamma, \mathit{Steps}_\gamma, \mathit{Time}_\gamma, \mathit{Faulty}_\gamma, H_\gamma \rangle$.

$$\begin{aligned}
\mathit{Faulty}_\gamma(s) &= \emptyset, \forall s \in \mathit{Steps}_\alpha \\
H_\gamma(t,i) &= \begin{cases} H_\beta(t,i) & \forall t \leq t_\gamma, p_i \in B_2 \\ H_\alpha(t,i) & \forall t \leq t_\gamma, p_i \in B_1 \\ \emptyset & \forall t > t_\gamma, \forall p_i \in \Pi \end{cases} \\
\mathit{Init}_\gamma(i) &= \begin{cases} v_1 & , \forall p_i \in B_1 \\ v_2 & , \forall p_i \in B_2 \end{cases}
\end{aligned}$$

$\mathit{Steps}_\gamma$ and $\mathit{Time}_\gamma$ satisfy the following:

- For every $t \leq t_\alpha$, if there is $s \in \mathit{Steps}_\alpha$ such that $\mathit{Time}_\alpha(s) = t$, then $s \in \mathit{Steps}_\gamma$ and $\mathit{Time}_\gamma(s) = t$;
- For every $t \leq t_\beta$, if there is $s \in \mathit{Steps}_\beta$ such that $\mathit{Time}_\beta(s) = t$, then $s \in \mathit{Steps}_\gamma$ and $\mathit{Time}_\gamma(s) = t$;
- If $s \in \mathit{Steps}_\gamma$ is such that $\mathit{Time}_\gamma(s) \leq t_\gamma$, then $s \in \mathit{Steps}_\alpha$ or $s \in \mathit{Steps}_\beta$;
- For every step $s \in \mathit{Steps}_\alpha$ such that a process $p_i \in B_1$ sends a message $m$ to a process $p_j \in B_2$, there is a step $s' \in \mathit{Steps}_\gamma$ such that $p_j$ receives $m$ in $s$ and $\mathit{Time}_\gamma(s) > t_\gamma$;
- For every step $s \in \mathit{Steps}_\beta$ such that a process $p_j \in B_2$ sends a message $m$ to a process $p_i \in B_1$, there is a step $s' \in \mathit{Steps}_\gamma$ such that $p_i$ receives $m$ in $s$ and $\mathit{Time}_\gamma(s) > t_\gamma$.

A process $p_i \in B_1$ cannot distinguish execution $E_\alpha$ from execution $E_\gamma$, whereas a process $p_j \in B_2$ cannot distinguish execution $E_\beta$ from execution $E_\gamma$. Thus, $p_i$ and $p_j$ decide upon $v_1$ and $v_2$, respectively, violating the agreement property. We conclude that $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ must satisfy Byzantine Partition to enable a solution to strong consensus.

□

**An algorithm for strong consensus**

This section describes AsyncByz an asynchronous consensus algorithm for Byzantine failures. AsyncByz is based on the rotating coordinator paradigm and proceeds in asynchronous rounds, as the **AsyncCrash** protocol we described in Section 3.4.1. Different from **AsyncCrash**, AsyncByz assumes a public key scheme: every process $p_i$ has a pair of keys $\langle \sigma_i, \delta_i \rangle$, such that $\sigma_i$ is the private key and $\delta_i$ is the public key. Only $p_i$ has access to its private key $\sigma_i$, whereas any process $p_j \neq p_i$ has free access to the public key $\delta_i$ of $p_i$. Every message $m$ process $p_i$ sends to $p_j$ has a certificate generated with $\sigma_i$. This certificate is unforgeable, that is, only a process under possession of $\sigma_i$ may generate a valid certificate for a given message. Upon reception of a message from $p_i$, process $p_j$ uses $\delta_i$ to assert that the message was constructed by $p_i$. Note that a faulty process $p_i$ may leak its private key to other processes, allowing them to forge messages of $p_i$.

We assume FIFO channels. This feature is necessary to prevent faulty processes from hampering the system. A process $p_i$ that sends out-of-order messages, by definition, is not mute. Thus, the failure detector $M$ is not guaranteed to eventually suspect $p_i$. If this kind of behavior is not detected by the algorithm, then there are executions of AsyncByz in which $p_i$ prevents some correct process $p_j$ from deciding. This observation will become clear after we describe the algorithm. A possible implementation of these FIFO channels uses counters to order the messages of a channel. More specifically, the communication subsystem only delivers message number $x$ to process $p_i$ once it delivers all messages prior to $x$. Also, the communication subsystem does not deliver messages with duplicate sequence numbers.

In an execution of AsyncByz, a correct process proceeds from one round to the next until it decides. The execution of a round for a process $p_i$ is divided into stages, as shown in Figure 3.6. A process $p_i$ begins the execution of round $r$ in stage *Start*, and the first action of $p_j$ is to send its estimate to the coordinator in a **Estimate** message. Initially, every process has as its estimate its own initial value.

Let $p_c$ be the coordinator of round $r$. Process $p_c$ transitions to stage *WaitEstimates* after sending an **Estimate** message to itself. Every other process $p_j$ transitions to stage *WaitCertEst*. Upon reception of one **Estimate** message from every process in some survivor set $S_{i_1}$, the coordinator then sends a **CertEstimate** message to all processes, and transitions to stage *WaitCertEst*. This message contains the current estimate of the coordinator, where this estimate is a value that the coordinator selects based on the received **Estimate** messages from processes.

The estimate sent in the **CertEstimate** message is chosen in the following manner: if there is a pair of survivor sets $S_{i_2}, S_{i_3} \in \mathcal{S}_\Pi$ such that every process in $S_{i_2} \cap S_{i_3}$ has the same estimate $v \in V$, then the coordinator updates its estimate to $v$, otherwise it keeps the previous estimate. After receiving a **CertEstimate** from the coordinator, a process $p_j$ replies with an **Echo** message. If $p_j = p_c$, then $p_j$ waits for **Echo** messages (transitions to *WaitEchoes*). Otherwise, $p_j$ transitions to *C_WaitRoundEst*. Once the coordinator receives one **Echo** message from every process in some survivor set, it sends a **RoundEstimate** message to all the other processes and transitions to *WaitRoundEst*. Upon reception of the first **RoundEstimate** message, a correct process forwards it to all the other processes, and transitions to *WaitRoundEst*.

In the *WaitRoundEst/C_WaitRoundEst* stage, a process either decides upon a value or prepares to proceed to the next round. If a correct process receives one **RoundEstimate** message from every process in some survivor set, then it decides and sends a **Decide** message to all the other processes. A process that receives a well-formed **Decide** message decides upon the value in this message. Note that a **Decide** message is not associated with any particular round or stage. That is, if $p_j$ receives a well-formed **Decide** message, independently of the round or stage it is, then $p_j$ decides. We explain the requirements necessary for a message to be well-formed later in this section.

Figure 3.5 shows an execution without failures or suspicions. The diagram in this figure shows the sequence of messages starting with **Estimate** messages and ending with **RoundEstimate** messages. It omits **Decide** messages for the sake of clarity.

During any stage of a round, a process may suspect the coordinator and broad-

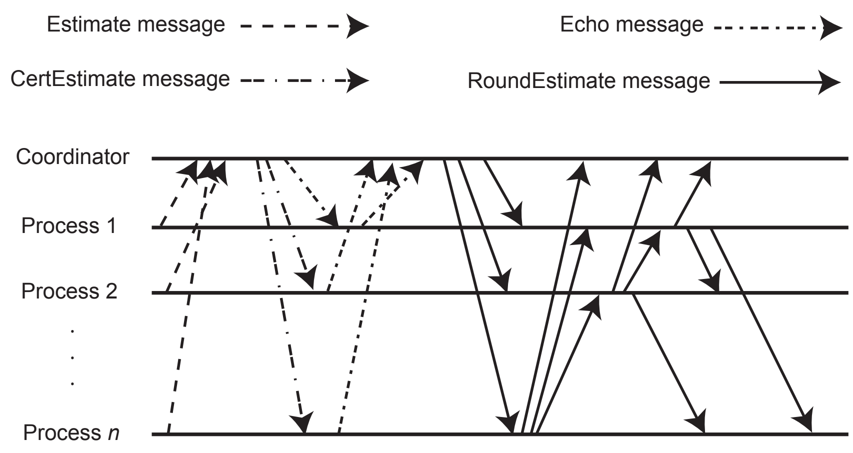

**Figure 3.5:** AsyncByz: Execution with no failures or suspicions

cast a **Suspicion** message. A process that eventually receives a **Suspicion** message from every process in some survivor set sends a **MoveOn** message. Upon reception of one **MoveOn** message from every process in some survivor set, a process $p_i$ updates its estimate with the most recent one in the set of received **MoveOn** messages and proceeds to the next round.

During this brief description of the algorithm, some details were omitted to preserve clarity. Now we discuss in more detail some of the aspects omitted.

Because we are assuming that processes may fail arbitrarily, the messages a faulty process sends are not guaranteed to have the correct form. Thus, it is necessary to have some mechanism to certify the content of a message besides authentication. Based on the definition of [DS98], we say a message is well formed if it follows some particular rules we establish for each type of message. A common rule for every type of message is that a message must be properly signed by its sender in order to be well formed. The rules particular to each type of message are as follows:

- An **Estimate** message is well formed if it contains an estimate value *Estimate* certified by a set *EstimateCert*, which is either empty or contains a collection of well-formed **Echo** messages, and a set *InitCert*, which is either empty or contains

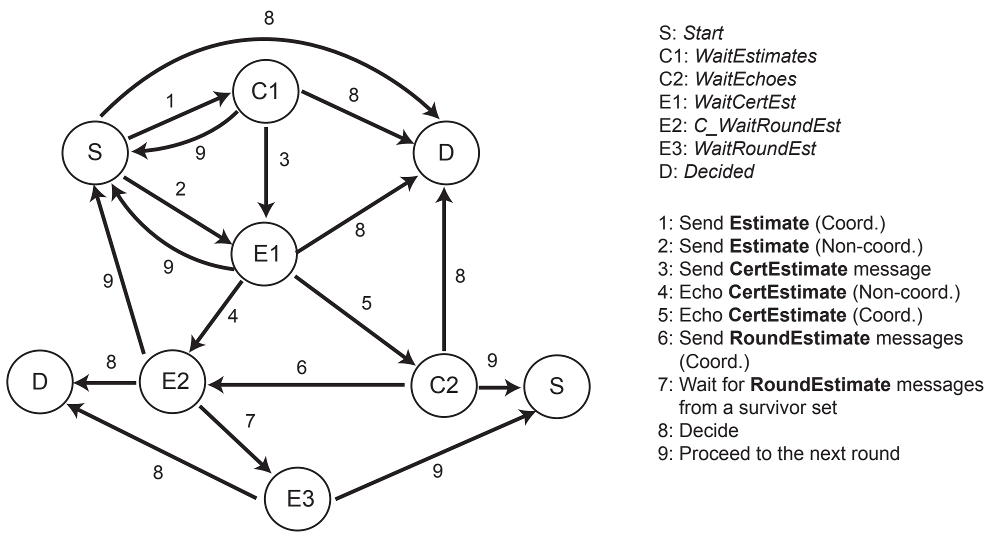
S: *Start*
C1: *WaitEstimates*
C2: *WaitEchoes*
E1: *WaitCertEst*
E2: *C_WaitRoundEst*
E3: *WaitRoundEst*
D: *Decided*
1: Send **Estimate** (Coord.)
2: Send **Estimate** (Non-coord.)
3: Send **CertEstimate** message
4: Echo **CertEstimate** (Non-coord.)
5: Echo **CertEstimate** (Coord.)
6: Send **RoundEstimate** messages (Coord.)
7: Wait for **RoundEstimate** messages from a survivor set
8: Decide
9: Proceed to the next round
S
C1
D
E1
E2
C2
E3

**Figure 3.6:** AsyncByz: Stage diagram for process $p_i$

a set of well-formed **MoveOn** messages. The **Echo** messages must be such that, for some survivor set $S$ and for every process $p_i \in S$, there is a message from $p_i$ in *EstimateCert*. Note that, in the algorithm, the set *EstimateCert*, if not empty, is the certificate of a well-formed **RoundEstimate** message. If it is empty, then the receiver assumes that this is the initial value of the sender. The set *InitCert* certifies the choice for the pair (*Estimate*, *EstimateCert*);

- A **RoundEstimate** message is well formed if it is properly signed by the coordinator, and it contains an estimate value *Estimate* certified by a set *EchoCert* of well-formed **Echo** messages, such that, for some survivor set $S$ and for every process $p_i \in S$, there is an **Echo** message from $p_i$ in *EchoCert*. As we show later in this section, in each round, the coordinator can only generate one well-formed **RoundEstimate** message;

- An **Echo** message is well formed if it is certified by a well-formed **CertEstimate** message;

- A **CertEstimate** message is well formed if it contains an estimate value that is certified by *InitEstCert*, which contains two sets: 1) the set of **Estimate** messages received by the coordinator; 2) the set *EstimateCert* that determined the estimate value of the coordinator in a previous round. The value *InitEstimate* proposed in the **CertEstimate** message by the coordinator depends on the **Estimate** messages received. If there are survivor sets $S_i$ and $S_j$, such that for every $p_i \in S_i$, there is a message in *InitEstCert* from $p_i$, and for every $p_j \in S_i \cap S_j$, there is a message from $p_j$ in *InitEstCert* proposing the same value $v \in V$, then $v$ is the value of *InitEstimate*. Otherwise, it is the value of *Estimate* in the end of the previous round;

- A **Suspicion** message is well formed if it is properly signed by its sender, and it requires no certification;

- A **MoveOn** message is well formed if it is properly signed by its sender, and it is certified by a set *SuspicionCert* of well formed **Suspicion** messages, such that, for some survivor set $S$ and for every process $p_i \in S$, there is a message from $p_i$ in *SuspicionCert*. Moreover, it contains an estimate value Estimate certified by a set *EstimateCert* of well-formed **Estimate** messages;

- A **Decide** message is well formed if it is properly signed by its sender, and it is certified by a set *RoundCert* of well-formed **RoundEstimate** messages, such that, for some survivor set $S$ and for every process $p_i \in S$, there is a well-formed message in *RoundCert* from $p_i$.

Late messages – messages sent in a previous round or in a previous stage of the same round – are simply discard, well-formed or not. In more detail, we assume that any message received by a process $p_i$ at stage $x$ of round $r$ that refers to a previous round $r' < r$, or to a stage $y$ of round $r$ that $p_i$ has been through already is discarded. On the other hand, a process $p_i$ keeps messages that refer to possible future stages or future rounds, and delivers them when appropriate.

A last comment is on the *MostRecentEstimate* function. This function traverses a set of **MoveOn** messages received by a process and determines the estimate value that was updated more recently. If there is no estimate updated in a more recent round, then the function returns *null*. As we show with the following lemmas, by using this update strategy, we guarantee that once some correct process decides, the decision value $v$ is locked, meaning that no correct process can decide on a different value $v'$. We present the pseudocode for algorithm AsyncByz in Figures 3.7 and 3.8. Table 3.3 presents a brief description of the variables used in the pseudocode. Additionally, Appendix E presents a specification of AsyncByz in TLA+.

**Table 3.3:** Variables used in the algorithm AsyncByz

| | |
|---|---|
| *Stage* | Indicates the stage the process is in the current round. |
| *r* | Current round of process $p_i$. |
| *c* | Process id of the coordinator of the current round. |
| *Estimate* | Current estimate of process $p_i$. |
| *EstimateCert* | Certificate for *Estimate*. |
| *InitCert* | Justifies the choice of the pair (*Estimate*, *EstimateCert*). |
| *InitEstimate* | Used by the coordinator to determine the value to propose in the current round. |
| *InitEstCert* | Certificate for InitEstimate. |
| *RoundEstimateCert* | Certificate for **RoundEstimate** messages. |
| *DecisionCert* | Certificate for **Decide** messages. |
| *SuspicionCert* | Certificate for **MoveOn** messages. |
| *MoveOnCert* | Certificate for the value of *Estimate*. |

We now provide a proof of correctness for AsyncByz. We assume a system profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ that satisfies Byzantine Intersection. The proof presented here follow a structure similar to the one for the proof of correctness of AsyncCrash.

**Lemma 3.4.16** *Let $E$ be an execution of AsyncByz. If a correct process $p_i$ does not decide in round $r > 0$, then $p_i$ eventually proceeds to round $r + 1$.*

**Proof:**

```
**Algorithm** AsyncByz **for process** $p_i$
**Input:** set $\Pi$ of processes; set $\mathcal{S}$ of survivor sets; initial value $v_i$
**Variables:**
  Stage $\leftarrow$ Start; $r \leftarrow 1$; $c \leftarrow 1$; Estimate $\leftarrow v_i$; InitEstimate $\leftarrow v_i$; InitEstCert $\leftarrow \emptyset$;
  EstimateCert $\leftarrow \emptyset$; SuspicionCert $\leftarrow \emptyset$; DecisionCert $\leftarrow \emptyset$;
  MoveOnCert $\leftarrow \emptyset$; RoundEstimateCert $\leftarrow \emptyset$; InitCert $\leftarrow \emptyset$;
**Stages:** Start; WaitEstimates; WaitCertEst; WaitEchoes; C_WaitRoundEst; WaitRoundEst

**Transitions:**
  **When** (Stage = Start)
    Send(**Estimate**, $i$, $r$, Estimate, EstimateCert, InitCert) to $c$
    **if** ($i = c$) **then** Stage $\leftarrow$ WaitEstimates
    **else** Stage $\leftarrow$ WaitCertEst

  **upon reception of** well formed (**Estimate**, $j$, $r$, $e_j$, $ec_j$, $ic_j$) and (Stage = WaitEstimates)
    InitEstCert $\leftarrow$ InitEstCert $\cup$ EstimateCert $\cup$ $\{$(**Estimate**, $p_j$, $r$, $e_j$, $ec_j$, $ic_j$)$\}$
    **if** ($\exists S \in \mathcal{S}_\Pi$ such that $\forall p_j \in S$, (**Estimate**, $j$, $r$, $e_j$, $ec_j$, $ic_j$) $\in$ InitEstCert)
    **then if** ($\exists S' \in \mathcal{S}_\Pi : \exists v \in V : \forall p_l \in (S \cap S')$: (**Estimate**, $l$, $r_l$, $v$, $ec_l$, $ic_l$) $\in$ InitEstCert)
        **then** InitEstimate $\leftarrow v$
        **else** InitEstimate $\leftarrow$ Estimate
    SendAll(**CertEstimate**, $i$, $r$, InitEstimate, InitEstCert)
    Stage $\leftarrow$ WaitCertEst

  **upon reception of** well formed (**CertEstimate**, $j$, $r$, $e_j$, $ec_j$) and
  (Stage = WaitCertEst)
    $m \leftarrow$ (**CertEstimate**, $j$, $r$, $e_j$, $ec_j$)
    Send(**Echo**, $i$, $m$) to $p_c$
    **if** ($i = c$) **then** Stage $\leftarrow$ WaitEchoes
             **else** Stage $\leftarrow$ C_WaitRoundEst

  **upon reception of** well formed (**Echo**, $j$, $m$) and
  (Stage $\in$ $\{$WaitEchoes$\}$)
    RoundEstimateCert $\leftarrow$ RoundEstimateCert $\cup$ $\{$ (**Echo**, $j$, $m$) $\}$
    **if** ($\exists S \in \mathcal{S}_\Pi$ such that $\forall p_j \in S$, (**Echo**, $j$, $m$) $\in$ RoundEstimateCert) **then**
      SendAll(**RoundEstimate**, $i$, $r$, Estimate, RoundEstimateCert)
      Stage $\leftarrow$ C_WaitRoundEst

  **upon reception of** well formed (**RoundEstimate**, $j$, $r$, $e_j$, $re_j$) and
  (Stage = C_WaitRoundEst)
    SendAll(**RoundEstimate**, $j$, $r$, $e_j$, $re_j$)
    **if** ($i \neq c$) **then** Estimate $\leftarrow e_j$; EstimateCert $\leftarrow re_j$
    DecisionCert $\leftarrow$ DecisionCert $\cup$ $\{$(**RoundEstimate**, $j$, $r$, $e_j$, $re_j$)$\}$
    Stage $\leftarrow$ WaitRoundEst

  **upon reception of** well formed (**RoundEstimate**, $j$, $r$, $e_j$, $re_j$) and
  (Stage = WaitRoundEst)
    DecisionCert $\leftarrow$ DecisionCert $\cup$ $\{$(**RoundEstimate**, $j$, $r$, $e_j$, $re_j$)$\}$
    **if** ($\exists S \in \mathcal{S}$ such that $\forall p_j \in S$, (**RoundEstimate**, $j$, $r$, $e_j$, $re_j$) $\in$ DecisionCert)
    **then** Decide upon $e_j$
        SendAll(**Decide**, $i$, $e_j$, DecisionCert)
        **Halt**
```

**Figure 3.7:** Algorithm AsyncByz

```
Transitions:
  upon suspicion of $p_c$
    SendAll(Suspicion, $i$, $r$)

  upon reception of well formed (Suspicion, $j$, $r$)
    SuspicionCert $\leftarrow$ SuspicionCert $\cup$ $\{$(Suspicion, $j$, $r$)$\}$
    if($\exists S \in \mathcal{S}_\Pi$ such that $\forall p_j \in S$, (Suspicion, $j$, $r$) $\in$ SuspicionCert)
    then SendAll(MoveOn, $i$, $r$, SuspicionCert, Estimate, EstimateCert)
      Stage $\leftarrow$ MoveOn

  upon reception of (MoveOn, $j$, $r$, $cs_j$, $e_j$, $ec_j$)
    MoveOnCert $\leftarrow$ MoveOnCert $\cup$ $\{$(MoveOn, $j$, $r$, $cs_j$, $e_j$, $ec_j$)$\}$
    if($\exists S \in \mathcal{S}_\Pi$ such that $\forall p \in S$, (MoveOn, $j$, $r$, $cs_j$, $e_j$, $ec_j$) $\in$ MoveOnCert)
    then (tmpEstimate, tmpEstimateCert) $\leftarrow$ MostRecentEstimate(MoveOnCert)
           if (tmpEstimate $\neq$ null)
           then Estimate $\leftarrow$ tmpEstimate; EstimateCert $\leftarrow$ tmpEstimateCert
                  InitCert $\leftarrow$ MoveOnCert
           $r \leftarrow r + 1$; $c \leftarrow ((c + 1) \bmod |\Pi|) + 1$
           SuspicionCert $\leftarrow \emptyset$; DecisionCert $\leftarrow \emptyset$; MoveOnCert $\leftarrow \emptyset$; RoundEstimateCert $\leftarrow \emptyset$; InitEstCert $\leftarrow \emptyset$
           Stage $\leftarrow$ Start

  upon reception of well formed (Decide, $j$, $e_j$, $ed_j$)
    Decide upon $e_j$
    SendAll(Decide, $i$, $e_j$, $ed_j$)
    Halt
```

**Figure 3.8:** Algorithm AsyncByz (*cont.*)

Proof by contradiction. Suppose that $p_i$ neither decides in $r$ nor proceeds to round $r + 1$. If $p_i$ does not decide in $r$, then it neither receives a well-formed **Decide** nor receives well-formed **RoundEstimate** messages from a survivor set. If $p_i$ does not receive a well-formed **Decide** message from some other process $p_j$, then no non-faulty process has decided in round $r$. This implies that no non-faulty process receives **RoundEstimate** messages from a survivor set and decides.

If no correct process receives **RoundEstimate** messages from a survivor set, then, by the algorithm, no correct process receives a well-formed **RoundEstimate** message in round $r$. This implies that the coordinator is mute to every correct process. By the mute completeness property of the failure detector, every correct process eventually suspects the coordinator. This implies that process $p_i$ eventually receives well-formed **MoveOn** messages from a survivor set and proceeds to round $r + 1$.

□

**Lemma 3.4.17** *Let $E$ be an execution of AsyncByz. In every round $r$ of $E$, the coordinator of $r$ can not construct two well-formed* **RoundEstimate** *messages $m$ and $m'$ such that the estimate of $m$ is $v$ and the estimate of $m'$ is $v'$, $v \neq v'$.*

**Proof:**

A **RoundEstimate** message is certified by a set $M$ of **Echo** messages, such that $M$ contains one **Echo** message from every process $p_i \in S$, for some survivor set $S \in \mathcal{S}_\Pi$. To generate two well-formed **RoundEstimate** messages, each one proposing values $v$ and $v'$, $v \neq v'$, the coordinator has to receive a well formed **Echo** message from every process $p_j \in (S \cup S')$, for $S, S' \in \mathcal{S}_\Pi$. Each **Echo** message from a process in $S$ certifies the reception of a well formed **CertEstimate** with estimate value $v$, whereas each **Echo** message from a process in $S'$ certifies the reception of a well formed **CertEstimate** with estimate value $v'$. By Byzantine Intersection, for every $S, S' \in \mathcal{S}_\Pi$, we have that there is a process $p_j \in (S' \cap S)$, such that $p_j$ is correct. From the algorithm, $p_j$ sends an **Echo** message upon reception of the first **CertEstimate** message. In the case that $p_j$ receives two well-formed **CertEstimate** messages from the coordinator, $p_j$ replies with an **Echo** message only upon the reception of the first message. Thus, it is not possible for the coordinator of $r$ to have two sets $M$ and $M'$ of **Echo** messages, $M \neq M'$, such that $M$ certifies value $v$, whereas $M'$ certifies value $v'$.

□

**Lemma 3.4.18** *Let $E$ be an execution of AsyncByz, $r$ be the first round of $E$ in which some non-faulty process $p_i$ decides, and $v \in V$ be the value $p_i$ decides upon. For every round $r' \geq r$, if a correct process $p_j$ proceeds to round $r' + 1$, then the estimate of $p_j$ is $v$.*

**Proof:**

We prove this lemma by induction on the round numbers $\rho$.

**Base case.** $\rho = r$. If some non-faulty process $p_i$ decides in round $r$, then it receives either one well-formed **RoundEstimate** message from every process in some survivor set $S_{re} \in \mathcal{S}_\Pi$ or a well-formed **Decide** message from some other process.

If $p_i$ receives a well-formed **Decide** message, then there is a sequence of processes $p_{j_1}, p_{j_2}, \ldots, p_{j_d}$ ($j_1 = i$, $d \geq 2$) such that $p_{j_k}$ decides by receiving a **Decide** message from $p_{j_{k+1}}$, $p_{j_d}$ sends **Decide** messages by receiving **RoundEstimate** from processes in some survivor set $S$, and $j_k \neq j_{k'}$, $k \neq k'$. Since the certificate of the **Decide** messages induced by this sequence is not altered by assumption (otherwise one of these messages is not well formed), $p_i$ has also to decide upon the same value as $p_{j_d}$. Thus, suppose without loss of generality that $p_i$ decides due to the reception of one well-formed **RoundEstimate** from every process in some $S_{re} \in \mathcal{S}_\Pi$.

If a correct process $p_j$ proceeds to round $r + 1$, then it must receive a well-formed **MoveOn** message from every process in some survivor set $S_{mo} \in \mathcal{S}_\Pi$ in round $r$. By the Byzantine Intersection property, there is at least one correct process $p_{i_2}$, such that $p_{i_2} \in (S_{mo} \cap S_{re})$. The estimate of $p_{i_2}$ must be $v$ because $p_{i_2}$ forwards a **RoundEstimate** message with value $v$ to $p_i$ and updates its estimate before doing so. The estimate $p_{i_2}$ sends in the **MoveOn** message is therefore $v$, the same value $p_i$ decides upon, updated in round $r$. Process $p_j$, by receiving a well-formed **MoveOn** message from every process in $S_{mo}$, updates its estimate to the value proposed in the latest round. From our previous observation, at least process $p_{i_2}$ sends a well formed **MoveOn** message with estimate $v$ and update round $r$. By Lemma 3.4.17, no other process $p_{i_3}$ sends a well-formed **MoveOn** message with value $v'$, $v \neq v'$. Process $p_j$ consequently proceeds to round $r + 1$ with estimate $v$.

**Induction step.** Now assume that the proposition is valid for every round $\rho = r' \geq r$. We prove for $r' + 1$. By the induction hypothesis, every correct process that proceeds to round $r' + 1$ does so with estimate $v$. By assumption, there is at least one survivor set $S_c \in \mathcal{S}_\Pi$ such that $S_c$ contains only correct processes. To certify the value in a **CertEstimate** message, the coordinator of round $r' + 1$ collects an **Estimate** message from every process $p_{i_1} \in S_{es}$, for some survivor set $S_{es} \in \mathcal{S}_\Pi$. By the Byzantine Intersection property, the intersection $S_c \cap S_{es}$ is not empty and contains only correct processes. From the algorithm, a well-formed **CertEstimate** sent by the coordinator $p_c$ or round $r' + 1$ has consequently to propose $v$. By Lemma 3.4.17, if the coordinator eventually sends a well formed **RoundEstimate**, then it must propose $v$.

Suppose $p_j$ receives one well-formed **MoveOn** message from every process in some survivor set $S_{mo} \in \mathcal{S}_\Pi$. If some correct process $p_{i_2}$ changes its estimate before sending **MoveOn** messages in a round, then $p_{i_2}$ does so upon reception of a well-formed **RoundEstimate**. This message, as observed before, has to propose $v$. If such a process $p_{i_2}$ is in $S_{mo}$, then $p_j$ updates its estimate to $v$ with update round equal to $r' + 1$. Otherwise, there is some process $p_{i_3} \in S_{mo}$, such that the estimate of $p_{i_3}$ is properly certified and is the most recent.

The estimate of $p_{i_3}$ must be $v$. Suppose the contrary. That is, the estimate of $p_{i_3}$ is $v' \neq v$. In this case, there is a set $M$ of well-formed **MoveOn** messages sent in round $r'$, one from each process in some survivor set $S'_{mo} \in \mathcal{S}_\Pi$ such that one of the messages in $M$ certifies the estimate $v'$ and this estimate was updated in round $r''$. By the argument for the base case, we have that $r'' > r$. By the algorithm, the certificate for this estimate is a collection of **Echo** messages, which contain a copy of the **Estimate** messages the coordinator of $r''$ used to select the round estimate. By the induction hypothesis, the estimate of every correct process in the beginning of round $r''$ is $v$. Thus, the estimate of the coordinator of $r''$ must be $v$, by the algorithm.

We conclude that $p_j$ also updates its estimate to $v$ at the end of round $r' + 1$ and therefore proceeds to round $r' + 2$ with estimate $v$.

□

**Lemma 3.4.19** *Let $E$ be an execution of AsyncByz, $r$ be the first round of $E$ in which some non-faulty process $p_i$ decides, and $v \in V$ be the value $p_i$ decided upon. For every round $r' > r$, if the coordinator of $r'$ sends a well-formed* **RoundEstimate** *during the execution of round $r'$, then the value in this message is $v$.*

**Proof:**

By the algorithm, the coordinator of a round can only send a well-formed **RoundEsti-**

**mate** message after receiving well-formed **Echo** messages from the processes in some survivor set. To receive well-formed **Echo** messages, the coordinator must have sent first well-formed **CertEstimate** messages. The coordinator can only send well-formed **CertEstimate** messages after receiving **Estimate** messages from a survivor set, which certify its choice for the estimate value proposed in these messages.

From Lemma 3.4.18, the estimate of every correct process at stage *Start* of round $r' > r$ is $v$. By assumption, there is at least one survivor set $S_c \in \mathcal{S}_\Pi$ such that $S_c$ contains only correct processes. Let $S_{es} \in \mathcal{S}_\Pi$ be a survivor set such that the coordinator $p_c$ of round $r'$ receives one well-formed **Estimate** message from every process in $S_{es}$. By Byzantine Intersection, $S_{es} \cap S_c$ is not empty and contains only correct processes. Moreover, for every $S \in \mathcal{S}_\Pi$, $S_{es} \cap S$ contains at least one correct process. Thus, there are no two survivor sets $S, S' \in \mathcal{S}_\Pi$ such that:

1. The estimate of every process in $S_{es} \cap S$ is $v$, $v \in V$
2. The estimate of every process in $S_{es} \cap S'$ is $v'$, $v' \in V$
3. $v \neq v'$

A well formed **CertEstimate** sent by the coordinator of round $r'$ hence must propose $v$.

By sending **CertEstimate** messages, the coordinator eventually receives one well-formed **Echo** message from every process in some survivor set $S_{es} \in \mathcal{S}_\Pi$. Each one of these **Echo** messages is well formed if it contains a copy of the **CertEstimate** message that the coordinator sent. Consequently, a well-formed **Echo** message acknowledges the value $v$ as the proposed value for round $r'$. A well-formed **RoundEstimate** message sent by the coordinator, consequently, must propose $v$.

□

**Lemma 3.4.20** *Let $E$ be an execution of AsyncByz, $r$ be the first round of $E$ in which some non-faulty process decides, and $p_i$ be a non-faulty process that decides in round $r' \geq r$. Process $p_i$ decides upon the value proposed by the coordinator of some round $r'$', $r' \geq r'' \geq r$.*

**Proof:**

A process decides by either receiving a well-formed **Decide** message or receiving well-formed **RoundEstimate** messages from a survivor set. If $p_i$ decides by receiving a **Decide** message, then there is a sequence of processes $p_{j_1}, p_{j_2}, \ldots, p_{j_d}$ ($j_1 = i$, $d \geq 2$) such that $p_{j_k}$ decides by receiving a **Decide** message from $p_{j_{k+1}}$, $p_{j_d}$ sends **Decide** messages by receiving **RoundEstimate** from processes in some survivor set $S$, and $j_k \neq j_{k'}$, $k \neq k'$. Since the certificate of the **Decide** messages induced by this sequence is not altered by assumption (otherwise one of these messages is not well-formed), $p_i$ also decides upon the same value as $p_{j_d}$.

Suppose that $p_{j_d}$ decides in round $r_{j_d} \leq r'$. By assumption, $r_{j_d}$ must be greater or equal to $r$. By the algorithm, $p_{j_d}$ must receive well-formed **RoundEstimate** messages from some survivor set in $\mathcal{S}_\Pi$. Such messages can only be well formed if they were signed by the coordinator. Moreover, by Lemma 3.4.17, the coordinator can only propose one single value in a round.

By the same argument, if $p_i$ decides by receiving well-formed **RoundEstimate** messages from a survivor set, then it must decide upon the value of the coordinator of round $r'$.

□

**Lemma 3.4.21** *Let $E$ be an execution of AsyncByz in which the initial value $v_j$ of every correct process $p_j$ is $v \in V$. If a correct process $p_i$ executing round $r$, $r \geq 1$, eventually proceeds to round $r+1$, then it does so with estimate value $v' = v$.*

**Proof:**

We prove this lemma by induction on the round numbers $r$.

**Base case:** $r = 1$. Suppose the coordinator of round 1, $p_1$, constructs a well-formed **CertEstimate** message. To construct such a message, the coordinator receives one **Estimate** message from every process $p_{i_1}$ in some survivor set $S_{es} \in \mathcal{S}_\Pi$. From the Byzantine Intersection property, $S_{es} \cap S_c$ is not empty, where $S_c$ is a survivor set containing only correct processes (such a survivor set exists by assumption). Thus, for every process $p_{i_1} \in S_{es} \cap S_c$, the **Estimate** message $p_{i_1}$ sends to the coordinator $p_1$ proposes $v$. Moreover, there is no survivor set $S \in \mathcal{S}_\Pi$, such that:

$$\forall p_{i_1} \in S_{es} \cap S : p_{i_1} \text{ sends an } \textbf{Estimate} \text{ message to } p_1 \text{ containing a value } v' \neq v$$

because $S_{es} \cap S \cap S_c$ is not empty. If $p_1$ eventually constructs a well-formed **RoundEstimate** and sends to other processes in $\Pi$, then the value proposed in these messages must be $v$.

To proceed to round $r + 1 = 2$, a correct process $p_i$ has to receive a well-formed **MoveOn** message from every process in some survivor set $S_{mo} \in \mathcal{S}_\Pi$. Suppose that there is at least one process $p_{i_2} \in S_{mo}$ such that $p_{i_2}$ receives at least one well-formed **RoundEstimate** message during round $1$. The value proposed in the **RoundEstimate** message must be $v$, and consequently $p_{i_2}$ updates its estimate with value $v$. Otherwise, every **MoveOn** message contains the initially uncertified value of the process that has sent the message. In the former case, $p_i$ updates its estimates with $v$, updated in round $1$. In the latter case, $p_i$ does not update its estimate, and keeps its initial value, which is $v$. Every correct process $p_i$ therefore proceeds to round $1$ with estimate value $v$.

**Induction step.** Suppose that the proposition is valid for every round $r \geq 1$. We prove for round $r + 1$. To proceed to round $r + 2$, a correct process $p_i$ has to receive a well-formed **MoveOn** message from every process in some survivor set $S_{mo} \in \mathcal{S}_\Pi$. Suppose that there is at least one process $p_{i_1} \in S_{mo}$ such that $p_{i_1}$ receives at least one well-formed **RoundEstimate** message during round $r$. By the inductive hypothesis and the Byzantine Intersection property, the value proposed in a well-formed **RoundEstimate** message from the coordinator of round $r$ must be $v$, and consequently $p_{i_1}$ updates its estimate with value $v$.

If there is no such a process $p_{i_1}$, then for every process $p_{i_2} \in S_{mo}$, the estimate $p_{i_2}$ sends in a well-formed **MoveOn** message to $p_i$ in round $r + 1$ is still the estimate $p_{i_2}$ has when it proceeds to round $r + 1$, which is $v$. Moreover, there is a process $p_{mr}$ such that $p_{mr}$ has the most recent certified estimate. Suppose that $p_{mr}$ updates this estimate in round $r'$, $1 < r' < r + 1$.

If the estimate $p_{mr}$ is $v'$, then there is a set of well-formed **MoveOn** messages from round $r'$, containing one message from each process in some survivor set $S'_{mo} \in \mathcal{S}_\Pi$, and certifying the value $v'$ as the estimate of $p_{mr}$. To form such a set, a process needs, by the algorithm, a set $M$ of well-formed **Echo** messages certifying the choice of the coordinator of round $r'$ for the estimate $v'$. By the algorithm, an **Echo** message from some process $p_j$ contains a copy of the **CertEstimate** message $p_j$ receives from the coordinator of round $r'$. The certificate of a **CertEstimate** message is a set of **Estimate** messages. Again by the algorithm, the **CertEstimate** message proposes the value $v'$ that is the estimate of every process in the intersection of two survivor sets $S, S' \in \mathcal{S}_\Pi$. By the induction hypothesis, every correct process proceeds to round $r'$ (or starts in $r'$) with estimate $v$. By Byzantine Intersection, $S \cap S' \cap S_c$ is not empty, where $S_c$ is a survivor set containing only correct processes. We conclude that the coordinator of round $r'$ can not build a valid certificate for **CertEstimate** messages certifying $v'$. Consequently, the estimate of $p_{mr}$ must be $v$ and $p_i$ proceeds to round $r + 2$ with estimate $v$.

□

**Lemma 3.4.22** *Let $E$ be an execution of AsyncByz, and $r$ be a round of $E$ such that the coordinator of $r$ sends at least one well-formed* **RoundEstimate** *message* msg. *If every correct process $p_i$ that executes round $r$ has $v$ as its estimate when $p_i$ starts round $r$, then the estimate of* msg *must be $v$.*

**Proof:**

To construct a well-formed **RoundEstimate** message, the coordinator of round $r$ must receive a set $M$ of well-formed **Echo** messages such that there is one message in $M$ from every process in some survivor set $S_e \in \mathcal{S}_\Pi$. Each **Echo** message in $M$ from process $p_i$ contains a copy of the **CertEstimate** message that the coordinator of round $r$ generates. By the algorithm, such a **CertEstimate** message must be well-formed to be received by $p_i$. Again by the algorithm, if there is some value $v'$ such that $v'$ is the estimate of every process in the intersection of survivor sets $S$ and $S'$, then $v'$ is the estimate in **CertEstimate**. By assumption, there is some survivor set $S_c$ containing only correct processes, and the estimate of every process in $S \cap S_c$ is $v$. Moreover, by Byzantine Intersection, there is no other survivor set $S''$ such that the estimate of every process in $S \cap S''$ is $v'' \neq v$ because $S \cap S'' \cap S_c$ is not empty. Thus, the round estimate value in *msg* must be $v$.

□

**Theorem 3.4.23** *AsyncByz satisfies Termination.*

**Proof:**

From the weak accuracy property of the failure detector, for every execution $E$, there is a time $t$ and some correct process $p_i$ such that for every $t' \geq t$ and correct $p_j$, $p_i \notin H_E(t', j)$. By lemma 3.4.16, a correct process that does not decide in round $r$ eventually proceeds to round $r+1$. Therefore, there is a round $r'$ such that every correct process $p_j$ that does not decide in a round $r'' < r'$ starts executing round $r'$ at time $t' \geq t$ and $p_i$ is the coordinator of $r'$. In this round, no correct process suspects the coordinator, which implies that no correct process receives a sufficient number of **MoveOn** messages to proceed to the next round (every survivor set contains at least one correct process by the Byzantine Intersection property and the assumption that at least one survivor set contains only correct processes). Eventually, every correct process receives either one **RoundEstimate** message from every process $p_{i_1} \in S_{re}$, for some $S_{re} \in \mathcal{S}_\Pi$, or one **Decide** message and decides.

□

**Theorem 3.4.24** *AsyncByz satisfies Strong Validity.*

**Proof:**

Let $E$ be an execution of AsyncByz such that the initial value of every correct process is $v$, and some correct process $p_i$ decides in round $r$. If $r = 1$, then by the algorithm and Lemma 3.4.22, $p_i$ decides upon $v$. If $r > 1$, then there are two cases to analyze: 1) $r$

is the first round in which some non-faulty process decides; 2) some non-faulty process decides in round $r' < r$.

Suppose that $r$ is the first round in which some process decides. By Lemma 3.4.21 the estimate of every correct process is $v$ in the beginning of round $r$. By Lemma 3.4.22 the round estimate must be $v$. Finally, by Lemma 3.4.20, $p_i$ must decide upon the round estimate proposed by the coordinator of round $r$, which is $v$.

If $r$ is not the first round in which some non-faulty process decides, then there is some round $r' < r$ such that some non-faulty process $p_j$ decides. By the argument given for the first case, $p_j$ must decide upon $v$. By Lemma 3.4.19 and Lemma 3.4.17, if the coordinator of round $r'' \geq r'$ proposes a round estimate, then this estimate must be $v$. By Lemma 3.4.20, $p_i$ must decide upon the value proposed by the coordinator of some round $r'' \geq r$, which must be $v$.

□

**Theorem 3.4.25** *AsyncByz satisfies Agreement.*

**Proof:**
Let $E$ be an execution of AsyncByz. Let $r$ be the first round of $E$ in which some correct process decides. From Lemma 3.4.20, if two correct processes decide in round $r$, then they have to decide upon the same value $v$. If a correct process decides in round $r' > r$, then by Lemmas 3.4.19 and 3.4.20 $p_i$ must decide upon $v$.

□

## 3.5 Conclusions

This chapter presented several results related to the traditional consensus problem for both synchronous and asynchronous systems. These results show some benefits of having a more expressive failure model as opposed to one single threshold to model the number of failures. The most important conclusions are:

- **The number of rounds for synchronous algorithms can be different for different types of failure.** We have presented synchronous algorithms for both crash and arbitrary failures, and showed an example of a system in which solving consensus for crash failures takes fewer rounds compared to solving consensus for Byzantine failures. This shows that a traditional theoretical result for the threshold model does not hold when we consider dependent failures;

- **Solutions to consensus may require fewer replicas.** The partition and intersection properties determine the minimum amount of replication to solve consensus in our model of cores and survivor sets. For Byzantine failures, the example of Chapter 2 show that we can solve consensus with the algorithms we presented in this chapter, but no solution exists if we assume a threshold on the number of failures. This exposes an artificial constraint the threshold model imposes on systems to enable solutions.

To design algorithms with cores and survivor sets, we in general used two equivalent properties that are necessary and sufficient to solve consensus. For three out of the four cases analyzed, we used a partition property to show that no system that does not satisfy this property can solve consensus, and an equivalent intersection property to design algorithms. The only case we did not use the same techniques (synchronous systems, crash failures) was a simpler case, which has no strict constraint on the amount of replication to enable a solution to consensus.

The next chapter develops on the idea of extending the set of partition and intersection properties we presented so far, with the goal of applying the same techniques to other problems.

Chapter 4

# New replication predicates for dependent-failure algorithms

The previous chapter discusses properties that are necessary and sufficient to solve the consensus problem in different system and failure models, using cores and survivor sets to describe sets of faulty processes instead of a threshold. This chapter presents general versions of the properties of the previous chapter. These properties generalize predicates of the form $n > k \cdot t$, and they are interesting because they are applicable to a broad set of problems, thus constituting, along with the core/survivor set model, a theoretical framework for designing fault-tolerant distributed algorithms. We begin by observing how lower bounds on process replication have been traditionally derived.

Lower bounds for the amount of process replication typically rely upon an argument with the following structure:

1. Partition the $n$ processes into $k$ blocks, where each block has at most $\lceil t/b \rceil$ processes, $t \geq \lceil nb/k \rceil$, and $k, b$ are positive integers such that $k > b \geq 1$.

2. Construct a set of executions. For each block $A$, there is at least one of the executions in which all the processes in $A$ are faulty.

3. Given the set of executions, show that some property of interest is violated. Con-

clude that if the maximum number of faulty processes in an execution is never larger than $t$, then $t < \lceil nb/k \rceil$ and $n > \lfloor kt/b \rfloor$.[1]

Examples of such proofs include consensus with arbitrary process failures and no digital signatures requiring $n > 3 \cdot t$ ($k = 3$, $b = 1$) [LSP82], primary backup with general omission failures requiring $n > 2 \cdot t$ ($k = 2$, $b = 1$) [Mul95b], and consensus with the eventually strong failure detector $\Diamond S$ requiring $n > 2 \cdot t$ ($k = 2$, $b = 1$) [CHT96, CT96].

We call a predicate like $n > \lfloor kt/b \rfloor$ a *replication predicate*: it gives a lower bound on the number of processes required given all possible sets of faulty processes. Expressing bounds in terms of $t$ is often referred to as a *threshold model*. Using $t$ to express the number of faulty processes is convenient, but the bounds can lead to mistaken conclusions when processes do not fail independently or do not have identical probabilities of failure. Assuming that *any* subset of $t$ processes can be faulty implies that failures are independent and identically distributed (IID). To use an algorithm developed under the threshold model on a system that does not have IID failures, one can compute the maximum number of processes that can fail in any execution, and then use that number as $t$. On the other hand one may be able to use fewer processes if an algorithm based on non-IID failures is used instead.

In the previous chapter, we studied consensus under the core/survivor set model. We derived replication requirements in our new model and presented algorithms that showed these bounds to be tight. This chapter generalizes these results to algorithms other than consensus. We show how the lower bound argument given above can be easily generalized to accommodate our model of dependent failures. This argument leads to a replication predicate that we call $k$–Partition, which generalizes the replication predicate $n > k \cdot t$ ($b = 1$) for when failures are not IID. The $k$–Partition property, however, may not prove to be very useful when designing an algorithm. An equivalent property, which we call $k$–Intersection, is often more useful for this purpose. It is more useful for designing algorithms because algorithms often refer to minimal sets of correct

[1]Some authors have used different symbols, such as $f$, to indicate an upper bound on the number of faulty processes.

processes ($n - t$ processes when process failures are IID). These properties generalize the two properties we developed for consensus in Chapter 3.

In this chapter, we define the replication predicate $k$–Partition for $k > 1$. We then define $k$–Intersection and show that it is equivalent to $k$–Partition. We illustrate the utility of $k$–Intersection by showing that the *M-Consistency* property [MR97a] for Byzantine Quorum Systems is equivalent to $4$–Intersection. Thus, a system that requires M-Consistency has a replication predicate of $4$–Partition. Finally, we examine one point in the space of replication predicates for $b > 1$. We do so by considering a weak version of the *Leader Election* problem for synchronous systems that can suffer receive-omission failures. We review a previously-given lower bound proof that argues $n > \lfloor 3t/2 \rfloor$ ($k = 3$, $b = 2$) for IID failures. This proof yields a definition that we call (3,2)-Partition. We derive an equivalent (3,2)-Intersection property and use it to develop an optimal protocol for weak leader election. An immediate consequence is that the lower bound $n > \lfloor 3t/2 \rfloor$ for IID failures is tight. This result is the first to show this bound to be tight.

## 4.1 $k$ properties

In the generic lower bound proof described in the introductory part of this chapter, one first partitions the set of processes into $k$ blocks, and then constructs a set of executions. For each block $A$, there is some execution in which all the processes in $A$ are faulty. Being able to fail all the processes of a particular block then enables the construction of an execution in which some property is violated. For example, for consensus, the property violated is *agreement*. For primary-backup protocols, the property violated is the one that says that at any time there is at most one primary.

Having derived a contradiction, the proof concludes by stating that one cannot partition the processes in the manner that was done. With the threshold model and $b = 1$, this conclusion implies that not all processes of any subset of size $\lceil n/k \rceil$ can be faulty, and consequently $t < \lceil n/k \rceil$. In our dependent failure model, this same conclusion implies that in any partition of the processes into $k$ blocks, there is at least one block $A$ that does not contain only faulty processes: $A$ contains a core. More formally, let $\mathcal{P}_k(\Pi)$ be the set of partitions of $\Pi$ into $k$ blocks. We then have the following property for a system profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$:

**Property 4.1.1** **$k$–Partition**, $k > 1$, $|\Pi| > k$: $\forall \mathcal{A} \in \mathcal{P}_k(\Pi) : \exists A_i \in \mathcal{A} : \exists C \in \mathcal{C}_\Pi : C \subseteq A_i$ □

Although $k$–Partition is useful for lower bound proofs, it is often not very useful for the design of algorithms because algorithms often refer to survivor sets and not cores. For example, the algorithm for consensus by Chandra and Toueg for crash failures in asynchronous systems with failure detectors of the class $\diamond S$ assumes at least $2t+1$ processes. For this number of processes, any pair of subsets of size $n-t$ has a non-empty intersection, and this property is crucial to avoid the violation of the agreement property of consensus. A more general way of stating this same constraint is to say any two survivor sets intersect, or equivalently that $\mathcal{S}_\Pi$ is a coterie [GMB85].

### 4.1.1 $k$–Intersection

We now state the property that we show to be equivalent to $k$–Partition and that references survivor sets instead of cores. We call it $k$–Intersection. $k$–Intersection states that for a system profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$, for every set $T \subset \mathcal{S}_\Pi$ of size $k$, there is some process that is in every element of $T$. Let $\mathcal{G}_x(A)$ be the set of all the subsets of $A$ of size $x$; if $|A| < x$, then $\mathcal{G}_x(A) = \emptyset$. We have the following property for a system profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$:

**Property 4.1.2** **$k$–Intersection**, $k > 1$, $|\Pi| > k$, $|\mathcal{S}_\Pi| > k$: $\forall T \in \mathcal{G}_k(\mathcal{S}_\Pi) : (\cap_{S \in T} S) \neq \emptyset$ □

As an example, the set $\mathcal{S}_\Pi$ in Example 2.3.1 satisfies 3–Intersection. We now show the equivalence between $k$–Partition and $k$–Intersection.

**Theorem 4.1.3** *$k$–Partition $\equiv$ $k$–Intersection*

**Proof:**

$\Rightarrow$: Proof by contrapositive. Suppose a system profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ such that there is a subset $\mathcal{S} = \{S_1, \ldots, S_k\} \subset \mathcal{S}_\Pi$ such that $\bigcap \mathcal{S} = \emptyset$. We then build a partition $\mathcal{A} = \{A_1, \ldots, A_k\}$ as follows:

$$\begin{aligned}
A_1 &= \Pi \setminus S_1 \\
A_2 &= \Pi \setminus (S_2 \cup A_1) \\
&\vdots \\
A_i &= \Pi \setminus (S_i \cup A_1 \cup A_2 \ldots \cup A_{i-1}) \\
&\vdots \\
A_k &= \Pi \setminus (S_k \cup A_1 \ldots \cup A_{k-1})
\end{aligned}$$

Suppose without loss of generality that no $A_i$ is empty (if some block is empty, then the partition is into $k'$ blocks, $k > k'$, although it is possible to re-arrange processes to obtain a partition into $k$ blocks). It is clear from the construction that no two distinct blocks $A_i, A_j$ intersect. It remains to show that: 1) $\bigcup \mathcal{A} = \Pi$; 2) $\forall i \in \{1, \ldots, k\} : A_i$ does not contain a core. To show 1), consider the following derivation:

$$\begin{aligned}
\bigcup \mathcal{A} &= (\Pi \setminus S_1) \cup (\Pi \setminus (S_2 \cup A_1)) \cup \ldots \\
&\quad \cup (\Pi \setminus (S_k \cup A_1 \cup A_2 \ldots \cup A_{k-1})) && (4.1)\\
&= \Pi \setminus ((S_1 \cap (S_2 \cup A_1)) \cap \ldots \cap (S_k \cup A_1 \cup A_2 \ldots \cup A_{k-1})) && (4.2)\\
&= \Pi \setminus (S_1 \cap S_2 \cap \ldots \cap (S_k \cup A_1 \cup A_2 \ldots \cup A_{k-1})) && (4.3)\\
&\vdots \\
&= \Pi \setminus (\cap_i S_i) && (4.4)\\
&= \Pi && (4.5)
\end{aligned}$$

Explaining the derivation:

- Line 4.1 to Line 4.2 follows from the observation that, for any subsets $A, B$ of $\Pi$, we have that $(\Pi \setminus A) \cup (\Pi \setminus B) = \Pi \setminus (A \cap B)$;

- Line 4.2 to Line 4.3: the intersection between $S_1$ and $A_1$ has to be empty, since $S_1$ contains exactly the elements we removed from $\Pi$ to form $A_1$;
- Line 4.3 to Line 4.4: by repeating inductively the process used to derive Line 4.3, we remove every term $A_i$ present in the equation;
- Line 4.4 to Line 4.5: By assumption, the intersection of $S_1$ through $S_k$ is empty.

To show 2), we just need to observe that any $A_i$ is such that we removed all the elements of $S_i$. By the definitions of a core and of a survivor set, a subset that does not contain elements from some survivor set does not contain a core.

$\Leftarrow$: Proof also by contrapositive. Let $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ be a system profile such that there is a partition $\{A_1, \ldots, A_k\}$ of $\Pi$ in which no $A_i$ contains a core. Because no block contains elements from every survivor set (no block contains a core), we have that, for every $A_i$, there is a survivor set $S_i$ such that $S_i \cap A_i = \emptyset$. Consequently, we have that $\cap_i S_i$ is empty, otherwise either some $A_i$ contains an element that is in $\bigcap_i S_i$ or $\{A_1, \ldots, A_k\}$ is not a partition, either way contradicting our previous assumptions.
$\square$

In the remainder of this section, we discuss the utility of these properties. In particular, we show the equivalence between 4–Intersection and M-Consistency [MR97a].

### 4.1.2 4–Intersection and M-Consistency

In [MR97a], the following *M-Consistency* property was defined. It was stated that this property was necessary to implement a *masking Byzantine quorum system*. This property allows a process to identify a result from a non-faulty server. The set $\mathcal{Q}$ used in this definition is the set of quorums, and $\mathcal{B}$ is the fail-prone system.

**Property 4.1.4 M-Consistency**: $\forall Q_1, Q_2 \in \mathcal{Q} : \forall B_1, B_2 \in \mathcal{B} : (Q_1 \cap Q_2) \setminus B_1 \not\subseteq B_2$
$\square$

The paper then shows that if all sets in $\mathcal{B}$ have the same size $t$, then M-Consistency implies $n > 4t$.

We show that M-Consistency is equivalent to 4–Intersection. Since a faulty process can stop sending messages, we can use $\mathcal{S}_\Pi$ as the set of quorums: waiting to receive messages from more than a survivor set could prove fruitless. A fail-prone set is the complement of a survivor set, and for any two sets $X$ and $Y$, $(X \setminus Y) \equiv (X \cap \bar{Y})$, where $\bar{Y}$ is the complement of $Y$. Hence, we can rewrite M-Consistency as:

$$\forall Q_1, Q_2 \in \mathcal{S}_\Pi : \forall B_1, B_2 \in \mathcal{B} : (Q_1 \cap Q_2) \cap \bar{B}_1 \not\subseteq B_2$$

Then, for any two sets $X$ and $Y$, $(X \not\subseteq Y) \equiv (X \cap \bar{Y} \neq \emptyset)$, and so:

$$\forall Q_1, Q_2 \in \mathcal{S}_\Pi : \forall B_1, B_2 \in \mathcal{B} : Q_1 \cap Q_2 \cap \bar{B}_1 \cap \bar{B}_2 \neq \emptyset.$$

Since $\bar{B}_i$ is a survivor set, $B_i \in \mathcal{B}$, this can be more compactly written as:

$$\forall Q_1, Q_2, S_1, S_2 \in \mathcal{S}_\Pi : Q_1 \cap Q_2 \cap S_1 \cap S_2 \neq \emptyset.$$

which is 4–Intersection. Hence, another way to write the replication requirement stated in M-Consistency is 4–Intersection, or equivalently 4–Partition.

## 4.2 An example of fractional $k$

The results of the previous section are perhaps not surprising to those who have designed consensus or quorum algorithms. For example, 2–Intersection states that the survivor sets form a coterie, and 3–Intersection states that the intersection of any two survivor sets contains a non-faulty process. It takes some effort to show that 4–Intersection is equivalent to M–consistency, and we expect that it will not be difficult to show that the $n > 5t$ requirement of Fast Byzantine Paxos [MA05] can be understood from 5–Intersection. We conjecture that it is possible to define classes of algorithms that, as for consensus [GR03], are built on top of quorums of various strengths, and whose communication requirements are easily understood in terms of $k$–Intersection. Such algorithms developed for the threshold model should easily translate into our model of non-IID failures.

Less well understood are algorithms that have fractional replication predicates. To further motivate the utility of intersection properties, we consider a problem that we call *weak leader election*. Given a synchronous system and assuming receive-omission failures (that is, a faulty process can crash or fail to receive messages), this problem requires $n > \lfloor 3t/2 \rfloor$. This lower bound is not new, but to the best of our knowledge, it has not been shown that the lower bound is tight. We show here that the bound is tight, which is a result of some theoretical value. Our primary reason for choosing this algorithm, however, is the insight we used from the intersection property to arrive at the solution.

We first specify the problem. Our specification allows for faulty (but non-crashed) processes to become elected. Such a feature is necessary because it requires more replication to detect receive-omission failures [Mul95b], and the original lower bound proof allowed such behaviors. We then discuss the lower bound on process replication for this problem using our model of dependent failures. We further provide an algorithm, showing that the lower bound is actually tight, and prove its correctness in Appendix A.

### 4.2.1 Weak leader election

Each process $p_i$ has a local boolean variable $p_i$.*elected*, where $p_i$.*elected* is false for a crashed process. Weak leader election has two safety and two liveness properties.

*Safety*: $\Box(|\{p_i \in \Pi : p_i.\mathit{elected}\}| < 2)$.

*LE-Liveness*: $\Box\Diamond(|\{p_i \in \Pi : p_i.\mathit{elected}\}| > 0)$.

*FF-Stability*: In a failure-free execution, only one process ever has *elected* set to true.

*E-Stability*: $\exists p_i \in \Pi : \Diamond\Box(\forall p_j \in \Pi : p_j.\mathit{elected} \Rightarrow (j = i))$.

These properties state that infinitely often some process elects itself (LE-liveness), and no more than one process elects itself at any time (safety). The third property states that, in a failure-free execution, only one process is ever elected. This property, however, does not rule out executions with failures in which two or more processes are elected infinitely often. We hence define E-stability.

### 4.2.2 Lower bound on process replication

In [BMST92], the following lower bound was shown. The proof was given in the context of showing a lower bound on replication for primary-backup protocols.

**Claim 4.2.1** Weak leader election for receive-omission failures requires $n > \lfloor 3t/2 \rfloor$.

**Proof:**

Assume that weak leader llection for receive-omission failures can be solved with $n = \lfloor 3t/2 \rfloor$. Partition the processes into three blocks $A$, $B$ and $C$, where $|A| = |B| = \lfloor t/2 \rfloor$ and $|C| = \lceil t/2 \rceil$. Consider an execution $E_\alpha$ in which the processes in $B$ and $C$ initially crash. From LE-liveness and E-stability, eventually a process in $A$ will be elected infinitely often. Similarly, let $E_\beta$ be an execution in which the processes in $A$ and $C$ crash. From LE-liveness and E-stability, eventually a process in $B$ will be elected infinitely often.

Finally, consider an execution $E_\gamma$ in which the processes in $A$ fail to receive all messages except those sent by processes in $A$, and the processes in $B$ fail to receive all messages except those sent by processes in $B$. This execution is indistinguishable from $E_\alpha$ to the processes in $A$ and is indistinguishable from $E_\beta$ to the processes in $B$. Hence, there will eventually be two processes, one in $A$ and one in $B$, elected infinitely often, violating either safety or E-stability.

□

To develop the algorithm, we first generalize the replication predicate for this problem using cores and survivor sets. From the lower bound proof, we consider any partition of the processes into three blocks. Then, one constructs three executions, where in each execution all of the processes in two of the three subsets are faulty. The conclusion of the proof is the following property for $k = 3$:

**Property 4.2.2** **($k$,$k-1$)-Partition**, $k > 1$, $|\Pi| > 2$: $\exists k' \in \{2, \ldots, \min(k, |\Pi|)\}$ : $\forall \mathcal{A} \in \mathcal{P}_{k'}(\Pi) : \exists \mathcal{A}' \in \mathcal{G}_{k'-1}(\mathcal{A}) : \exists C \in \mathcal{C}_\Pi : C \subseteq \bigcup \mathcal{A}'$ $\square$

($k$,$k-1$)-Partition says that for any partition of the set of processes in to $k'$ blocks, $k' \in \{2, \ldots, \min(k, |\Pi|)\}$, the union of some $k'-1$ blocks contains a core. The equivalent intersection property is:

**Property 4.2.3** **($k$,$k-1$)-Intersection**, $k > 1$, $|\Pi| > 2$, $|\mathcal{S}_\Pi| > 2$: $\exists k' \in \{2, \ldots, \min(k, |\Pi|)\}$ : $\forall \mathcal{T} \in \mathcal{G}_{k'}(\mathcal{S}_\Pi) : \exists T \in \mathcal{G}_2(\mathcal{T}) : (\cap_{S \in T} S) \neq \emptyset$ $\square$

Stated more simply, ($k$,$k-1$)-Intersection says that for any set of $k'$ survivor sets, $k' \in \{2, \ldots, \min(k, |\Pi|)\}$, at least two of them have a non-empty intersection. ($k$,$k-1$)-Intersection and ($k$,$k-1$)-Partition generalize replication predicates in the threshold model of the form $n > \lfloor kt/(k-1) \rfloor$. Thus, a profile that satisfies ($k+1$,$k$)-Intersection must also satisfy ($k$,$k-1$)-Intersection. To illustrate, a system profile satisfies (3,2)-Intersection if either it satisfies (2,1)-Intersection or, for every three survivor sets, two intersect. Also, note that (2,1)-Intersection is 2–Intersection.

**Example 4.2.4**

$$\begin{aligned}
\Pi &= \{p_{a_1}, p_{a_2}, p_{a_3}, p_{b_1}, p_{b_2}, p_{b_3}\} \\
\mathcal{C}_\Pi &= \{\{p_{i_1}, p_{i_2}, p_{i_3}, p_{i_4}\} : (i_1, i_2 \in \{a_1, a_2, a_3\}) \wedge (i_3, i_4 \in \{b_1, b_2, b_3\} \wedge i_1 \neq i_2 \wedge i_3 \neq i_4)\} \\
\mathcal{S}_\Pi &= \{\{p_{i_1}, p_{i_2}\} : ((i_1, i_2 \in \{a_1, a_2, a_3\}) \vee (i_1, i_2 \in \{b_1, b_2, b_3\})) \wedge i_1 \neq i_2\};
\end{aligned}$$

Consider now an example of a system that satisfies (3,2)-Intersection. It is based on a simple two-cluster system. A process can fail by crashing, and there is a threshold $t$ on the number of crash failures that can occur in a cluster. A cluster can suffer a total failure, which causes all of the processes in that cluster to crash. A total failure results from the failure of a shared resource such as storage, for example. We assume that total failures are rare enough that the probability of both clusters suffering total failures is negligible. However, processes can crash in one cluster at the same time that the other cluster suffers a total failure. Assuming that each cluster has three processes and $t = 1$, we have the system profile of Example 4.2.4, where processes with identifier $a_i$ are in one cluster and processes with identifier $b_i$ are in the other cluster. Note that this profile satisfies (3,2)-Intersection because out of any three survivor sets, at least two intersect.

We now show the equivalence of ($k$,$k-1$)-Partition and ($k$,$k-1$)-Intersection with the following theorem.

**Theorem 4.2.5** *($k$,$k-1$)-Partition $\equiv$ ($k$,$k-1$)-Intersection*

**Proof:**

$\Rightarrow$: Proof by contrapositive. Consider a system profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ such that, for every $k'$, $k' \in \{2, \ldots, \min(k, |\Pi|)\}$, there is a subset $\mathcal{S} = \{S_1, S_2, \ldots, S_{k'}\} \subseteq \mathcal{S}_\Pi$ such that $S_i \cap S_j = \emptyset$, $i \neq j$. That is, $\bigcup_{P \in \mathcal{G}_2(\mathcal{S})} \bigcap P = \emptyset$. For every $k'$, we then build a partition $\mathcal{A} = \{A_1, A_2, \ldots, A_{k'}\}$ as follows:

$$\begin{aligned}
A_1 &= \Pi \setminus (S_2 \cup S_3 \cup \ldots \cup S_{k'}) \\
A_2 &= \Pi \setminus (S_1 \cup S_3 \cup \ldots \cup S_{k'} \cup A_1) \\
&\vdots \\
A_i &= \Pi \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1} \cup S_{i+1} \cup \ldots \cup S_{k'} \cup A_1 \cup A_2 \ldots \cup A_{i-1}) \\
&\vdots \\
A_{k'} &= \Pi \setminus (S_1 \cup S_2 \cup \ldots S_{k'-1} \cup A_1 \ldots \cup A_{k'-1})
\end{aligned}$$

It is clear that $A_i$, $A_j$ are disjoint, $i \neq j$. We now have to show that: 1) $\bigcup \mathcal{A} = \Pi$; 2) For every subset $\mathcal{A}' = \{A_{i_1}, A_{i_2}, \ldots, A_{i_{k'-1}}\} \subset \mathcal{A}$, $\bigcup \mathcal{A}'$ does not contain a core. To show 1), let $\psi_i = \cup_{(S_j \in \mathcal{S} \setminus S_i)} S_j$, $i \in \{1, \ldots, k'\}$. We then have the following derivation:

$$\bigcup \mathcal{A} \;=\; (\Pi \setminus \psi_1) \cup (\Pi \setminus \psi_2 \cup A_1) \cup \ldots$$

$$
\begin{aligned}
& \cup(\Pi \setminus (\psi_{k'} \cup A_1 \cup A_2 \ldots \cup A_{k'-1})) && (4.1)\\
= \;& \Pi \setminus ((\psi_1 \cap (\psi_2 \cup A_1)) \cap \ldots \cap (\psi_{k'} \cup A_1 \cup A_2 \ldots \cup A_{k'-1})) && (4.2)\\
= \;& \Pi \setminus (\psi_1 \cap \psi_2 \cap \ldots \cap (\psi_{k'} \cup A_1 \cup A_2 \ldots \cup A_{k'-1})) && (4.3)\\
\vdots \;& \\
= \;& \Pi \setminus (\cap_i \psi_i) && (4.4)\\
= \;& \Pi && (4.5)
\end{aligned}
$$

- Line 4.1 to Line 4.2 follows from the observation that for any subsets $A, B$ of $\Pi$, we have that $(\Pi \setminus A) \cup (\Pi \setminus B) = \Pi \setminus (A \cap B)$;

- Line 4.2 to Line 4.3: the intersection between $\psi_1$ and $A_1$ has to be empty, since $\psi_1$ contains exactly the elements we removed from $\Pi$ to form $A_1$.

- Line 4.3 to Line 4.4: by repeating inductively the process used to derive Line 4.3, we are able to remove every term $A_i$ present in the equation.

- Line 4.4 to Line 4.5: Transforming from a conjunctive form to a disjunctive form, we have that $\bigcap_{P \in \mathcal{G}_{k'-1}(\mathcal{S})} \bigcup P = \bigcup_{P \in \mathcal{G}_2(\mathcal{S})} \bigcap P$. To see why this is true, note that for every pair $S_i, S_j \in \mathcal{S}$, $i \neq j$, and $P \in \mathcal{G}_{k'-1}(\mathcal{S})$, we have that $(S_i \in P) \vee (S_j \in P)$. Finally, we have that $\bigcup_{P \in \mathcal{G}_2(\mathcal{S})} (\cap_{S_i \in P} S_i) = \emptyset$ by assumption.

By the construction of the partition and from the assumption that for every $S_i, S_j \in \mathcal{S}$, $S_i \cap S_j = \emptyset$, we have that for every $i \in \{1, \ldots, k'\}$, there is $S_i \in \mathcal{S}$ such that $S_i \subseteq A_i$. From this, we conclude that for any $\mathcal{A}' = \{A_{i_1}, A_{i_2}, \ldots, A_{i_{k'-1}}\} \subset \mathcal{A}$, $\bigcup \mathcal{A}'$ does not contain elements from some survivor set, and consequently it does not contain a core.

$\Leftarrow$: Proof also by contrapositive. Suppose a system profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ such that for every $k'$, $k' \in \{2, \ldots, \min(k, |\Pi|)\}$, there is a partition $\mathcal{A} = \{A_1, A_2, \ldots, A_{k'}\}$ of $\Pi$ in which no union of $k'-1$ blocks of $\mathcal{A}$ contains a core. If a subset of processes does not contain a core, then it contains no elements from some survivor set. The complement of such a set of processes consequently contains a survivor set. Because no union of $k' - 1$ blocks in $\mathcal{A}$ contains a core, for every $A_i$ there is an $S_i \in \mathcal{S}_\Pi$ such that $S_i \subseteq A_i$. Thus, for all $A_i, A_j \in \mathcal{A}$, $i \neq j$, we have by construction that $A_i \cap A_j = \emptyset$, and hence $S_i \cap S_j = \emptyset$. We conclude that no pair $S_i, S_j \in \{S_1, S_2, \ldots, S_{k'}\}$ is such that $S_i \cap S_j \neq \emptyset$.

□

### 4.2.3 A weak leader election algorithm

We now develop a synchronous algorithm WLE for weak leader election. For this algorithm, we assume a system profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ that satisfies (3,2)-Intersection. WLE is round based: in each round a process receives messages sent in the previous round and then sends messages to all processes. We use $p_i.M(r)$ to denote the set of messages that $p_i$ receives in round $r$, and $p_i.s(r)$ to denote the set of processes from which process $p_i$ receives messages in round $r$.

We developed this algorithm by first observing what (3,2)-Intersection means. Given three survivor sets, at least two of them intersect. Put another way, if two survivor sets $S_1$ and $S_2$ are disjoint, then any survivor set $S_3$ intersects $S_1 \cup S_2$. Since a core is a minimal set that intersects every survivor set, the above implies that $S_1 \cup S_2$ contains a core. Thus, given any two disjoint survivor sets, at least one of them contains a correct process.

Our algorithm uses as a building block a weak version of uniform consensus that we call *RO consensus*. We call it RO consensus because of its resemblance to uniform consensus. RO consensus, however, is tailored to suit the requirements of WLE and therefore is fundamentally different.

In RO consensus, each process $p_i$ has an initial value $p_i.a \in V \cup \{\bot\}$, where $V$ is the set of initial values, and a decision value $p_i.d\,[1 \ldots n]$, where $p_i.d$ is a list and $p_i.d[j] \in V \cup \{\bot\}$. We use $v \in p_i.d$ to denote that there is some $p_\ell \in \Pi$ such that $p_i.d[\ell] = v$. If a process $p_i$ crashes, then we assume that its decision value $p_i.d$ is $\mathcal{N}$, where $\mathcal{N}$ stands for the $n$ element list $[\bot, \ldots, \bot]$. To avoid repetition throughout the discussion of our algorithm, we say that a process $p_i$ decides in an execution $E$ if $p_i.d \neq \mathcal{N}$.

As we describe later, we execute our algorithm for RO consensus, called ROC, two times in electing a leader. We then have that processes may crash before starting an execution $E$ of ROC. Such processes hence have initial value undefined in $E$. We therefore use $\perp$ to denote the initial value of crashed processes. That is, if $p_i.a = \perp$, then $p_i$ has crashed. We also use the relation $x \subseteq y$ for $x$ and $y$ lists of $n$ elements to denote that: $\forall i, 1 \leq i \leq n : (x[i] \neq \perp) \Rightarrow (x[i] = y[i])$.

The specification of RO consensus is composed of four properties as follows:

**Termination:** Every process that does not crash eventually decides on some value;

**Agreement:** If $p_i.d[\ell] \neq \perp$, $p_i, p_\ell \in \Pi$, then for every non-faulty $p_c$, $p_i.d[\ell] = p_c.d[\ell]$;

**RO Uniformity:** Let *vals* be the following set:[2] $\{d : \exists p_i \in \Pi \textit{ s.t. } (p_i.d = d)\} \setminus \mathcal{N}$. Then:

$\bigwedge \; 1 \leq |vals| \leq 2$

$\bigwedge \; \forall d, d\prime \in vals : d \subseteq d\prime \vee d\prime \subseteq d$

$\bigwedge \; \forall d_f, d_c \in \textit{vals}, d_f \subseteq d_c : \exists S_f, S_c \in \mathcal{S}_\Pi$:

$\quad \wedge \, \forall p \in S_f : (p \text{ crashes}) \;\vee\; (p.d = d_f)$

$\quad \wedge \, \forall p \in S_c : (p.d = d_c) \;\wedge\; (p \text{ is not faulty})$

That is, there can be no more than two non-$\mathcal{N}$ decision values, and if there are two then one is a subset of the other. Furthermore, if there are two different decision values, then these are the values that processes in two disjoint survivor sets decide upon, one for the processes of each survivor set.

[2]We use the “bulleted conjunction” and the “bulleted disjunction” notation list invented in TLA$^+$ [Lam02]. In Definition 5.2.1, the list corresponds to the conjunction of the statements to the right of the “$\bigwedge$” marks.

**Validity:** $\bigwedge$ If $p_j \in \Pi$ does not crash, then for all non-faulty $p_i$, $p_i.d[j] = p_j.a$

$\bigwedge$ If $p_j \in \Pi$ does crash, then for every non-faulty $p_i$, $p_i.d[j] \in \{\perp, p_j.a\}$

$\bigwedge$ If there are survivor sets $S_c, S_f \in \mathcal{S}_\Pi$ and values $v_c, v_f \in V$,

$v_c \neq v_f$, such that: $\wedge\ \forall p \in S_f : p.a \in \{v_f, \perp\}$

$\wedge\ \forall p \in S_c : ((p.a = v_c)\ \wedge\ (p \text{ is not faulty}))$

$\wedge\ \exists p_i, p_\ell \in \Pi : p_i.d[\ell] = v_f$

then for all $p_j$ that does not crash, $v_f \in p_j.d$

That is, if a process $p_i$ is not faulty and $p_i.d[j] \neq \perp$, then the value of $p_i.d[j]$ must be $p_j.a$. The value of $p_i.d[j]$, however, can be $\perp$ only if $p_j$ crashes. The third case exists because we use the decision values of an execution as the initial values for another execution. From RO uniformity, there can be two different non-$\mathcal{N}$ values $d_f$ and $d_c$. If this is the case, then there is a survivor set $S_c$ containing only correct processes such that all processes in $S_c$ decide upon $d_c$, and another survivor set $S_f$ containing only faulty processes such that all the processes in $S_f$ either crash or decide upon $d_f$. Let $v_f$ be $d_f$ and $v_c$ be $d_c$. By the third case, if some process that decides includes $v_f$ in its decision value, then every process that does not crash also includes $v_f$ in its decision value.

Figure 4.1 shows our algorithm ROC. In each round $r$, a process $p_i$ collects messages and updates its list of initial values $p_i.A$. Once it updates $p_i.A$, $p_i$ sends a message containing $p_i.A$ to all processes. A process $p_i$ also assigns $p_i.A$ to $p_i.A_p(r)$ once it updates $p_i.A$ at round $r$. This enables $p_i$ to verify in round $r + 2$ if a process $p_j$ has received the message $p_i$ sent in round $r$. As we describe below, $p_i$ uses $p_i.A_p(r)$ to determine if it is faulty.

ROC is an adaptation of a classic round-based synchronous consensus algorithm for crash failures. There are two main differences. First, it uses survivor sets rather than a threshold scheme. It does use a constant $t$ to bound the number of rounds; $t$ is the number of processes subtracted the size of the smallest survivor set. Second, it has each process verify if it has committed receive-omission failures.

There are two ways that a process can notice that it has committed an omission

```
**Algorithm** ROC on input $p_i.a$, $p_i.Procs$
**round** 0:
    $p_i.s(0) \leftarrow p_i.Procs$; $p_i.sr(0) \leftarrow p_i.s(0)$
    $p_i.A\,[i] \leftarrow p_i.a$
    for all $p_k \in \Pi$, $p_k \neq p_i$ : $p_i.A\,[i] \leftarrow \perp$
    $p_i.A_p(0) \leftarrow p_i.A$
    send $p_i.A$ to all
**round** 1:
    $p_i.sr(1) \leftarrow p_i.s(1)$
    **if** $\vee$ $p_i.s(1) \not\subseteq p_i.s(0)$
      $\vee$ $\nexists S \in \mathcal{S}_\Pi : S \subseteq p_i.sr(1)$
    **then** decide $\mathcal{N}$
    **else** for each $m \in p_i.M(1)$, $p_k \in \Pi$:
            **if** $(p_i.A\,[k] = \perp)$ $p_i.A\,[k] \leftarrow m.A\,[k]$
    $p_i.A_p(1) \leftarrow p_i.A$
    send $p_i.A$ to all
**round** $r$: $2 \leq r \leq t$:
    $p_i.sr(r) \leftarrow p_i.s(r) \setminus \{p_j : \exists m \in p_i.M(r) :$
        $p_i.A_p(r-2) \not\subseteq m.A \wedge m.\mathit{from} = p_j\}$
    **if** $\vee$ $p_i.s(r) \not\subseteq p_i.s(r-1)$
      $\vee$ $\nexists S \in \mathcal{S}_\Pi : S \subseteq p_i.sr(r)$
    **then** decide $\mathcal{N}$
    **else** for each $m \in p_i.M(r)$, $p_k \in \Pi$:
            **if** $(p_i.A\,[k] = \perp)$ $p_i.A\,[k] \leftarrow m.A\,[k]$
    $p_i.A_p(r) \leftarrow p_i.A$
    send $p_i.A$ to all
**round** $t+1$:
    $p_i.sr(t+1) \leftarrow p_i.s(t+1) \setminus$
        $\{p_j : \exists m \in p_i.M(t+1) :$
        $p_i.A_p(t-1) \not\subseteq m.A \wedge m.\mathit{from} = p_j\}$
    **if** $\vee$ $p_i.s(t+1) \not\subseteq p_i.s(t)$
      $\vee$ $\nexists S \in \mathcal{S}_\Pi : S \subseteq p_i.sr(t+1)$
    **then** decide $\mathcal{N}$
    **else** for each $m \in p_i.M(t+1)$, $p_k \in \Pi$:
            **if** $(p_i.A\,[k] = \perp)$ $p_i.A[k] \leftarrow m.A[k]$
    $p_i.A_p(t+1) \leftarrow p_i.A$
    decide $p_i.A$
```

**Figure 4.1:** ROC - Algorithm run by process $p_i$

failure:

1. Processes that have not decided or crashed send messages to all processes. We then have that for every non-faulty $p_i$ that receives messages in rounds $r$ and $r+1$: $p_i.s(r+1) \subseteq p_i.s(r)$. If this does not hold, then $p_i$ must have failed to receive some message;

2. Consider a message $m$ that $p_i$ receives from $p_j$ in round $r > 1$. Unless it crashes or discovers that it is faulty, a process sends a message to all processes in each round except the last. Let $m'$ be the message that $p_i$ sent to $p_j$ in round $r-2$. If $m$ indicates that $p_j$ has not received $m'$ ($p_i.A_p(r-2) \not\subseteq m.A$), then $p_i$ knows that $p_j$ is faulty. Let $p_i.sr(r)$ be the processes in $p_i.s(r)$ with all processes that $p_i$ knows to be faulty removed. By definition, we know that there is some survivor set that contains only correct processes. If $p_i.sr(r)$ does not contain a survivor set, then there is some correct process from which $p_i$ did not receive a message. Hence, $p_i$ can conclude that it has failed to receive a message.

Note that RO consensus differs from the definition of uniform consensus in that faulty processes may decide upon different values, although these values are not arbitrary and must be such as described by the RO Uniformity property. In the algorithm by Parvèdy and Raynal, for example, every process that decides must decide upon the same value [PR04].

Informally, ROC satisfies RO Uniformity because (3,2)-Intersection holds. To decide on a value other than $\mathcal{N}$, a process must receive in each round messages from a set of processes that contains a survivor set. (3,2)-Intersection implies a low enough replication that there can be a set $S$ of non-crashed faulty processes that communicate only among themselves. But, there cannot be two such sets $S$ and $S'$: if $S$ and $S'$ do not intersect, then (3,2)-Intersection implies that their union contains a core, and there must be a correct process either in $S$ or in $S'$.

A set $S$ of faulty processes that communicate only among themselves will decide on a value $d$ where $d[i] = \perp$ for $p_i \notin S$ and $d[i] = p_i.a$ for $p_i \in S$. In addition, a

```
**Algorithm** WLE
$P \leftarrow \Pi$
**repeat** {
$p_i$.elected $\leftarrow$ FALSE
**Phase** 1:
  Run ROC with
      $p_i.a \leftarrow i$; $p_i.Procs \leftarrow P$
  $P \leftarrow p_i.s(t+1)$
  **if** ($p_i.d = [\bot, \ldots, \bot]$) **then** stop
**Phase** 2:
  Run ROC with
      $p_i.a \leftarrow p_i.d$ from Phase 1; $p_i.Procs \leftarrow P$
  $P \leftarrow p_i.s(t+1)$
  **if** ($p_i.d = \mathcal{N}$) **then** stop
  let $x \in p_i.d$ be a value such that
    $\bigwedge p_i.d\,[x] \neq \mathcal{N}$
    $\bigwedge p_i.d$ has the least number of non-$\bot$ values
  **if** ($p_i$ is the first index of $x$ such that $x[i] \neq \bot$)
    **then** $p_i$.elected $\leftarrow$ TRUE
}
```

**Figure 4.2:** WLE - Algorithm run by process $p_i$

correct process will also decide $d[i] = p_i.a$ for $p_i \in S$. Of course, a non-crashed faulty process can read from different sets of processes in each round, but by using the two rules given above, such a process can determine that it is faulty. Hence, at worst some faulty processes will decide on a value $d_f$ and the correct processes will decide on a value $d_c$ such that $d_f \subseteq d_c$.

The algorithm in Figure 4.2 uses ROC to implement weak leader election. Algorithm WLE proceeds in iterations of an infinite repeat loop, where each iteration consists of two phases. In Phase 1, processes use ROC to distribute their process identifiers. In Phase 2, they use ROC to distribute what they decided on in Phase 1.

Appendix A presents a formal proof of correctness for WLE, and here we only present an informal argument for the correctness of WLE. Informally, this algorithm satisfies safety because of the following: it is possible for a set of faulty processes $S$ to decide on the smaller value $d_f$ in Phase 1, but by the end of Phase 2 the correct processes will know this. By Validity and RO Uniformity, every process that finishes Phase 2 uses the same list $d_f$ to determine whether it is the current leader or not. Having the processes decide based on the smaller list $d_f$ forces the receive-omission faulty processes to elect the same process as the correct processes. Note though, as mentioned above, that in this case the correct processes know that the elected process is faulty (although the elected process does not know). LE-liveness is obtained by repeatedly running the algorithm without resorting to a failure detector (which would require higher replication). If there are no faulty processes, each election will always elect the process with the lowest identifier, which implies FF-stability. To guarantee that there is no alternating behavior in which two processes are leaders infinitely often, non-crashed processes move forward the set of processes they believe are not crashed or have not stopped. That is, the input $p_i$.*Procs* in ROC takes the value $p_i.s(t+1)$ from the previous execution of ROC ($\Pi$ if it is the first execution of ROC). This implies E-stability.

## 4.3 Conclusions

This chapter generalized a common argument used in proofs of lower bounds on process replication. The argument is based on the threshold model: it makes the assumption that, given $n$ processes, any subset of $\lceil nb/k \rceil$ processes can be faulty. Then, after deriving a contradiction, the proof concludes that $n > \lfloor kt/b \rfloor$. In our generalization of the proof for $b = 1$, we conclude that $k$–Partition holds: if one partitions the processes into $k$ subsets, then at least one of the subsets contains a core. Thus, lower bounds for many protocols can be trivially generalized for when process failures are not IID. We then gave an equivalent property, $k$–Intersection, that is often useful when designing a protocol that takes advantage of non-IID process failures.

We considered a problem for which the lower bound has $b = 2$. The lower bound on process replication for weak leader election in a synchronous system with receive-omission failures was known to be $n > \lfloor 3 \cdot t/2 \rfloor$, but this bound was not known to be tight. We showed that this bound is tight by first determining the intersection property for this replication predicate (($k$,$k-1$)-Intersection, equivalent to ($k$,$k-1$)-Partition, $k = 3$) and using it to guide our development of a protocol.

# Chapter 5

# Coteries in multi-site systems

The previous chapters discussed theoretical applications of the core/survivor set model. This chapter is the first one to discuss the application of the core/survivor set model to realistic settings. We consider replication techniques for a specific type of system: wide-area systems composed of geographically dispersed sites. In particular, we consider quorum systems as a general replication technique and we use the survivor set abstraction to derive quorum systems with optimal availability.

There has been a proliferation of large distributed systems that support a diverse set of applications such as sensor nets, data grids, and large simulations. Such systems consist of multiple sites connected by a wide-area network, where a site is a collection of computing nodes running one or more processes. The sites are often managed by different organizations, and the systems are large enough that site and process failures are common facts of life rather than rare events.

Critical services in such systems can be made highly available using replication. In data grids, for example, data sets are the most important assets, and having them available under failures of sites is very desirable. To improve availability, the well-known *quorum update* technique can be used. This technique consists of implementing a mutual exclusion mechanism by reading and writing to sets of processes that intersect (*quorums*) [GMB85]. As another example, the Paxos protocol [Lam98] enables the implementation of fault-tolerant state machines for asynchronous systems. Paxos is a popular choice because of its ability to produce results when a majority of replicas survive, for its feature of not producing erroneous results when failures of more than a majority (indeed, up to a complete failure) occur, and its very weak assumptions about the environment. Underlying Paxos (and other similar protocols) is the same quorum update technique.

This chapter considers quorum constructions for multi-site systems. The problem area of quorums for multi-site systems is large and not well studied. We address a set of problems from this area as an early foray. We first give a failure model for multi-site systems that is simple and has intuitive appeal, and then give a second failure model that has less intuitive appeal but theoretical and practical interest. Because sites can fail, the failures of processes are not independent, and so an IID (independent, identically distributed) model is not ideal. We define a new metric for availability that is suitable to non-IID failures, and give optimal quorum constructions for both models. Using this metric, however, assumes that the survivor sets are known, and is hence not as simple as counting the minimum number of nodes that affect all quorums in a quorum system. We then discuss the implementability of the two failure models, and discuss an experiment of running Paxos on PlanetLab [pla06] that gives some validation of our results.

## 5.1 Related work

Quorum systems have been studied for over two decades. The first algorithms based on quorums use voting [Gif79]. Garcia-Molina and Barbara generalized the notion of voting mechanisms, and proposed the use of minimal collections of intersecting sets, or *coteries* [GMB85]. Most of the following work (such as [Kum91, Mae85, PW95b]) has concentrated on how quorums can be constructed to give good availability, load and capacity assuming relatively simple system properties (such as identical processes and independent failures) [AW98a, BGM86, NW98]. Only recently the problem of choosing quorums according to properties of the system (such as location) has attracted some attention [GM04]. Of particular interest to our work are the constructions of [Kum91] and [BBB$^+$04]. In [Kum91], Kumar proposed, to the best of our knowledge, the first hierarchical quorum construction, and showed that by doing so one can have smaller quorums. The analysis in [Kum91], however, assumes IID failures. The work by Busca *et al.* assumes a multi-site system similar to what we assume here, and their quorum construction [BBB$^+$04] is very similar to our *Qsite* construction. Their focus, however, was on performance. If one considers the distribution of response times from a quorum system, performance is often measured using the average or median, while availability is a property of the tail of the distribution. Thus, high performance does not necessarily imply high availability. Availability in quorum systems has been studied before [AW98a, BGM86, NW98], but we argue here that the previous metrics are not suitable for multi-site systems. A notable exception is the work by Amir and Wool [AW96], which evaluates several existing quorum constructions in the context of a small, real network.

A *network partition* is a failure event that leads to one set of non-faulty processes being unable to communicate with another set of non-faulty processes (and, often, vice versa). Quorum systems are asynchronous, and so a network partition is treated identically to slow-to-respond processes. Long-lasting network partitions can make it impossible to obtain a quorum. A recent paper by Yu presents a probabilistic construction that does increase availability in the face of partitions, but it assumes a uniform distribution of servers across the network [Yu04]. In comparison, our constructions are deterministic and make no assumption about distributions of sites.

## 5.2 System model

We consider a system of a set $\Pi$ of processes. The processes are partitioned into *sites* $\mathcal{B} = \{B_1, B_2, \ldots, B_k\}$, and between each pair of processes there is a bidirectional communication channel. Processes can fail by crashing, and a crashed process can recover. Similarly, a site can fail and recover. A site failure represents the loss of a key resource used by the processes in the site (such as network, power, or a storage server) or some event that causes physical damage to the equipment on the site (such as loss of A/C); the processes in the site are all effectively crashed while the site is faulty.

Let $\mathcal{E}$ denote the executions of the system as before. Recall that each execution $E = \langle \mathit{Init}, \mathit{Steps}, \mathit{Time}, \mathit{Faulty} \rangle \in \mathcal{E}$ is a sequence of steps of processes, and there is a system state (a collection of states from all processes) associated to each step. Each step $s \in \mathit{Steps}$ of $E$ has an associated *failure pattern* $\mathit{Faulty}(s) \subseteq \Pi$, which is the set of processes that are faulty in $s$. If site $B_i$ is faulty in $s$, then all of the processes in $B_i$ are in $\mathit{Faulty}(s)$. Because we do not make use of all elements of executions as defined in Section 2, we use as shortcuts the following notation for an execution $E = \langle \mathit{Init}, \mathit{Steps}, \mathit{Time}, \mathit{Faulty} \rangle$:

- $s \in E$ to denote a step $s$ that is the set *Steps* of steps of $E$;
- $F(s, E)$ to denote $\mathit{Faulty}(s)$, where $s \in E$. That is, the set of faulty processes of step $s$ of execution $E$;
- $\mathit{NF}(s, E) = \Pi \setminus \mathit{Faulty}(s)$, where $s$ is a step of $E$. That is, the set of non-faulty processes of a step $s$ of $E$.

We say that a failure pattern $f$ is valid iff $\exists E \in \mathcal{E} : \exists s \in E : f = F(s, E)$.

We use survivor sets to express valid failure patterns. Recall that, informally, a survivor set is a minimal subset of non-faulty processes. This definition does not rely on probabilities directly, although failure probabilities can be used to determine survivor sets; we postpone this discussion until later in this chapter. The definition is:

**Definition 5.2.1** Given a set of processes $\Pi$, a set $S$ is a survivor set if and only if:

$$\begin{array}{l} \bigwedge S \subseteq \Pi \\ \bigwedge \exists E = \langle \mathit{Init}, \mathit{Steps}, \mathit{Time}, \mathit{Faulty} \rangle \in \mathcal{E} : \exists s \in E : S = \mathit{NF}(s, E) \\ \bigwedge \forall p \in S : \forall E = \langle \mathit{Init}, \mathit{Steps}, \mathit{Time}, \mathit{Faulty} \rangle \in \mathcal{E} : \forall s \in E : S \setminus \{p\} \neq \mathit{NF}(s, E) \end{array}$$

We use $\mathcal{S}_\Pi$ to denote the set of survivor sets of $\Pi$, and we call a pair $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ a *system profile*. Note that in this chapter, a system profile does not include a set of cores.

We do so because the constructions we present in the following sections do not make use of cores.

We now repeat a few definitions that have appeared elsewhere in the literature and that we use in this chapter. A coterie $\mathcal{Q}$ is a set of subsets of $\Pi$ that satisfies the following two properties [GMB85]: 1) $\forall Q_i, Q_j \in \mathcal{Q} : Q_i \cap Q_j \neq \emptyset$; 2) $\forall Q_i, Q_j \in \mathcal{Q}, Q_i \neq Q_j : Q_i \not\subset Q_j \wedge Q_j \not\subset Q_i$. The first property is 2–Intersection (Chapter 4), and it says that quorums in a coterie pairwise intersect. This property guarantees mutual exclusion when executing operations on quorums, such as reads and writes, as every pair of quorums must have at least one process in common. The second property states that all quorums are minimal. A coterie $\mathcal{Q}$ is *dominated* if there is a coterie $\mathcal{Q}'$ such that: 1) $\mathcal{Q} \neq \mathcal{Q}'$; 2) $\forall Q \in \mathcal{Q} : \exists Q' \in \mathcal{Q}' : Q' \subseteq Q$. If no coterie dominates a coterie $\mathcal{Q}$, then we say that $\mathcal{Q}$ in *non-dominated*.

A *transversal* of a coterie is a subset of processes that intersects every quorum in the coterie. We use $\mathcal{T}(\mathcal{Q})$ to denote the set of transversals of the coterie $\mathcal{Q}$. Transversals are useful for defining the availability of a coterie: a coterie $\mathcal{Q}$ is available in a step $s$ of some execution $E$ if and only if $F(s, E) \notin \mathcal{T}(\mathcal{Q})$.

## 5.3 Computing availability

The availability of coteries can be computed in various ways. One metric is *node vulnerability* which is the minimum number of nodes that, if removed, make it impossible to obtain a quorum [BGM86]. A similar metric, *edge vulnerability*, counts the minimum number of channels whose removal makes it impossible to obtain a quorum (no connected component contains a quorum). Both of these metrics are appropriate when failures are independent and identically distributed (IID) because they measure the minimum number of failures necessary to halt the system. They are not necessarily good metrics for multi-site systems. Consider the following three-site system in which a survivor set is the union of majorities of processes in a site for some majority of sites:[1]

[1]We use $x_1 x_2 \ldots x_n$ as a short notation for the set $\{x_1, x_2, \ldots, x_n\}$.

$$\begin{aligned}
\Pi &= \{a_1, a_2, a_3, b_1, b_2, b_3, c_1, c_2, c_3\} \\
\mathcal{B} &= \{a_1a_2a_3, b_1b_2b_3, c_1c_2c_3\} \\
\mathcal{S}_\Pi &= \{a_ia_jb_lb_m : i, j, l, m \in \{1, 2, 3\} \wedge\ i \neq j \wedge l \neq m\} \\
&\cup \{a_ia_jc_lc_m : i, j, l, m \in \{1, 2, 3\} \wedge\ i \neq j \wedge l \neq m\} \\
&\cup \{b_ib_jc_lc_m : i, j, l, m \in \{1, 2, 3\} \wedge\ i \neq j \wedge l \neq m\}
\end{aligned}$$

From our system model, processes are pairwise connected. According to the results in [BGM86], the best strategy for both node and edge vulnerability is then to use quorums formed of majorities, which for this system is any subset of five processes. By definition, for every $S \in \mathcal{S}_\Pi$, there is some step $s$ of some execution $E \in \mathcal{E}$ such that $S = \mathit{NF}(s, E)$, where $\mathcal{E}$ is the set of executions of $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$. As $\Pi$ contains nine processes and every $S \in \mathcal{S}_\Pi$ contains four processes, there are five faulty processes in such a step, and hence no majority quorum can be obtained. If one uses $\mathcal{S}_\Pi$ as a coterie, however, then there is one quorum available in every step, by construction. $\mathcal{S}_\Pi$ has therefore better availability than the majority construction.

An alternative to node and edge vulnerability is, given probabilities of failures, to directly compute the probability of the most likely failure patterns that make it impossible to obtain a quorum. Probability models, however, can become quite complex when failures are not IID. To avoid such complexity, we use a different counting metric: the number of survivor sets that allow a quorum to be obtained. More carefully:

**Definition 5.3.1** Let $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ be a system profile and $\mathcal{Q}$ be a coterie over $\Pi$. The availability of $\mathcal{Q}$ is given by: $\mathcal{A}(\mathcal{Q}) = |\{S : S \in \mathcal{S}_\Pi \wedge (\Pi \setminus S) \notin \mathcal{T}(\mathcal{Q})\}|$

A coterie $\mathcal{Q}$ *covers* a survivor set $S$ if there is a quorum $Q \in \mathcal{Q}$ such that $Q \subseteq S$. For some survivor set $S$, if $(\Pi \setminus S) \notin \mathcal{T}(\mathcal{Q})$ holds, then $S$ must contain a quorum of $\mathcal{Q}$. That is, $\mathcal{Q}$ covers $S$. According to the definition, $\mathcal{A}(\mathcal{Q})$ is hence the number of survivor sets that $\mathcal{Q}$ covers. Note, however, that if $\mathcal{Q}$ is dominated, then the $S \in \mathcal{T}(\mathcal{Q})$ may not imply that $(\Pi \setminus S) \notin \mathcal{T}(\mathcal{Q})$ (due to Lemma 2.8 of [PW95a]). This observation implies that we cannot simply replace $(\Pi \setminus S) \notin \mathcal{T}(\mathcal{Q})$ with $S \in \mathcal{T}(\mathcal{Q})$ to obtain an equivalent definition.

This is a good metric because in every step $s$ of an execution $E$, there is at least one survivor set in $\mathcal{S}_\Pi$ that does not intersect $F(s, E)$. If a coterie allows a quorum to be obtained for more survivor sets, then this coterie is available during more steps. As with node vulnerability and edge vulnerability, $\mathcal{A}()$ is a deterministic metric and as such has a similar limitation with respect to probabilities. If we assign probabilities of failure to subsets of processes, then our metric may lead to wrong conclusions, as there might be higher available coteries that include discarded survivor sets. For the constructions and examples we discuss in this chapter, however, using this metric gives us coteries with optimal availability.

If a coterie $\mathcal{Q}$ is dominated, then by definition there is some other coterie $\mathcal{Q}'$ that dominates $\mathcal{Q}$. Under reasonable assumptions, the availability of $\mathcal{Q}'$ is at least as high as the availability of $\mathcal{Q}$. Thus, we use domination to break ties between coteries that cover the same number of survivor sets. We say that $\mathcal{Q} \prec_a \mathcal{Q}'$ iff:

$$\begin{array}{l} \bigvee \mathcal{A}(\mathcal{Q}') > \mathcal{A}(\mathcal{Q}) \\ \bigvee (\mathcal{A}(\mathcal{Q}') = \mathcal{A}(\mathcal{Q})) \wedge \mathcal{Q}' \text{ dominates } \mathcal{Q} \end{array}$$

In Section 5.5, we give quorum constructions that are optimal with respect to this metric without the tiebreaker rule. We do not discuss how to construct non-dominated coteries from dominated ones; possible ways to do so are discussed in [BI95] and [GMB85].

## 5.4 Failure models

In this section, we present two failure models that we use to derive quorum constructions: one that enables survivor sets to be used directly as quorums, and one that requires survivor sets to be discarded to form a quorum system. Both models are specific to multi-site systems, and although they both model site failures, they model different system properties as we discuss in Section 5.7.

### 5.4.1 The multi-site hierarchical model

The first model, which we call the *multi-site hierarchical model*, decouples site failures from process failures. The failure model has two components: $\mathcal{F}_s$, which characterizes the failures of sites, and $\mathcal{F}_p$, which characterizes failures within a site. More specifically, $\mathcal{F}_s$ is a set of maximal subsets of sites that can fail simultaneously, $|\mathcal{F}_s| > 0$. $\mathcal{F}_p$ is an array with one entry for each site, where $\mathcal{F}_p[i]$ is the set of maximal subsets of processes that can be simultaneously faulty in site $B_i$ when $B_i$ is not faulty, $|\mathcal{F}_p[i]| > 0$, $i \in \{1, \ldots, |\mathcal{B}|\}$. Given an instance of this model, a set $S_i \subseteq P$ is in $\mathcal{S}_\Pi$ if and only if:

$$\begin{aligned} \exists FS \in \mathcal{F}_s : \; & \bigwedge \forall B_j \in \mathcal{B} \setminus FS : \exists FP \in \mathcal{F}_p[j] : B_j \cap S_i = B_j \setminus FP \\ & \bigwedge \forall B_j \in FS : S_i \cap B_j = \emptyset \end{aligned}$$

The multi-site threshold model proposed in [JM05b] is a threshold-based version of this model: $f_s$ is the maximum number of sites that fail simultaneously, and $F_p[i]$ is the maximum number of processes that fail simultaneously in site $B_i$.

### 5.4.2 The bimodal model

The *bimodal model* is similar to the multi-site hierarchical model: it also has two components $\mathcal{F}_s$ and $\mathcal{F}_p$. In general, this model represents settings in which there are multiple sites ($|\mathcal{B}| > 1$), all sites can fail but one, and if only one site is not faulty, then all processes in it are correct. Thus, each site is a survivor set. If multiple sites are non-faulty, then the non-faulty sites can have faulty processes. We describe these process failures with $\mathcal{F}_s$ and $\mathcal{F}_p$. Finally, we assume that there exists at least one site $B_i$ such that $B_i \notin FS$ for every $FS \in \mathcal{F}_s$. Although $B_i$ is not in any element of $\mathcal{F}_s$, it can still fail in the case that there is one non-faulty site $B_j$ with no faulty processes, and $j \neq i$. This assumption is necessary to derive an optimal construction, as we explain in Section 5.5.2.

The bimodal model contains the same failure patterns as the multi-site hierarchical model for the same components $\mathcal{F}_s$ and $\mathcal{F}_p$, but it contains $|\mathcal{B}|$ additional failure patterns, one for each site $B_i$. More specifically, a set $S_i \subseteq P$ is a survivor set for an instance of this model if and only if:

$$\begin{aligned}\bigvee\ & \exists FS \in \mathcal{F}_s : \bigwedge \forall B_j \in \mathcal{B} \setminus FS : \exists FP \in \mathcal{F}_p[j] : B_j \cap S_i = B_j \setminus FP \\ & \qquad\qquad \bigwedge \forall B_j \in FS : S_i \cap B_j = \emptyset \\ \bigvee\ & \exists B_j \in \mathcal{B} : S_i = B_j\end{aligned}$$

To construct a proper set of survivor sets, we need to impose the following constraint: $\forall FS \in \mathcal{F}_s : (|\mathcal{B} \setminus FS| > 1) \wedge (\forall B_i \in FS : \mathcal{F}_p[i] \neq \{\emptyset\})$. Without this constraint, the set of survivor sets might not be minimal, violating minimality.

The bimodal model does not have the intuitive appeal of the multi-site hierarchical model. Nonetheless, we argue in Section 5.7 that for at least two-site systems, it is practical. In addition, it has theoretical interest, which we describe in Section 5.5.

## 5.5 Quorum constructions

In this section, we use the failure models described to derive quorum constructions that are optimal with respect to the metric $\mathcal{A}()$. The first construction covers all survivor sets in $\mathcal{S}_\Pi$ by using $\mathcal{S}_\Pi$ itself. We provide a necessary and sufficient condition for this to hold. The other construction is for systems in which it is not possible to cover all survivor sets. Such a construction is important when survivor sets do not pairwise intersect. This construction is also optimal with respect to the metric $\mathcal{A}()$ except that the resulting coterie may be dominated.

### 5.5.1 Achieving optimal availability

Let $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ be a system profile, and suppose that we use the multi-site hierarchical model to determine $\mathcal{S}_\Pi$. To cover all survivor sets in $\mathcal{S}_\Pi$, it is necessary and sufficient that $\mathcal{F}_s$ and $\mathcal{F}_p$ satisfy the following property:

$$\begin{aligned}\forall FS, FS' \in \mathcal{F}_s :\ \exists B_i \in \mathcal{B} :\ & \bigwedge B_i \notin FS \\ & \bigwedge B_i \notin FS' \\ & \bigwedge\ \forall FP, FP' \in \mathcal{F}_p[i] :\ \exists p \in B_i : p \notin FP \wedge p \notin FP'\end{aligned}$$

In words, we require that there is at least one site shared between any two survivor sets, and within that site there is at least one process that is shared between the two survivor sets. To show that this property is necessary, suppose that this property is violated. That is, there are $\mathit{FS}, \mathit{FS}'$ in $\mathcal{F}_s$ such that, for every $B_i \in \mathcal{B}$, at least one of the following holds: 1) $B_i \in \mathit{FS}$; 2) $B_i \in \mathit{FS}'$; 3) there are $\mathit{FP}, \mathit{FP}' \in \mathcal{F}_p[i]$ such that for every $p \in B_i$, either $p \in \mathit{FP}$ or $p \in \mathit{FP}'$. This implies that there are at least two disjoint survivor sets $S$ and $S'$ in $\mathcal{S}_\Pi$. Now suppose by way of contradiction that there is a coterie $\mathcal{Q}$ that covers all survivor sets in $\mathcal{S}_\Pi$, *i.e.*, $\mathcal{A}(Q) = |\mathcal{S}_\Pi|$. We then have that there is a quorum $Q \in \mathcal{Q}$ such that $Q \subseteq S$. Similarly, there is a quorum $Q' \in \mathcal{Q}$ such that $Q' \subseteq S'$. Thus, if $S \cap S' = \emptyset$, then $Q \cap Q' = \emptyset$. We conclude that $\mathcal{Q}$ cannot be a coterie because it violates the 2–Intersection property.

To see that the property is sufficient is straightforward: by the definition of survivor sets, no survivor set is strictly contained in another, and the intersection property is guaranteed by assumption.

If we use $\mathcal{S}_\Pi$ as a coterie, then we have achieved the best possible value for our availability metric because it covers all the survivor sets (i.e., $\mathcal{A}(\mathcal{S}_\Pi)$ has the maximum value of $|\mathcal{S}_\Pi|$). Using all the sites in the system, however, may be unnecessary. For example, if the system satisfies $k$–Intersection for some $k > 2$, then we may be able to construct a coterie over fewer sites.[2] We illustrate this point with a threshold version of the multi-site hierarchical model. Suppose that every set $\mathit{FS} \in \mathcal{F}_s$ has the same size $f_s \geq 0$, and that for every $B_i \in \mathcal{B}$ and every $\mathit{FP} \in \mathcal{F}_p[i]$, we have that $|\mathit{FP}| = t$ for some nonnegative integer $t$. Then, if $|\mathcal{B}| \geq 2f_s + 1$, we only need to select a subset $\mathcal{B}' \subseteq \mathcal{B}$ of $2f_s + 1$ sites. For each site $B_i \in \mathcal{B}'$, we select $2t + 1$ processes from $B_i$. A quorum is obtained by selecting a majority of processes from a majority of sites in $\mathcal{B}'$.

We call this construction *Qsite*. As an example, suppose that $|\mathcal{B}| = 4$, $f_s = 1$, and for each site $B_i$, we have that $|B_i| = 4$ and $t = 1$. We then use 3 sites, as $2f_s+1 = 3$, and 3 processes from each site, as $2t + 1 = 3$. From the construction, a quorum in $\mathcal{Q}$ is hence composed of four processes, two from a site $B_i$ and two from a site $B_j$, $i \neq j$.

[2] $k$–Intersection generalizes 2–Intersection, and states that all subsets of $k$ quorums intersect.

**Table 5.1:** Quorum sizes

| | $\mathbf{t = 1}$ | | $\mathbf{t = 2}$ | |
|---|---|---|---|---|
| $\mathbf{f_s}$ | **Majority** | **Qsite** | **Majority** | **Qsite** |
| 1 | 5 | 4 | 8 | 6 |
| 2 | 8 | 6 | 13 | 9 |
| 3 | 11 | 8 | 18 | 12 |
| 4 | 14 | 10 | 23 | 15 |

This system has nine processes, and so a majority would consist of five processes. For both majority and Qsite, the coterie is available as long as there are $f_s + 1 = 2$ non-faulty sites. Majority, however, not only requires that two sites are non-faulty, but also that at least one of the sites contains no faulty processes. A coterie generated by Qsite does not have this same constraint, and it is available as long as there are two non-faulty sites, each non-faulty site containing two non-faulty processes. This happens because majority uses larger quorums, and it tolerates fewer process failures.

It is not hard to see that Qsite requires fewer processes compared to majority coteries, and that the difference increases with the value of $f_s$. Table 5.1 shows quorum sizes for different values of $f_s$ and $t$. The main observation is that Qsite requires fewer processors in all the cases, and the difference between the two constructions increases with the value of $f_s$. Using fewer processors in each quorum reduces the load handled by any particular processor. Assuming that quorums are uniformly chosen by clients, having smaller quorums implies that processors have to handle fewer requests. Load is inversely proportional to the capacity, and by reducing load we are actually increasing the total capacity of the system, where the capacity is the number of requests the system can handle per unit of time [NW98].

### 5.5.2 The bimodal construction

It may be the case that the set of survivor sets do not satisfy 2–Intersection, and so can not be used as a coterie. For example, in the bimodal model, for each site $B_i$,

$B_i$ is a survivor set, and since sites are disjoint, $\mathcal{S}_\Pi$ is not a coterie.

One can construct a coterie from any $\mathcal{S}_\Pi$, though, by simply discarding survivor sets until remaining sets satisfy 2-Intersection. This procedure clearly will terminate with a coterie since a single set is a coterie of one quorum. To obtain a coterie that is optimal with respect to $\mathcal{A}()$, we need to determine the minimal set $\mathcal{S} \subset \mathcal{S}_\Pi$ such that $\mathcal{S}_\Pi \setminus \mathcal{S}$ is a coterie. The problem of computing the minimum number of survivor sets that have to be removed from $\mathcal{S}_\Pi$ to obtain a coterie, however, is in general NP-Complete. We present a proof in the next section.

Under the bimodal model, it is simple to determine which survivor sets to discard. Consider the following intersection property that we call $k$-bimodal Intersection, $k > 1$:

$$\forall \text{ distinct } S_1, S_2, \ldots, S_{k+2} \in \mathcal{S}_\Pi : \quad \begin{array}{l} \bigvee \exists i, j \in [1, k] : S_i \cap S_j \neq \emptyset \\ \bigvee S_{k+1} \cap S_{k+2} \neq \emptyset \end{array}$$

Assume $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ follows the bimodal failure model. According to the model, it contains $|\mathcal{B}|$ survivor sets that are disjoint, one for each site $B_i \in \mathcal{B}$. Also by the failure model, there is a site $B_i$ such that $B_i \notin$ *FS*, for every *FS* $\in \mathcal{F}_s$. Let $S_i$ be the survivor set consisting of the processes of $B_i$. If $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ also satisfies $k$-bimodal Intersection, $k = |\mathcal{B}|$, then we know that any two survivor sets $S_a, S_b$ in $\mathcal{S}_\Pi \setminus \{S_1, S_2, \ldots S_k\}$ intersect, and that $S_i \cap S_a \neq \emptyset$ and $S_i \cap S_b \neq \emptyset$. Since this is true for any $S_a$ and $S_b$, the set $\mathcal{Q}_\ell = \{S_i\} \cup (\mathcal{S}_\Pi \setminus \{S_1, S_2, \ldots S_k\})$ is a coterie, and $\mathcal{A}(\mathcal{Q}_\ell) = |\mathcal{S}_\Pi| - (k-1)$. This is clearly optimal, since all of the remaining $k-1$ survivor sets do not intersect $S_i$. Also, if $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ does not satisfy $k$-bimodal Intersection, then there is no coterie that covers $|\mathcal{S}_\Pi| - (k-1)$ survivor sets, as there is no subset of $\mathcal{S}_\Pi$ of size $|\mathcal{S}_\Pi| - (k-1)$ that pairwise intersect. We call this construction *Bsite*.

## 5.6 NP-Completeness of the DSS problem

In this section, we show that the problem of determining the minimum number of survivor sets that has to be removed from a set $S_P$ such that the remaining survivor sets form a coterie is NP-Complete. Our strategy consists in defining a decision problem, and showing that this problem is NP-Hard with a reduction from the Vertex-Cover problem. We then argue that an algorithm for the decision problem can be used to solve the search problem, which consists in determining the actual set of survivor sets to remove. Finally, we argue that we can use a search solution to solve the optimization problem. The optimization problem consists in determining a minimal set of survivor sets to remove.

We now define the DSS (Disjoint Survivor Sets) decision problem. Instead of assuming only a set of survivor sets $\mathcal{S}_\Pi$ as the input of the problem, we also assume pairs of survivor sets in $\mathcal{S}_\Pi$ that are disjoint. To compute such pairs from the set of survivor sets $\mathcal{S}_\Pi$ in a system profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ can be done in polynomial time by simply checking the intersection of every pair of survivor sets. A simple solution for this algorithm has time complexity $O(|P| \cdot |\mathcal{S}_\Pi|)$. This simple algorithm consists in having a matrix $M$ with $|P|$ rows and $|\mathcal{S}_\Pi|$ columns. For each process $p$, we set $M_{p,s}$ if the survivor set associated with column $s$ contains process $p$. Another pass on the matrix provides the information we are looking for. This algorithm is perhaps not the best one to use. A further discussion of algorithms to optimally find such disjoint sets is out of the scope of this work.

We then state the DSS decision problem as follows:

**Instance** : A set $\mathcal{S}_\Pi$ of survivor sets, a set $\mathcal{P}$ of pairs of survivor sets in $\mathcal{S}_\Pi$, and a positive integer $r \leq |\mathcal{S}_\Pi|$.

**Question** : Is there a subset $\mathcal{S} \subseteq \mathcal{S}_\Pi$ such that:

1. For every $(S_1, S_2) \in \mathcal{P}$: $S_1 \in \mathcal{S} \vee S_2 \in \mathcal{S}$;
2. $|\mathcal{S}| \leq r$.

Consider now the definition of the VC (Vertex Cover) problem, taken from [GJ79]:

**Instance** : A graph $G = (V, E)$ and a positive integer $k \leq |V|$.

**Question** : Is there a vertex cover of size $k$ or less for $G$, that is, a subset $V' \subseteq V$ such that $|V'| \leq k$, and for each edge $\{u, v\} \in E$, at least one of $u$ and $v$ belongs to $V'$?

The following two claims show that DSS is NP-complete.

**Claim 5.6.1** VC $\leq_m$ DSS

**Proof:**

To show this claim, we have to provide a polynomial-time algorithm that outputs an instance $\langle \mathcal{S}_\Pi, \mathcal{P}, r\rangle$ of the DSS problem given an instance $\langle G, k\rangle$ of the VC problem such that:

$$\langle G, k\rangle \in \text{VC} \rightarrow \langle \mathcal{S}_\Pi, \mathcal{P}, r\rangle \in \text{DSS} \tag{5.1}$$

$$\langle \mathcal{S}_\Pi, \mathcal{P}, r\rangle \in \text{DSS} \rightarrow \langle G, k\rangle \in \text{VC} \tag{5.2}$$

Consider the following algorithm:

```
**Algorithm** VCtoDSS on input $\langle G = (V, E), k\rangle$
$\mathcal{S}_\Pi \leftarrow \emptyset$
$\mathcal{P} \leftarrow \emptyset$
For every $v \in V$:
    $\mathcal{S}_\Pi \leftarrow \mathcal{S}_\Pi \cup \{S_v\}$
For every edge $\{u, v\} \in E$:
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{(S_u, S_v)\}$
$r \leftarrow k$
output $\langle \mathcal{S}_\Pi, \mathcal{P}, r\rangle$
```

The algorithm clearly runs in polynomial time. We then need to show implications 5.1 and 5.2. First we show 5.1. Suppose that $\langle G, k\rangle \in$ VC. We then have that there is a vertex cover $V'$ of size at most $k$ for $G$. That is, there is a subset $V'$ of $V$ of size at most $k$ such that for every edge $\{u, v\}$, either $u$ or $v$ is in $V'$. By the construction of the algorithm, if $\mathcal{S}$ is the set of all $S_v$ such that $v \in V'$, then $\mathcal{S}$ must be such that for all $(S_v, S_u) \in \mathcal{P}$, either $S_v \in \mathcal{S}$ or $S_u \in \mathcal{S}$. We conclude that $\langle \mathcal{S}_\Pi, \mathcal{P}, r\rangle$ is in DSS.

It remains to show implication 5.2. Suppose that $\langle \mathcal{S}_\Pi, \mathcal{P}, r\rangle$ is in DSS. This means that there is a subset $\mathcal{S}$ of $\mathcal{S}_\Pi$ such that $|\mathcal{S}| \leq r$ and for every $(S_v, S_u) \in \mathcal{P}$, either $S_v \in \mathcal{S}$ or $S_u \in \mathcal{S}$. We then have that $V' = \{v : S_v \in \mathcal{S}\}$ must be a vertex cover for $G$, and $|V'| \leq k$, by the construction of the algorithm. This concludes the proof of our claim.

□

With a simple modification of the algorithm presented in the previous proof, we can also show that DSS map-reduces to VC. This is important because there are efficient approximation algorithms for the vertex cover problem, such as the ones in [Hal02], that can be used to compute a solution for DSS.

**Claim 5.6.2** DSS is in NP

**Proof:**

We need to provide a polynomial-time verifier that takes an instance $\langle \mathcal{S}_\Pi, \mathcal{P}, r\rangle$ of the DSS problem and a certificate $C$. The verifier then outputs whether $\langle \mathcal{S}_\Pi, \mathcal{P}, r\rangle \in$ DSS or not. The certificate consists of a subset of $\mathcal{S}_\Pi$. We have that the verifier works as follows:

```
Verifier on input $\langle \mathcal{S}_\Pi, \mathcal{P}, r\rangle, C$
  Parse $C$ into set $\mathcal{S}$
  Verify if $|\mathcal{S}| \leq r$
  Verify if $\mathcal{S} \subseteq \mathcal{S}_\Pi$
  Verify if for every $(S_i, S_j) \in \mathcal{P}$,
          either $S_i \in \mathcal{S}$ or $S_j \in \mathcal{S}$
```

If any verification fails return false, else return true

The verifier clearly runs in polynomial time on the size of the input.

□

A well-known result from complexity theory says that search reduces to decision for NP-complete problems [BG94]. For DSS, the search problem consists of finding a subset $\mathcal{S}$ of $\mathcal{S}_\Pi$ such that $|\mathcal{S}| \leq r$ and for every $(S_i, S_j) \in \mathcal{P}$, either $S_i \in \mathcal{S}$ or $S_j \in \mathcal{S}$. The algorithm returns $\perp$ if no such set exists. Thus, we can only have a polynomial-time algorithm for the search problem if there is a polynomial-time algorithm for the decision problem. Such an algorithm only exists if P = NP. To find the smallest set $\mathcal{S}$ given a system profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ we can run a search algorithm for smaller values of $r$, and return the set $\mathcal{S}$ returned by the smallest value of $r$ for which the search algorithm does not return $\perp$. We then also have that there is a polynomial-time algorithm for the optimization problem if and only if P = NP.

## 5.7 Failure models in practice

The failure models presented in Section 5.4 are abstract views of failures in a multi-site system. In this section, we present probabilistic models that we use to extract the parameters of our failure models. First, we use data from a real system to argue why we believe site failures are common in multi-site systems. In the remainder of the section, we discuss process failures for the two models we propose in this chapter. For each model, we discuss a framework based on a Markov chain and illustrate it with an example.

### 5.7.1 Site failures

To understand how sites fail in a multi-site system, we studied the failure data of a particular system, the BIRN Grid [bir06]. We obtained monthly availability data for

15 BIRN sites from January 2004 through August 2004.[3] The monthly availability of a site is given by:

$$\text{Availability} = \frac{\text{Total hours} - \text{Unplanned outages}}{\text{Total hours}} \times 100$$

where "Total hours" is the total number of hours in a month deducted the scheduled down time, and "Unplanned outages" is the total number of hours that the site was not available not considering scheduled down time.

According to this availability data, a site becoming unavailable is a surprisingly common event. On average, each site did not have 100% availability on 5 out of the 8 months, which implies that in a given month several sites have unplanned outages and become unavailable as a consequence.

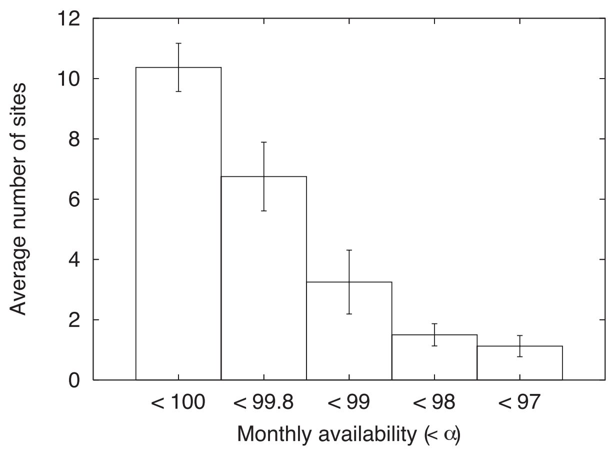

**Figure 5.1:** Site availability: Number of sites with availability below $\alpha$. The error bars correspond to the standard error.

Figure 5.1 summarizes the availability of sites. For each month, we counted the number of sites that had availability below some value $\alpha$, for different values of $\alpha$. We then computed the average across the eight months for each value. This average is what we plot in Figure 5.1. From the figure, on average over 10 sites do not have 100% availability in a month.

[3]This data is consistently collected by the BIRN staff, and made available through their web page. To determine availability of a site, they use pings and notifications from the SRB (Storage Resource Broker) service.

**Table 5.2:** Average worst-case unavailability across eight months with standard error in parentheses

| **Number of sites** | **Unavailability in minutes** |
|---|---|
| 1 | 3288 (979) |
| 2 | 87 (33) |
| 3 | 1.9 (1.0) |
| 4 | 0.017 (0.009) |

According to our definition of availability, if the availability of a site is $99\%$, then it was down for approximately 7 hours in that month, and hence every $1\%$ of unavailability corresponds to 7 hours of unplanned down time. From Figure 5.1, there is on average at least one site with availability less than $97\%$, which implies that such a site is unavailable for over 21 hours. In fact, we observed availability values as low as $79\%$.

Because we aim at constructing coteries that leverage the existence of multiple sites, we computed the average worst-case down time. Table 5.2 shows the worst-case unavailability for different numbers of sites averaged across the eight months. In more detail, for each month we determine the $x$ sites, $x \in \{1, 2, 3, 4\}$, with lowest availability, and then compute how many minutes during that month we expect the $x$ sites to be simultaneously unavailable. To compute such numbers, we assume that the events causing the sites to be unavailable happen independently. For each value of $x$, we then average the number of minutes obtained across the eight months. We observe that the average worst case varies from over 55 hours for a single site to a fraction of a minute for four sites.

Back to the model of Section 5.4, one can determine the value of $f_s$ by looking at the values of Table 5.2 and choosing the one that corresponds to the requirements of the application. For example, if an application sets $f_s$ to zero and uses a single site, then it will experience, considering the worst case, 55 hours of unavailability in a month on average.

In trying to determine the causes of low monthly availabilities in multi-site systems, we identified a few causes for a site to be unavailable, observed in BIRN sites, in TeraGrid sites, and in a local computer cluster. These causes are:

1. Software incompatibility/misconfiguration;
2. Power failures;
3. Failure of shared resources (*e.g.* storage);
4. Broken pipes causing floodings;
5. Local campus network problems;
6. Loss of air-conditioning.

Note that the order in this list is arbitrary. We are currently attempting to further quantify these failures.

### 5.7.2 Obtaining the multi-site hierarchical model

The multi-site hierarchical model has two components: $\mathcal{F}_s$ that describes site failures, and $\mathcal{F}_p$ that describes the failures of processes within a site. We can determine $\mathcal{F}_s$ using, for example, data such as described in Section 5.7.1. To determine $\mathcal{F}_p$, we need a model of failures within a site. Even when sites are not faulty, individual processes can fail due to, for example, hardware faults. In many multi-site systems, hardware and software platforms are the same across the computing nodes (where processes run) of a site because of the difficulty in managing a heterogeneous environment. We hence assume that the reliability of processes within a site is uniform and independent. Of course, this assumption may be violated by viruses and worms. We discuss techniques to cope with such threats in Chapter 6.

We can model failures in sites using a *Markov chain* [Ros00]. Instead of modeling the whole system, we have chosen to model sites individually. We assume that sites operate independently, and that outside of expected message communication the operation of a process at a site has little or no influence on the operation of a process at another site. As a consequence, sites change their failure states concurrently.

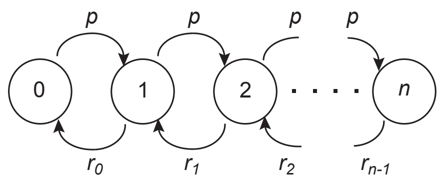

**Figure 5.2:** Model for a single site with $n$ process

As process failures are independent, states of the model correspond to the number of faulty processes in a site, and the probability of undergoing a transition from a state with $f$ faulty processes to a state with $f+1$ process is $p$. Repair transitions (from $f+1$ to $f$), however, may have probabilities that change with the value of $f$. For example, resources to repair processes can be progressively allocated as more processes fail. As a result, the repair probability remains constant or even increases with the value of $f$.

Figure 5.2 depicts the chain we just described. Assuming that no transition probability is zero, we have that this chain is irreducible and ergodic. According to the model, processes fail independently, but the probability of repair (undergoing a transition from state $f+1$ to $f$) may change with the value of $f$. In our model, we use $r_f$ to denote the probability that the site undergoes a transition from state $f+1$ to state $f$.

Repairs in different sites happen independently, and hence the probability of a repair transition does not increase with failures in different sites. That is, if a process fails in site $B_i$ and another in site $B_j$, $i \neq j$, they do not mutually affect their repair probabilities.

Using this model, we can easily compute a threshold on the number of failures for each site. First, we need to determine a target degree of reliability $\rho$, which is the probability that the number of simultaneous process failures in any site is higher than expected. Because our model is an irreducible ergodic Markov chain, we can compute the limiting probabilities of all states: the probability of being at a state $j$ after a long time has elapsed, independent of the initial state $i$ ($\pi_j = \lim_{n\to\infty} P^n_{ij}$) [Ros00]. Using these limiting probabilities, we can determine a threshold for each site: the threshold for a site $S_i$ is the number of failures associated to the first state that has a limiting probability smaller than $\rho$. This implies that any state with failures above the threshold has probability lower than $\rho$.

To illustrate the process of obtaining a threshold for a site, we give an example. Let $\mathcal{B}$ be a collection of sites such that each site has three processes. Suppose that the probabilities of failure and repair are the same across all the sites. These probabilities are as follows: $p = 0.01$, $r_0 = 0.3$, $r_1 = 0.4$, and $r_2 = 0.5$. Computing the limiting probabilities, we have the following: $\pi_0 = 0.96695, \pi_1 = 0.03223, \pi_2 = 0.00080, \pi_3 = 0.00002$. If $\rho$ is $0.001$, for instance, we have that the threshold is one for every site, and $\mathcal{F}_p$ is as follows: $\mathcal{F}_p[i] = \{a_i : a_i \in B_i\}, \forall B_i \in \mathcal{B}$.

Note that the reverse order is also possible: choose a value for $t$ (the threshold on the number of process failures in a site) and compute the corresponding probability of violating this threshold. Using one method or the other depends on design constraints.

### 5.7.3 Obtaining the bimodal model

From the description of the bimodal model in Section 5.4, when $k - 1$ sites fail, the remaining sites have no faulty processes. This means that the processes of each site comprise a survivor set. At the same time, it is possible that all available sites have faulty processes. We model this with a framework based on a Markov chain. Due to the complexity of this model, our framework is only meant to give a more practical view rather than serve as a general framework.

As in the previous section, the basic idea consists in determining probabilities for the possible states of the system, and to use a degree of reliability (a value $\rho \in [0, 1]$) to determine the states that we consider as normal states. Compared to the chain from the previous section, a state corresponds to failures across all the sites. We then label states with counters, one for each site. That is, we have one state for each possible value of the string $f_1 \cdot f_2 \cdot \ldots \cdot f_k$, where $0 \leq f_i \leq |B_i|$ and $B_i \in \mathcal{B}$. Using a directed graph as a way of visualizing the model, we have that the states are represented by nodes, and the transitions by edges, where each edge has a weight that is the transition probability.

In this model, we have three types of edges: site-failure edges, process-failure edges, and repair edges. A site-failure edge corresponds to the transition from a state in which a given site has one or more available processes to one in which all processes in this site are faulty. Using probability notation, let $X_s$ be the random variable representing the state at step $s$. We then have that:

$$\begin{aligned} Pr\{X_{s+1} = f_1 \cdots |B_i| \cdot f_{i+1} \cdots f_k | X_s = f_1 \cdots f_i \cdot f_{i+1} \cdots f_k\} &= p_s, f_i < |B_i| - 1 \\ Pr\{X_{s+1} = f_1 \cdots |B_i| \cdot f_{i+1} \cdots f_k | X_s = f_1 \cdots |B_i| - 1 \cdot f_{i+1} \cdots f_k\} &= p_f + p_s, f_i \\ &= |B_i| - 1 \end{aligned}$$

where $p_f$ is the probability of a process failure, and $p_s$ is the probability of a site failure. As a simplifying assumption, we have that $p_s$ and $p_f$ are constant across the sites. A process-failure edge is a transition from a state in which some site $B_i$ has $f_i$ faulty processes to a state in which $B_i$ has $f_i + 1$ faulty processes, $f_i < |B_i|$. Using probability notation, we have:

$$Pr\{\ X_{s+1} = f_1 \cdots f_i + 1 \cdots f_k | X_s = f_1 \cdots f_i \cdots f_k\} = p_f, f_i < |B_i|$$

Finally, we call a repair edge a transition from a state in which $f_i + 1$ processes of some site $B_i$ are faulty to a state in which $f_i$ processes of $B_i$ are faulty. That is:

$$Pr\{\ X_{s+1} = f_1 \cdots f_i \cdots f_k | X_s = f_1 \cdots f_i + 1 \cdots f_k\} = p_r(f_1 \cdots f_i + 1 \cdots f_k), f_i \geq 0$$

where $p_r()$ is a repair probability mapping. Different from $p_s$ and $p_f$, we assume that the repair probability may differ for different states. In fact, this control over repair probabilities is what we use to guarantee that the properties of the bimodal model hold. An additional assumption that completes the model is that all other possible transitions have zero probability.

Figure 5.3 illustrates a model for two identical sites $B_1$ and $B_2$ of $n$ processes each. In the figure, we mark the undesirable states by including them in a gray region.

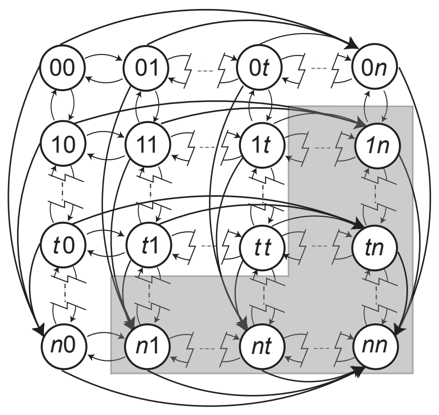

**Figure 5.3:** Model for two sites

These states are the ones that violate the Bsite construction, and therefore must have low probability. To determine the probability of a state, we use also limiting probabilities.

In this model, we assume that probabilities of failure are constant, and they cannot be changed as the system changes states. We assume, however, that we are able to have different repair probabilities for different states. As a physical explanation, repair probabilities change as the effort spent to repair the system changes. Thus, we can increase the repair probability for an undesirable state, thereby decreasing the probability of being in this state. In practice, this means that the amount of physical resources used to repair processes must increase with the number of failures in the system. It is then necessary to be able to detect failures. A failure detector for this application, however, can be unreliable as the only side-effect is to have more resources used to repair processes unnecessarily. Having an unreliable failure detector implies that the repair probabilities have to take into account false positives. We therefore assume that it is possible to bound and estimate the frequency with which the failure detector makes mistakes.

As an example, suppose that $n = 3$, $p_s = 0.004$, $p_f = 0.001$, and $p_r(f_1 \cdot f_2) = 0.1$, if $f_1 \cdot f_4$ is outside the gray region, and $p_r(f_1 \cdot f_2) = 0.4$, if $f_1 \cdot f_2$ is inside the gray region. We have chosen these values using the following guidelines. First, as we observed in Section 5.7.1, site failures are common. We then assume that the probability of a site failure is higher than that of a process failure, although we kept them in the same order of magnitude. Second, we assume that repair probabilities are much higher than the failure probabilities.

One still needs to choose a value for $t$, as repair probabilities depend upon this value. For such a small system, this choice is constrained to be either $t = 0$ or $t = 1$. If $t$ is greater than 1, then the 2-bimodal intersection property cannot be satisfied, and we are not able to construct a coterie using the technique proposed in Section 5.5. We therefore assume that $t = 1$, and we have the following limiting probabilities:

$$M = \begin{bmatrix} 0.7815 & 0.0391 & 0.0332 & 0.0344 \\ 0.0391 & 0.0020 & 0.0004 & 0.0004 \\ 0.0332 & 0.0004 & 0.0003 & 0.0004 \\ 0.0344 & 0.0004 & 0.0004 & 0.0004 \end{bmatrix}$$

where $M[f_1+1, f_2+1]$ is the limiting probability of state $f_1 \cdot f_2$. More specifically, we have that $\pi_{f_1 \cdot f_2} = M[f_1+1, f_2+1]$.

Suppose now that $a_1, a_2, a_3$ are the processes in one site, $b_1, b_2, b_3$ are the processes in the other site, and $4 \times 10^{-4} < \rho < 2 \times 10^{-3}$. We have that: 1) $\mathcal{F}_s = \{\emptyset\}$; 2) $\mathcal{F}_p[1] = \{a_i : a_i \in B_1\}$; 3) $\mathcal{F}_p[2] = \{b_i : b_i \in B_2\}$. The set of survivor sets is as follows:

$$\mathcal{S}_\Pi \quad = \quad \{a_1a_2a_3, b_1b_2b_3\} \cup \{a_ia_jb_lb_m : i,j,l,m \in \{1,2,3\} \wedge i \neq j \wedge l \neq m\}$$

and from the Bsite construction, we have, for example, the following coterie:

$$\mathcal{Q} \quad = \quad \{a_1a_2a_3\} \cup \{a_ia_jb_lb_m : i,j,l,m \in \{1,2,3\} \wedge \ i \neq j \wedge l \neq m\}$$

which is dominated by:

$$\mathcal{Q}' = \{a_1a_2a_3\} \cup \{a_ib_jb_l : i,j,l \in \{1,2,3\} \wedge \ j \neq l\}$$

and we have by definition that $\mathcal{Q} \prec_a \mathcal{Q}'$. From the matrix $M$, observe that $\mathcal{Q}'$ is unavailable only in state 30, considering only allowed states (states that have probability greater than the degree of reliability). This is optimal as there is no coterie that is available for both states 30 and 03.

Although the system of the example is a simple one, it illustrates well that the bimodal model is implementable. We believe that the results can be generalized for two-site systems with more processes, but it is an open question whether there is a practical implementation for systems with more than two sites.

## 5.8 Evaluating coteries on PlanetLab

To evaluate the different choices for quorums in a multi-site system, we conducted an experiment on PlanetLab using an implementation of the Paxos algorithm [Lam98]. In brief, Paxos assumes that processes have one or more of the three following roles: Proposer, Acceptor, and Learner. Proposers propose ballots that are accepted by Acceptors. To propose, a Proposer has to read from and write to a quorum of Acceptors. Once an Acceptor accepts a ballot, it notifies the set of Learners. A Learner decides upon a value once it receives notifications from a quorum of Acceptors.

In our experiment, we have three settings. In all settings, one single host (a UCSD host) has the roles of both a Proposer and a Learner, whereas the Acceptors are PlanetLab hosts spread across three sites (UC Davis, UT Austin, Duke). The settings are:

**3Sites:** One host from each site. A quorum consists of any set of two hosts. This is the Qsite construction, for $f_s = 1$, and $t = 0$;

**3SitesMaj:** Three hosts from each site. A quorum consists of majorities of hosts from two sites, and it has size four. This is the Qsite construction for $f_s = 1$, and $t = 1$;

**SimpleMaj:** Three hosts from each site. A quorum consists of any simple majority of sites. That is, any subset of five Acceptors.

For each setting, we have the Proposer issuing a new ballot every 15 minutes, and we log the time it takes to decide upon a value for this ballot. To implement a reliable channel, we create a new thread for every message sent, and this thread tries to send the message through a TCP connection until it succeeds. As a consequence, we have that every message sent by one process to another is eventually received, as long as the receiving process eventually recovers if it fails.

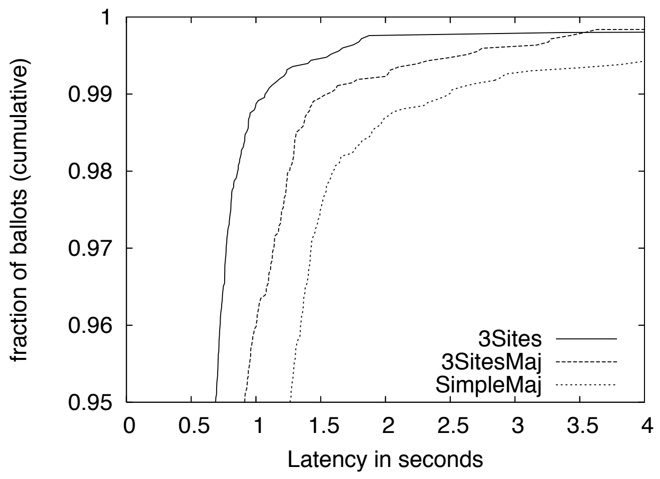

**Figure 5.4:** Cumulative latency distribution

To register failures, every time establishing a TCP connection to another host times out, we log it to a file. A failure in this case is the inability to reach the Acceptor, not necessarily implying that the host has crashed. That is, the unavailability of a host may be caused by a network partition.

We had these three settings running in parallel for 27 days in April 2005. Figure 5.4 shows part of the cumulative distribution function for the latency of reaching agreement on each ballot. We additionally show in Table 5.3 the fraction of samples with value greater than $4s$, and in Figures 5.5, 5.6, and 5.7 the time series for each of the constructions.

It is not surprising that 3Sites has the best response time for the average case, followed by 3SitesMaj and SimpleMaj, since quorums have fewer Acceptors in this exact order. However, the graph shows that there is a point (around $3.5s$) in which the curve for 3SitesMaj crosses 3Sites. This implies that there are fewer samples for 3SitesMaj with latency greater than $3.5s$ than for 3Sites. As the tail of the distribution for 3SitesMaj contains fewer samples, it has best availability among the three in this experiment.

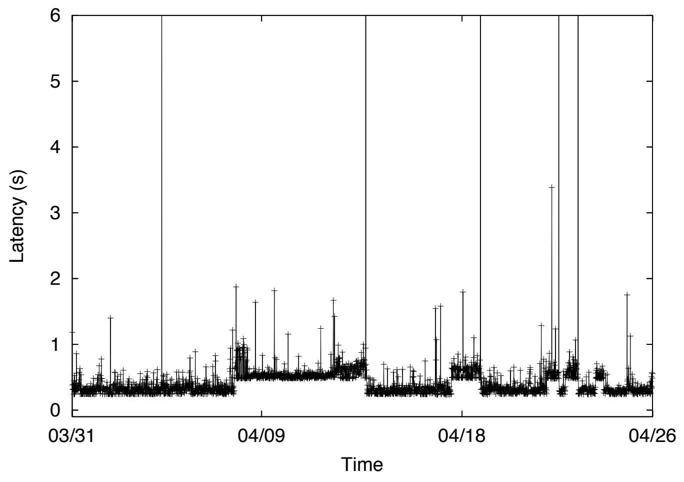

**Figure 5.5:** 3Sites latency to learn - time series

**Table 5.3:** Samples with value greater than $4$ seconds.

| | **Latency** $> 4s$ |
|---|---|
| **3Sites** | 0.0020 |
| **3SitesMaj** | 0.0016 |
| **SimpleMaj** | 0.0057 |

To understand why this is the case, we need to understand what components are involved in deciding upon a ballot. The latency of a ballot has two main components: message latency and process failures. From the graph, the message latency component dominates until $3.5s$. After $3.5s$, the delay is mostly caused by the inability to reach enough Acceptors. Having more processes increases the latency for 3SitesMaj and SimpleMaj compared to 3Sites for values under $3.5s$, where the message latency component has more weight. On the other hand, 3SitesMaj presents better response time for values greater than $3.5s$, when there are process failures. Thus, there is a tension between obtaining good response time on average and having a larger fraction of the samples within a bounded response time. This information is important, for example, when de-

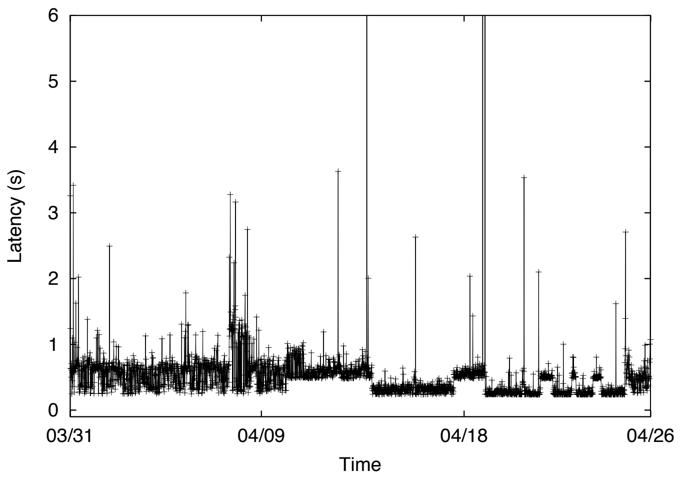

**Figure 5.6:** 3SitesMaj latency to learn - time series

termining the time-out for a quorum-based service. Considering our three settings, if a time-out value greater than $3.5s$ is chosen, then 3SitesMaj is likely to time out less often than 3Sites and SimpleMaj. Finally, an interesting observation is that SimpleMaj not only had the worst average response time, but also had the largest fraction of samples with response time greater than $4s$. This indicates that using majority quorums is a poor choice for multi-site systems.

We also counted the number of ballots for which the Proposer could not initially contact enough Acceptors to obtain a quorum, and the decision on the ballot was therefore delayed until enough Acceptors were available. When this happens to a ballot, we say that this ballot is *postponed*. For each setting, we have the following:

**3Sites:** There were 3 postponed ballots;

**3SitesMaj:** There were 2 postponed ballots. Only for one of these ballots, there would be one quorum available in the simple majority scheme;

**SimpleMaj:** There were 4 postponed ballots. For all these ballots, using the majorities of two sites would give us an available quorum.

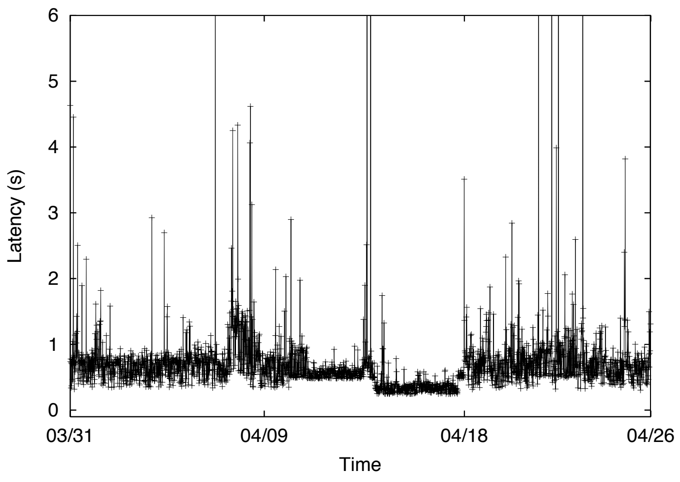

**Figure 5.7:** SimpleMaj latency to learn - time series

The data presented in this section is perhaps not conclusive because the number of failures observed was too small to be statistically valid. Moreover, PlanetLab is not a production system in the sense that sites are not designed to be highly available, and node repair is often leisurely. On the other hand, the results presented do not contradict any of our assumptions, thus indicating that our models may be suitable even for multi-site systems such as PlanetLab.

## 5.9 Conclusions

The constructions in this chapter constitute a first step into the practical construction of coteries for multi-site systems. We base one coterie construction on a failure model that we motivate from failure measurements from a deployed multi-site system and from a Markov model. We also consider a weaker failure model that has some theoretical and practical interest. We define optimality by introducing a metric that is suitable to dependent failures, and we show that our quorum constructions are optimal with respect to this metric.

Being a first step, this work leaves some questions unanswered. First, our multi-site hierarchical model is intuitive and is based on some failure data from a real system. How typical is this system? Is the model broadly applicable? Second, our bimodal model is based on the idea of having different repair probabilities for different states. This technique, which essentially integrates operating procedures with the failure model, appears to be a potentially powerful new direction for the design of novel and efficient protocols. Finally, we describe a method of building a coterie from survivor sets that do not satisfy 2–Intersection. The survivor sets are defined by some target availability, and the availability of the quorum system is reduced by discarding survivor sets. How does this strategy compare with one in which the initial target availability is increased until the survivor sets satisfy 2–Intersection?

# Chapter 6

# Surviving Internet catastrophes

This chapter also shows that our core/survivor set model has practical value by using the core abstraction to design a system that protects users' data against large-scale Internet attacks.

The Internet today is highly vulnerable to Internet epidemics: events in which a particularly virulent Internet pathogen, such as a worm or email virus, compromises a large number of hosts. Starting with the Code Red worm in 2001, which infected over 360,000 hosts in 14 hours [MSB02], such pathogens have become increasingly virulent in terms of speed, extent, and sophistication. Sapphire scanned most IP addresses in less than 10 minutes [MPS$^+$03], Nimda reportedly infected millions of hosts, and Witty exploited vulnerabilities in firewall software explicitly designed to defend hosts from such pathogens [MS04]. We call such epidemics *Internet catastrophes* because they result in extensive wide-spread damage costing billions of dollars [MSB02]. Such damage ranges from overwhelming networks with epidemic traffic [MPS$^+$03, MSB02], to providing zombies for spam relays [myd04] and denial of service attacks [sob], to deleting disk blocks [MS04]. Given the current ease with which such pathogens can be created and launched, further Internet catastrophes are inevitable in the near future.

Defending hosts and the systems that run on them is therefore a critical problem, and one that has received considerable attention recently. Approaches to defend against Internet pathogens generally fall into three categories. Prevention reduces the size of the vulnerable host population [Sym, WFBA00, WGSZ04]. Treatment reduces the rate of infection [crc01, SK03]. Finally, containment techniques block infectious communication and reduce the contact rate of a spreading pathogen [MSVS03, Wil02, Won04].

Such approaches can mitigate the impact of an Internet catastrophe, reducing the number of vulnerable and compromised hosts. However, they are unlikely to protect all vulnerable hosts or entirely prevent future epidemics and risk of catastrophes. For example, fast-scanning worms like Sapphire can quickly probe most hosts on the Internet, making it challenging for worm defenses to detect and react to them at Internet scale [MSVS03]. The Witty worm embodies a so-called *zero-day worm*, exploiting a vulnerability soon after patches were announced. Such pathogens make it increasingly difficult for organizations to patch vulnerabilities before a catastrophe occurs. As a result, we argue that defenses are necessary, but not sufficient, for fully protecting distributed systems and data on Internet hosts from catastrophes.

In this chapter, we propose a new approach for designing distributed systems to survive Internet catastrophes called informed replication. The key observation that makes informed replication both feasible and practical is that Internet epidemics exploit shared vulnerabilities. By replicating a system service on hosts that do not have the same vulnerabilities, a pathogen that exploits one or more vulnerabilities cannot cause all replicas to fail. For example, to prevent a distributed system from failing due to a pathogen that exploits vulnerabilities in Web servers, the system can place replicas on hosts running different Web server software.

The software of every system inherently is a shared vulnerability that represents a risk to using the system, and systems designed to use informed replication are no different. Substantial effort has gone into making systems themselves more secure, and our design approach can certainly benefit from this effort. However, with the dramatic rise of worm epidemics, such systems are now increasingly at risk to large-scale failures due to vulnerabilities in *unrelated* software running on the host. Informed replication reduces this new source of risk.

This chapter makes four contributions. First, we develop a system model using the *core* abstraction of Chapter 2. Recall that a core is a reliable minimal subset of components such that the probability of having all hosts in a core failing is negligible. Compared to the use of cores in the previous chapters, we need a method for determining the dependencies among hosts that cause dependent failures, and that enables a proper selection of cores. To reason about dependent failures, we associate *attributes* with hosts. Attributes represent characteristics of the host that can make it prone to failure, such as its operating system and network services. Since hosts often have many characteristics that make it vulnerable to failure, we group host attributes together into *configurations* to represent the set of vulnerabilities for a host. A system can use the configurations of all hosts in the system to determine how many replicas are needed, and on which hosts those replicas should be placed, to survive a worm epidemic.

Second, the efficiency of informed replication fundamentally depends upon the degree of software diversity among the hosts in the system, as more homogeneous host populations result in a larger storage burden for particular hosts. To evaluate the degree of software heterogeneity found in an Internet setting, we measure and characterize the diversity of the operating systems and network services of hosts in the UCSD network. The operating system is important because it is the primary attribute differentiating hosts, and network services represent the targets for exploit by worms. The results of this study indicate that such networks have sufficient diversity to make informed replication feasible.

Third, we develop heuristics for computing cores that have a number of attractive features. They provide excellent reliability guarantees, ensuring that user data survives attacks of single- and double-exploit pathogens with probability greater than $0.99$. They have low overhead, requiring fewer than 3 copies to cope with single-exploit pathogens, and fewer than 5 copies to cope with double-exploit pathogens. They bound the number of replica copies stored by any host, limiting the storage burden on any single host. Finally, the heuristics lend themselves to a fully distributed implementation for scalability. Any host can determine its replica set (its core) by contacting a constant number of other hosts in the system, independent of system size.

Finally, to demonstrate the feasibility and utility of our approach, we apply informed replication to the design and implementation of Phoenix. Phoenix is a cooperative, distributed remote backup system that protects stored data against Internet catastrophes that cause data loss [MS04]. The usage model of Phoenix is straightforward: users specify an amount $F$ of bytes of their disk space for management by the system, and the system protects a proportional amount $F/k$ of their data using storage provided by other hosts, for some value of $k$. We implement Phoenix as a service layered on the Pastry DHT [RD01] in the Macedon framework [RKB$^+$04], and evaluate its ability to survive emulated catastrophes on the PlanetLab testbed.

The remainder of this chapter is organized as follows. Section 6.1 discusses related work. Section 6.2 describes our system model for representing dependent failures. Section 6.3 describes our measurement study of the software diversity of hosts in a large network, and Section 6.4 describes and evaluates heuristics for computing cores. Section 6.5 describes the design and implementation of Phoenix, and Section 6.6 describes the evaluation of Phoenix. Finally, Section 6.8 concludes.

## 6.1 Related work

Most distributed systems are not designed such that failures are independent, and there has been recent interest in protocols for systems where failures are correlated. Quorum-based protocols, which implement replicated update by reading and writing overlapping subsets of replicas, are easily adapted to correlated failures. A model of dependent failures was introduced for Byzantine-tolerant quorum systems [MR97a]. This model, called a *fail-prone system*, is a dual representation of the model (*cores*) that we use here. Our model was developed as part of a study of lower bounds and optimal protocols for consensus in environments where failures can be correlated.

The ability of Internet pathogens to spread through a vulnerable host population on the network fundamentally depends on three properties of the network: the number of susceptible hosts that could be infected, the number of infected hosts actively spreading the pathogen, and the contact rate at which the pathogen spreads. Various approaches have been developed for defending against such epidemics that address each of these properties.

Prevention techniques, such as patching [Mic, Sym, WGSZ04] and overflow guarding [CPM$^+$98, WFBA00], prevent pathogens from exploiting vulnerabilities, thereby reducing the size of the vulnerable host population and limiting the extent of a worm outbreak. However, these approaches have the traditional limitations of ensuring soundness and completeness, or leave windows of vulnerability due to the time required to develop, test, and deploy.

Treatment techniques, such as disinfection [cod01, crc01] and vaccination [SK03], remove software vulnerabilities after they have been exploited and reduce the rate of infection as hosts are treated. However, such techniques are reactive in nature and hosts still become infected. Further, counter-worms have questionable legality, and automatic vaccination has limiting constraints on deployment (*e.g.*, requiring source to patch).

Software patching also treats infected hosts, although the slow rate at which users patch infected systems has little impact on the initial propagation of worms and defending against them. For example, during the Code-Red epidemic it took sixteen days for most hosts to eliminate the underlying vulnerability and thousands had not patched their systems six weeks later [MSB02]. Recent proposals include anti-worms that disinfect hosts using the same propagation methods as the original worm [cod01, crc01], and vaccinating hosts by automatically detecting infection in software and generating and applying patches online [SK03].

Containment techniques, such as throttling [Lis06, Wil02] and filtering [MSVS03, TK02], block infectious communication between infected and uninfected hosts, thereby reducing or potentially halting the contact rate of a spreading pathogen. The efficacy of reactive containment fundamentally depends upon the ability to quickly detect a new pathogen [LLO$^+$03, MVS01, WSP04, ZGGT03], characterize it to create filters specific to infectious traffic [Hya04, KA94, KC03, SEVS04], and deploy such filters in the network [LMK$^+$03, TW03]. Unfortunately, containment at Internet scales is challenging, requiring short reaction times and extensive deployment [MSVS03, Won04]. Again, since containment is inherently reactive, some hosts always become infected.

Proactive defenses include reducing the propagation of pathogens, such as the La Brea "tarpit" for slowing TCP-based worms [Lis06] and throttling connection rates [Wil02]. Reactive defenses contain the spread of pathogens by blocking infectious communication in reaction to the onset of an epidemic [MSVS03, TK02].

Such defenses typically involve installing filters in firewalls or routers to block infected communication. For example, Cisco's Network Based Application Recognition (NBAR) feature [Cis] allows a router to block particular TCP sessions based on the presence of individual strings in the TCP stream. NBAR was first used in blocking the spread of the Code-Red worm, enabling networks to prevent the propagation of worm probes on port 80 while allowing legitimate traffic to reach Web servers.

Various approaches take advantage of software heterogeneity to make systems fault-tolerant. N-version programming uses different implementations of the same service to prevent correlated failures across implementations. Castro's Byzantine fault tolerant NFS service (BFS) is one such example [CL02] and provides excellent fault-tolerance guarantees, but requires multiple implementations of every service. Scrambling the layout and execution of code can introduce heterogeneity into deployed software [BAF$^+$03]. However, such approaches can make debugging, troubleshooting, and maintaining software considerably more challenging. In contrast, our approach takes advantage of existing software diversity.

Phoenix is one of many proposed cooperative systems for providing archival and backup services. For example, Intermemory [Che99] and Oceanstore [KBC$^+$00] enable stored data to persist indefinitely on servers distributed across the Internet. As with Phoenix, Oceanstore proposes mechanisms to cope with correlated failures [WMK02]. The approach, however, is reactive and does not enable recovery after Internet catastrophes. With Pastiche [CN02], pStore [BBST01], and CIBS [LEB$^+$03], users relinquish a fraction of their computing resources to collectively create a backup service. However, these systems target localized failures simply by storing replicas offsite. Such systems provide similar functionality as Phoenix, but are not designed to survive widespread correlated failures of Internet catastrophes. Finally, Glacier is a system specifically designed to survive highly correlated failures like Internet catastrophes [HMD05]. In contrast to Phoenix, Glacier assumes a very weak failure model and instead copes with catastrophic failures via massive replication. Phoenix relies upon a stronger failure model, but replication in Phoenix is modest in comparison.

## 6.2 System model

As a first step toward the development of a technique to cope with Internet catastrophes, in this section we describe our system model for representing and reasoning about dependent failures, and discuss the granularity at which we represent software diversity. This system model is based on the core abstraction of Chapter 2.

### 6.2.1 Representing dependent failures

Consider a system composed of a set $\mathcal{H}$ of hosts each of which is capable of holding certain objects. These hosts can fail (for example, by crashing) and, to keep these objects available, they need to be replicated. A simple replication strategy is to determine the maximum number $t$ of hosts that can fail at any time, and then maintain more than $t$ replicas of each object.

However, using more than $t$ replicas may lead to excessive replication when host failures are correlated. As a simple example, consider three hosts $\{h_1, h_2, h_3\}$ where the failures of $h_1$ and $h_2$ are correlated while $h_3$ fails independent of the other hosts. If $h_1$ fails, then the probability of $h_2$ failing is high. As a result, one might set $t = 2$ and thereby require $t + 1 = 3$ replicas. However, if we place replicas on $h_1$ and $h_3$, the object's availability may be acceptably high with just two replicas.

To better address issues of optimal replication in the face of correlated failures, we have defined an abstraction that we call a *core*. A core is a minimal set of hosts such that, in any execution, at least one host in the core does not fail. In the above example, both $\{h_1, h_3\}$ and $\{h_2, h_3\}$ are cores. $\{h_1, h_2\}$ would not be a core since the probability of both failing is too high and $\{h_1, h_2, h_3\}$ would not be a core since it is not minimal. Using this terminology, a central problem of informed replication is the identification of cores based on the correlation of failures.

An Internet catastrophe causes hosts to fail in a correlated manner because all hosts running the targeted software are vulnerable. Operating systems and Web servers are examples of software commonly exploited by Internet pathogens [MSB02, Sop04]. Hence we characterize a host's vulnerabilities by the software they run. We associate with each host a set of *attributes*, where each attribute is a canonical name of a software package or system that the host runs; in Section 6.2.2 below, we discuss the tradeoffs of representing software packages at different granularities. We call the combined representation of all attributes of a host the *configuration* of the host. An example of a configuration is $\{$*Windows, IIS, IE*$\}$, where *Windows* is a canonical name for an operating system, *IIS* for a Web server package, and *IE* for a Web browser. Agreeing on canonical names for attribute values is essential to ensure that dependencies of host failures are appropriately captured.

An Internet pathogen can be characterized by the set of attributes $A$ that it targets. Any host that has none of the attributes in $A$ is not susceptible to the pathogen. A core is a minimal set $C$ of hosts such that, for each pathogen, there is a host $h$ in $C$ that is not susceptible to the pathogen. Internet pathogens often target a single (possibly cross-platform) vulnerability, and the ones that target multiple vulnerabilities target the same operating system. Assuming that any attribute is susceptible to attack, we can redefine a core using attributes: a core is a minimal set $C$ of processes such that no attribute is common to all hosts in $C$. In Section 6.4.4, we relax this assumption and show how to extend our results to tolerate pathogens that can exploit multiple vulnerabilities.

To illustrate these concepts, consider the system described in Example 6.2.1.

In this system, hosts are characterized by six attributes which we classify for clarity into operating system, Web server, and Web browser.

$H_1$ and $H_2$ comprise what we call an *orthogonal core*, which is a core composed of hosts that have disjoint configurations. Given our assumption that Internet pathogens target only one vulnerability or multiple vulnerabilities on one platform, an orthogonal core will contain exactly two hosts. $\{H_1, H_3, H_4\}$ is also a core because there is no attribute present in all hosts, and it is minimal.

**Example 6.2.1**

*Attributes:* *Operating System* = $\{$Unix, Windows$\}$*;*

*Web Server* = $\{$Apache, IIS$\}$*;*

*Web Browser* = $\{$IE, Netscape$\}$.

*Hosts:* $H_1 = \{$Unix, Apache, Netscape$\}$*;*

$H_2 = \{$Windows, IIS, IE$\}$*;*

$H_3 = \{$Windows, IIS, Netscape$\}$*;*

$H_4 = \{$Windows, Apache, IE$\}$.

*Cores* $= \{\{H_1, H_2\}, \{H_1, H_3, H_4\}\}$.

The smaller core $\{H_1, H_2\}$ might appear to be the better choice since it requires less replication. Choosing the smallest core, however, can have an adverse effect on individual hosts if many hosts use this core for placing replicas. To represent this effect, we define *load* to be the amount of storage a host provides to other hosts. In environments where some configurations are rare, hosts with the rare configurations may occur in a large percentage of the smallest cores. Thus, hosts with rare configurations may have a significantly higher load than the other hosts. Indeed, having a rare configuration can increase a host's load even if the smallest core is not selected. For example, in Example 6.2.1, $H_1$ is the only host that has a flavor of Unix as its operating system. Consequently, $H_1$ is present in both cores.

To make our argument more concrete, consider the well-known worms in Table 6.1 that were unleashed in the past few years. For each worm, given two hosts with one not running Windows or not running a specific server such as a Web server or a database, at least one survives the attack. With even a very modest amount of heterogeneity, our method of constructing cores includes such pairs of hosts.

### 6.2.2 Attribute granularity

Attributes can represent software diversity at many different granularities. The choice of attribute granularity balances resilience to pathogens, flexibility for placing replicas, and degree of replication. An example of the coarsest representation is for a host to have a configuration comprising a single attribute for the generic class of operating system, e.g., "Windows", "Unix", etc. This single attribute represents the potential vulnerabilities of all versions of software running on all versions of the same class of operating system. As a result, replicas would always be placed on hosts with different operating systems. A less coarse representation is to have attributes for the operating system as well as all network services running on the host. This representation yields more freedom for placing replicas. For example, we can place replicas on hosts with the same class of operating system if they run different services. The core $\{H_1, H_3, H_4\}$ in Example 6.2.1 is an example of this situation since $H_3$ and $H_4$ both run Windows. More fine-grained representations can have attributes for different versions of operating systems and applications. For example, we can represent the various releases of Windows, such as "Windows 2000" and "Windows XP", or even versions such as "NT 4.0sp4" as attributes. Such fine-grained attributes provide considerable flexibility in placing replicas. For example, we can place a replica on an NT host and an XP host to protect against worms such as Code Red that exploit an NT service but not an XP service. But doing so greatly increases the cost and complexity of collecting and representing host attributes, as well as computing cores to determine replica sets.

Our initial work [JBM$^+$03] suggested that informed replication can be effective with relatively coarse-grained attributes for representing software diversity. As a result, we use attributes that represent just the class of operating system and network services on hosts in the system, and not their specific versions. In subsequent sections,

**Table 6.1:** Recent well-known pathogens

| **Worm** | **Form of infection (Service)** | **Platform** |
|---|---|---|
| Code Red | port 80/http (MS IIS) | Windows |
| Nimda | multiple: email; Trojan horse versions using open network shares (SMB: ports 137–139 and 445); port 80/HTTP (MS IIS); Code Red backdoors | Windows |
| Sapphire | port 1434/udp (MS SQL, MSDE) | Windows |
| Sasser | port 445/tcp (LSASS) | Windows |
| Witty | port 4000/udp (BlackICE) | Windows |

we show that, when representing diversity at this granularity, hosts in an enterprise-scale network have substantial and sufficient software diversity for efficiently supporting informed replication. Our experience suggests that, although we can represent software diversity at finer attribute granularities such as specific software versions, there is not a compelling need to do so.

## 6.3 Host diversity

With informed replication, the difficulty of identifying cores and the resulting storage load depend on the actual distribution of attributes among a set of hosts. To better understand these two issues, we measured the software diversity of a large set of hosts at UCSD. In this section, we first describe the methodology we used, and discuss the biases and limitations our methodology imposes. We then characterize the operating system and network service attributes found on the hosts, as well as the host configurations formed by those attributes.

### 6.3.1 Methodology

On our behalf, UCSD Network Operations used the *Nmap* tool [Ins04] to scan IP address blocks owned by UCSD to determine the host type, operating system, and network services running on the host. Nmap uses various scanning techniques to classify devices connected to the network. To determine operating systems, Nmap interacts with the TCP/IP stack on the host using various packet sequences or packet contents that produce known behaviors associated with specific operating system TCP/IP implementations. To determine the network services running on hosts, Nmap scans the host port space to identify all open TCP and UDP ports on the host. We anonymized host IP addresses prior to processing.

Due to administrative constraints collecting data, we obtained the operating system and port data at different times. We had a port trace collected between December 19–22, 2003, and an operating system trace collected between December 29, 2003 and January 7, 2004. The port trace contained 11,963 devices and the operating system trace contained 6,395 devices.

Because we are interested in host data, we first discarded entries for specialized devices such as printers, routers, and switches. We then merged these traces to produce a combined trace of hosts that contained both operating system data and open port data for the same set of hosts. When fingerprinting operating systems, Nmap determines both a class (e.g., Windows) as well as a version (e.g., Windows XP). For added consistency, we discarded host information for those entries that did not have consistent OS class and version info. The result was a data set with operating system and port data for 2,963 general-purpose hosts.

Our data set was constructed using assumptions that introduced biases. First, worms exploit vulnerabilities that are present in network services. We make the assumption that two hosts that have the same open port are running the same network service and thus have the same vulnerability. In fact, two hosts may use a given port to run different services, or even different versions (with different vulnerabilities) of the same service. Second, ignoring hosts that Nmap could not consistently fingerprint could bias the host traces that were used. Third, DHCP-assigned host addresses are reused. Given the time elapsed between the time operating system information was collected and port information was collected, an address in the operating system trace may refer to a different host in the port trace. Further, a host may appear multiple times with different addresses either in the port trace or in the operating system trace. Consequently, we may have combined information from different hosts to represent one host or counted the same host multiple times.

The first assumption can make two hosts appear to share vulnerabilities when in fact they do not, and the second assumption can consistently discard configurations that otherwise contribute to a less skewed distribution of configurations. The third assumption may make the distribution of configurations seem less skewed, but operating system and port counts either remain the same (if hosts do not appear multiple times in the traces) or increase due to repeated configurations. The net effect of our assumptions is to make operating system and port distributions appear to be less diverse than it really is, although it may have the opposite effect on the distribution of configurations.

Another bias arises from the environment we surveyed. A university environment is not necessarily representative of the Internet, or specific subsets of it. We suspect that such an environment is more diverse in terms of software use than other environments, such as the hosts in a corporate environment or in a governmental agency. On the other hand, there are perhaps thousands of universities with a large setting connected to the Internet around the globe, and so the conclusions we draw from our data are undoubtedly not singular.

### 6.3.2 Attributes

Together, the hosts in our study have 2,569 attributes representing operating systems and open ports. Table 6.2 shows the ten most prevalent operating systems and open ports identified on the general purpose hosts. Table 6.2.a shows the number and percentage of hosts running the named operating systems. As expected, Windows is the most prevalent OS (54% of general purpose hosts). Individually, Unix variants vary in prevalence (0.03–10%), but collectively they comprise a substantial fraction of the hosts (38%).

Table 6.2.b shows the most prevalent open ports on the hosts and the network services typically associated with those port numbers. These ports correspond to ser-

**Table 6.2:** Top 10 operating systems (a) and ports (b) among the 2,963 general-purpose hosts

(a)

| OS | |
|---|---|
| **Name** | **Count (%)** |
| Windows | 1604 (54.1) |
| Solaris | 301 (10.1) |
| Mac OS X | 296 (10.0) |
| Linux | 296 (10.0) |
| Mac OS | 204 (6.9) |
| FreeBSD | 66 (2.2) |
| IRIX | 60 (2.0) |
| HP-UX | 32 (1.1) |
| BSD/OS | 28 (0.9) |
| Tru64 Unix | 22 (0.7) |

(b)

| Port | |
|---|---|
| **Number** | **Count (%)** |
| 139 (netbios-ssn) | 1640 (55.3) |
| 135 (epmap) | 1496 (50.4) |
| 445 (microsoft-ds) | 1157 (39.0) |
| 22 (sshd) | 910 (30.7) |
| 111 (sunrpc) | 750 (25.3) |
| 1025 (various) | 735 (24.8) |
| 25 (smtp) | 575 (19.4) |
| 80 (httpd) | 534 (18.0) |
| 21 (ftpd) | 528 (17.8) |
| 515 (printer) | 462 (15.6) |

vices running on hosts, and represent the points of vulnerability for hosts. On average, each host had seven ports open. However, the number of ports per host varied considerably, with 170 hosts only having one port open while one host (running a firewall software) had 180 ports open. Windows services dominate the network services running on hosts, with netbios-ssn (55%), epmap (50%), and domain services (39%) topping the list. The most prevalent services typically associated with Unix are sshd (31%) and sunrpc (25%). Web servers on port 80 are roughly as prevalent as ftp (18%).

These results show that the software diversity is significantly skewed. Most hosts have open ports that are shared by many other hosts (Table 6.2.b lists specific examples). However, most attributes are found on few hosts, *i.e.*, most open ports are open on only a few hosts. From our traces, we observe that the first 20 most prevalent attributes are found on 10% or more of hosts, but the remaining attributes are found on fewer hosts.

These results are encouraging for the process of finding cores. Having many attributes that are not widely shared makes it easier to find replicas that cover each

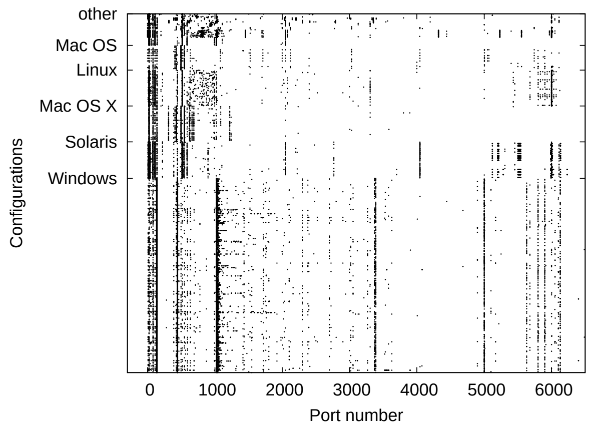

**Figure 6.1:** Visualization of UCSD configurations

other's attributes, preventing a correlated failure from affecting all replicas. We examine this issue next.

### 6.3.3 Configurations

Each host has multiple attributes comprised of its operating system and network services, and together these attributes determine its configuration. The distribution of configurations among the hosts in the system determines the difficulty of finding core replica sets. The more configurations shared by hosts, the more challenging it is to find small cores.

Figure 6.1 is a qualitative visualization of the space of host configurations. It shows a scatter plot of the host configurations among the UCSD hosts in our study. The x-axis is the port number space from 0–6500, and the y-axis covers the entire set of 2,963 host configurations grouped by operating system family. A dot corresponds to an open port on a host, and each horizontal slice of the scatter plot corresponds to the configuration of open ports for a given host. We sort groups in decreasing size according to the operating systems listed in Table 6.2: Windows hosts start at the bottom, then Solaris, Mac OS X, etc. Note that we have truncated the port space in the graph; hosts had open ports above 6500, but showing these ports did not add any additional insight and obscured patterns at lower, more prevalent port numbers.

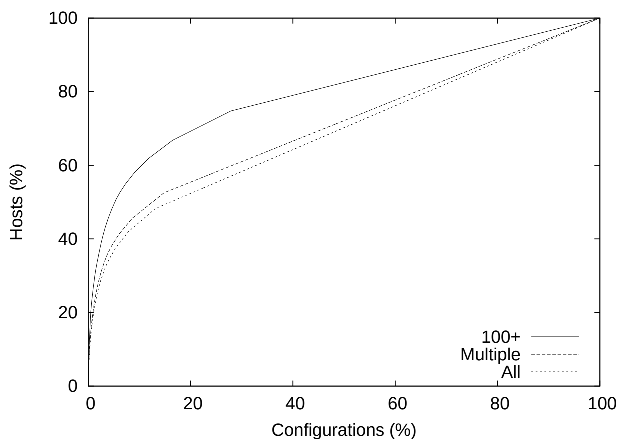

**Figure 6.2:** Distribution of configurations

Figure 6.1 shows a number of interesting features of the configuration space. The marked vertical bands within each group indicate, as one would expect, strong correlations of network services among hosts running the same general operating system. For example, most Windows hosts run the epmap (port 135) and netbios (port 139) services, and many Unix hosts run sshd (port 22) and X11 (port 6000). Also, in general, non-Windows hosts tend to have more open ports (8.3 on average) than Windows hosts (6.0 on average). However, the groups of hosts running the same operating system still have substantial diversity within the group. Although each group has strong bands, they also have a scattering of open ports between the bands contributing to diversity within the group. Lastly, there is substantial diversity among the groups. Windows hosts have different sets of open ports than hosts running variants of Unix, and these sets even differ among Unix variants. We take advantage of these characteristics to develop heuristics for determining cores in Section 6.4.

Figure 6.2 provides a quantitative evaluation of the diversity of host configurations. It shows the cumulative distribution of configurations across hosts for different classes of port attributes, with configurations on the x-axis sorted by decreasing order of prevalence. A distribution in which all configurations are equally prevalent would be a straight diagonal line. Instead, the results show that the distribution of configurations is skewed, with a majority of hosts accounting for only a small percentage of all configurations. For example, when considering all attributes, 50% of hosts comprise just 20% of configurations. In addition, reducing the number of port attributes considered further skews the distribution. For example, when only considering ports that appear on more than one host, shown by the “Multiple” line, 15% of the configurations represent over 50% of the hosts. And when considering only the port attributes that appear on at least 100 hosts, only 8% of the configurations represent over 50% of the hosts. Skew in the configuration distribution makes it more difficult to find cores for those hosts that share more prevalent configurations with other hosts. In the next section, however, we show that host populations with diversity similar to UCSD are sufficient for efficiently constructing cores that result in a low storage load.

## 6.4 Surviving catastrophes

With informed replication, each host $h$ constructs a core $\mathit{Core}(h)$ based on its configuration and the configuration of other hosts.[1] Computing a core of optimal size, however, is NP-hard. Hence, we use heuristics to compute $\mathit{Core}(h)$. We postpone a discussion on the complexity of finding cores until Section 6.7. In this section, we first discuss a data structure for representing advertised configurations that is amenable to heuristics for computing cores. We then describe four heuristics and evaluate via simulation the properties of the cores that they construct. As a basis for our simulations, we use the set of hosts $\mathcal{H}$ obtained from the traces discussed in Section 6.3.

### 6.4.1 Advertised configurations

Our heuristics are different versions of greedy algorithms: a host $h$ repeatedly selects other hosts to include in $\mathit{Core}(h)$ until some condition is met. Hence we chose a representation that makes it easier for a greedy algorithm to find good candidates to include in $\mathit{Core}(h)$. This representation is a three-level hierarchy.

[1]More precisely, $\mathit{Core}(h)$ is a core constrained to contain $h$. That is, $\mathit{Core}(h) \setminus \{h\}$ may itself be minimal, but we require $h \in \mathit{Core}(h)$.

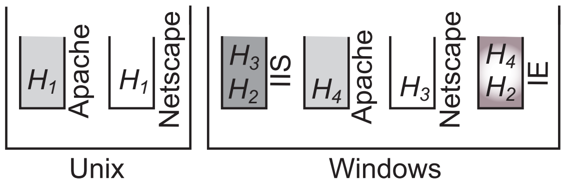

**Figure 6.3:** Illustration of containers and sub-containers

The top level of the hierarchy is the operating system that a host runs, the second level includes the applications that run on that operating system, and the third level are hosts. Each host runs one operating system, and so each host is subordinate to its operating system in the hierarchy (we can represent hosts running multiple virtual machines as multiple virtual hosts in a straightforward manner). Since most applications run predominately on one platform, hosts that run a different operating system than $h$ are likely good candidates for including in *Core*$(h)$. We call the first level the *containers* and the second level the *sub-containers*. Each sub-container contains a set of hosts. Figure 6.3 illustrates these abstractions using the configurations of Example 6.2.1.

More formally, let $\mathcal{O}$ be the set of canonical operating system names and $\mathcal{C}$ be the set of containers. Each host $h$ has an attribute $h.os$ that is the canonical name of the operating system on $h$. The function $m_c : \mathcal{O} \rightarrow \mathcal{C}$ maps operating system name to container; thus, $m_c(h.os)$ is the container that contains $h$.

Let $h.apps$ denote the set of canonical names of the applications that are running on $h$, and let $\mathcal{A}$ be the canonical names of all of the applications. We denote with $\mathcal{S}$ the set of sub-containers and with $m_s : \mathcal{C} \rightarrow 2^{\mathcal{S}}$ the function that maps a container to its sub-containers. The function $m_h : \mathcal{C} \times \mathcal{A} \rightarrow \mathcal{S}$ maps a container and application to a sub-container; thus, for each $a \in h.apps$, host $h$ is in each sub-container $m_h(m_c(h.os), a)$.

At this high level of abstraction, advertising a configuration is straightforward. Initially $\mathcal{C}$ is empty. To advertise its configuration, a host $h$ first ensures that there is a container $c \in \mathcal{C}$ such that $m_c(h.os) = c$. Then, for each attribute $a \in h.apps$, $h$ ensures that there is a sub-container $m_h(c, a)$ containing $h$.

### 6.4.2 Computing cores

The heuristics we describe in this section compute *Core*$(h)$ in time linear with the number of attributes in $h.apps$. These heuristics reference the set $\mathcal{C}$ of containers and the three functions $m_c, m_s$ and $m_h$, but they do not reference the full set $\mathcal{A}$ of attributes. In addition, these heuristics do not enumerate $\mathcal{H}$, but they do reference the configuration of hosts (to reference the configuration of a host $h'$, they reference $h'.os$ and $h'.apps$). Thus, the container/sub-container hierarchy is the only data structure that the heuristics use to compute cores.

#### Metrics

We evaluate our heuristics using three metrics:

- **Average core size**: $|$*Core*$(h)|$ averaged over all $h \in \mathcal{H}$. This metric is important because it determines how much capacity is available in the system. As the average core size increases, the total capacity of the system decreases.
- **Maximum load**: The load of a host $h'$ is the number of cores *Core*$(h)$ of which $h'$ is a member. The maximum load is the largest load of any host $h' \in \mathcal{H}$.
- **Average coverage**: We say that an attribute $a$ of a host $h$ is *covered* in *Core*$(h)$ if there is at least one other host $h'$ in *Core*$(h)$ that does not have $a$. Thus, an exploit of attribute $a$ can affect $h$, but not $h'$, and so not all hosts in *Core*$(h)$ are affected. The *coverage* of *Core*$(h)$ is the fraction of attributes of $h$ that are covered. The *average coverage* is the average of the coverages of *Core*$(h)$ over all hosts $h \in \mathcal{H}$. A high average coverage indicates a higher resilience to Internet catastrophes: many hosts have most or all of their attributes covered. We return to this discussion of what coverage means in practice in Section 6.4.3, after we present most of our simulation results for context.

For brevity, we use the terms core size, load, and coverage to indicate average core size, maximum load, and average coverage, respectively. Where we do refer to these terms in the context of a particular host, we say so explicitly.

A good heuristic will determine cores with small size, low load, and high coverage. Coverage is the most critical metric because it determines how well it does in guaranteeing service in the event of a catastrophe. Coverage may not equal one either because there was no host $h'$ that was available to cover an attribute $a$ of $h$, or because the heuristic failed to identify such a host $h'$. As shown in the following sections, the second case rarely happens with our heuristics.

Note that, as a single number, the coverage of a given *Core*$(h)$ does not fully capture its resilience. For example, consider host $h_1$ with two attributes and host $h_2$ with 10 attributes. If *Core*$(h_1)$ covers only one attribute, then *Core*$(h_1)$ has a coverage of 0.5. If *Core*$(h_2)$ has the same coverage, then it covers only 5 of the 10 attributes. There are more ways to fail all of the hosts in *Core*$(h_2)$ than those in *Core*$(h_1)$. Thus, we also use the number of cores that do not have a coverage of 1.0 as an extension of the coverage metric.

## Heuristics

We begin by using simulation to evaluate a naive heuristic called **Random** that we use as a basis for comparison. It is not a greedy heuristic and does not reference the advertised configurations. Instead, $h$ simply chooses at random a subset of $\mathcal{H}$ of a given size containing $h$.

The first row of Table 6.3 shows the results of **Random** using one run of our simulator. We set the size of the cores to 5, *i.e.*, **Random** chose 5 random hosts to form a core. The coverage of 0.977 may seem high, but there are still many cores that have uncovered attributes and choosing a core size smaller than five results in even lower coverage. The load is 12, which is significantly higher than the lower bound of 5.[2]

Our first greedy heuristic **Uniform** ("uniform" selection among operating systems) operates as follows. First, it chooses a host with a different operating system than

[2]To meet this bound, number the hosts in $\mathcal{H}$ from 0 to $|\mathcal{H}| - 1$. Let *Core*$(h)$ be the hosts $\{h + i \pmod{|\mathcal{H}|} : i \in \{0, 1, 2, 3, 4\}\}$.

**Table 6.3:** A typical run of the heuristics

| | **Core size** | **Coverage** | **Load** |
|---|---|---|---|
| **Random** | 5 | 0.977 | 12 |
| **Uniform** | 2.56 | 0.9997 | 284 |
| **Weighted** | 2.64 | 0.9995 | 84 |
| **DWeighted** | 2.58 | 0.9997 | 91 |

$h.os$ to cover this attribute. Then, for each attribute $a \in h.apps$, it chooses both a container $c \in \mathcal{C} \setminus \{m_c(h.os)\}$ and a sub-container $sc \in m_s(c) \setminus \{m_h(c, a)\}$ at random. Finally, it chooses a host $h'$ at random from $sc$. If $a \notin h'.apps$ then it includes $h'$ in *Core*$(h)$. Otherwise, it tries again by choosing a new container $c$, sub-container $sc$, and host $h'$ at random. **Uniform** repeats this procedure `diff_OS` times in an attempt to cover $a$ with *Core*$(h)$. If it fails to cover $a$, then the heuristic tries up to `same_OS` times to cover $a$ by choosing a sub-container $sc \in m_c(h.os)$ at random and a host $h'$ at random from $sc$.

The goal for having two steps, one with `diff_OS` and another with `same_OS`, is to first exploit diversity across operating systems, and then to exploit diversity among hosts within the same operating system group. Referring back to Figure 6.1, the set of prevalent services among hosts running the same operating system varies across the different operating systems. In the case the attribute cannot be covered with hosts running other operating systems, the diversity within an operating system group may be sufficient to find a host $h'$ without attribute $a$.

In all of our simulations, we set `diff_OS` to 7 and `same_OS` to 4. After experimentation, these values have provided a good trade-off between number of useless tries and obtaining good coverage. However, we have yet to study how to in general choose good values of `diff_OS` and `same_OS`. Figure 6.4 shows the pseudocode for **Uniform**.

The second row of Table 6.3 shows the performance of **Uniform** for a representative run of our simulator. The core size is close to the minimum size of two, and

```
Algorithm Uniform on input $h$:
integer $i$;
core $\leftarrow \{h\}$;
$\mathcal{C}' \leftarrow \mathcal{C} \setminus \{m_c(h.os)\}$
for each attribute $a \in h.apps$
  $i \leftarrow 0$
  while ($a$ is not covered) $\wedge$
        ($i \leq$ diff_OS + same_OS)
   if ($i \leq$ diff_OS) choose randomly $c \in \mathcal{C}'$
      else $c \leftarrow m_c(h.\mathrm{os})$
   choose randomly $sc \in m_s(c) \setminus \{m_h(c,a)\}$
   choose a host $h' \in sc : h' \neq h$
   if ($h'$ covers $a$) add $h'$ to core
   $i \leftarrow i + 1$
return core
```

**Figure 6.4: Uniform** heuristic

the coverage is very close to the ideal value of one. This means that using **Uniform** results in significantly better capacity and improved resilience than **Random**. On the other hand, the load is very high: there is at least one host that participates in 284 cores. The load is so high because $h$ chooses containers and sub-containers uniformly. When constructing the cores for hosts of a given operating system, the other containers are referenced roughly the same number of times. Thus, **Uniform** considers hosts running less prevalent operating systems for inclusion in cores a disproportionately large number of times. A similar argument holds for hosts running less popular applications.

This behavior suggests refining the heuristic to choose containers and applications weighted on the popularity of their operating systems and applications. Given a container $c$, let $N_c(c)$ be the number of distinct hosts in the sub-containers of $c$, and given a set of containers $C$, let $N_c(C)$ be the sum of $N_c(c)$ for all $c \in C$. The heuristic **Weighted** ("weighted" OS selection) is the same as **Uniform** except that for the first diff_OS attempts, $h$ chooses a container $c$ with probability $N_c(c)/N_c(\mathcal{C} \setminus \{m_c(h.os)\})$. Heuristic **DWeighted** ("doubly-weighted" selection) takes this a step further. Let $N_s(c,a)$ be $|m_h(c,a)|$ and $N_s(c,A)$ be the size of the union of $m_h(c,a)$ for all $a \in A$. Heuristic **DWeighted** is the same as **Weighted** except that, when considering attribute $a \in h.apps$, $h$ chooses a host from sub-container $m_h(c,a')$ with probability

$N_s(c, a')/N_s(c, \mathcal{A} \setminus \{a\})$.

In the third and fourth rows of Table 6.3, we show a representative run of our simulator for both of these variations. The two variations result in comparable core sizes and coverage as **Uniform**, but significantly reduce the load. The load is still very high, though: at least one host ends up being assigned to over 80 cores.

Another approach to avoid a high load is to simply disallow it at the risk of decreasing the coverage. That is, for some value of $L$, once a host $h'$ is included in $L$ cores, $h'$ is removed from the structure of advertised configurations. Thus, the load of any host is constrained to be no larger than $L$.

What is an effective value of $L$ that reduces load while still providing good coverage? We answer this question by first establishing a lower bound on the value of $L$. Suppose that $a$ is the most prevalent attribute (either service or operating system) among all attributes, and it is present in a fraction $x$ of the host population. As a simple application of the pigeonhole principle, some host must be in at least $l$ cores, where $l$ is defined as:

$$l = \left\lceil \frac{|\mathcal{H}| \cdot x}{|\mathcal{H}| \cdot (1 - x)} \right\rceil = \left\lceil \frac{x}{(1 - x)} \right\rceil \tag{6.1}$$

Thus, the value of $L$ cannot be smaller than $l$. Using Table 6.2, we have that the most prevalent attribute (port 139) is present in 55.3% of the hosts. In this case, $l = 2$.

Using simulation, we now evaluate our heuristics in terms of core size, coverage, and load as a function of the load limit $L$. Figures 6.5–6.8 present the results of our simulations. In these figures, we vary $L$ from the minimum 2 through a high load of 10. All the points shown in these graphs are the averages of eight simulated runs with error bars (although they are too narrow to be seen in some cases). For Figures 6.5–6.7, we use the standard error to determine the limits of the error bars, whereas for Figure 6.8 we use the maximum and minimum observed among our samples. When using load limit as a threshold, the order in which hosts request cores from $\mathcal{H}$ will produce different results. In our experiments, we randomly choose eight different orders of enumerating

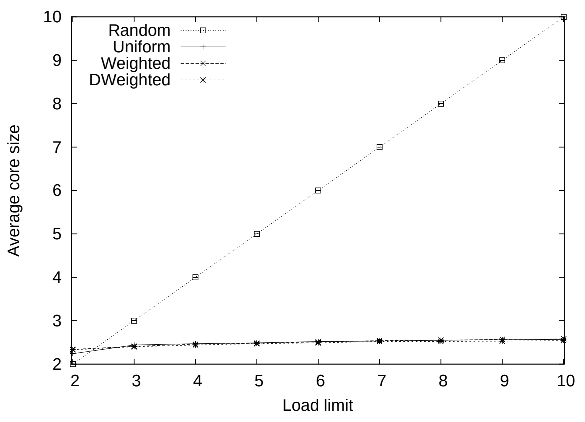

**Figure 6.5:** Average core size

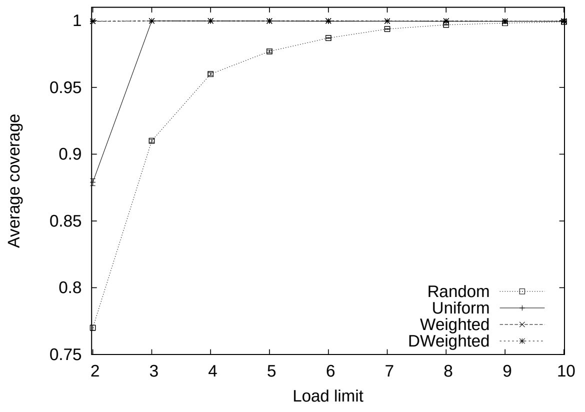

**Figure 6.6:** Average coverage

$\mathcal{H}$ for constructing cores. For each heuristic, each run of the simulator uses a different order. Finally, we vary the core size of **Random** using the load limit $L$ to illustrate its effectiveness across a range of core sizes.

Figure 6.5 shows the average core size for the four algorithms for different values of $L$. According to this graph, **Uniform**, **Weighted**, and **DWeighted** do not differ much in terms of core size. The average core size of **Random** increases linearly with $L$ by design.

In Figure 6.6, we show results for coverage. Coverage is slightly smaller than $1.0$ for **Uniform**, **Weighted**, and **DWeighted** when $L$ is greater or equal to three. For $L = 2$, **Weighted** and **DWeighted** still have coverage slightly smaller than $1.0$, but **Uniform** does significantly worse. Using weighted selection is useful when $L$ is small.

**Random** improves coverage with increasing $L$ because the size of the cores increases. Note that, to reach the same value of coverage obtained by the other heuristics, **Random** requires a large core size of $9$.

There are two other important observations to make about this graph. First, coverage is roughly the same for **Uniform**, **Weighted**, and **DWeighted** when $L > 2$. Second, as $L$ continues to increase, there is a small decrease in coverage. This is due to the nature of our traces and to the random choices made by our algorithms. Ports such as 111 (portmapper, rpcbind) and 22 (sshd) are open on several of the hosts with operating systems different than Windows. For small values of $L$, these hosts rapidly reach their threshold. Consequently, when hosts that do have these services as attributes request a core, there are fewer hosts available with these same attributes. On the other hand, for larger values of $L$, these hosts are more available, thus slightly increasing the probability that not all the attributes are covered for hosts executing an operating system different than Windows. We observed this phenomenon exactly with ports 22 and 111 in our traces.

This same phenomenon can be observed in Figure 6.7. In this figure, we plot the average fraction of hosts that are not fully covered, which is an alternative way of visualizing coverage. We observe that there is a share of the population of hosts that are not fully covered, but this share is very small for **Uniform** and its variations. Such a set is likely to exist due to the non-deterministic choices we make in our heuristics when forming cores. These uncovered hosts, however, are not fully unprotected. From our simulation traces, we note the average number of uncovered attributes is very small for **Uniform** and its variations. In all runs, we have just a few hosts that do not have all their attributes covered, and in the majority of the instances there is just a single uncovered attribute.

Finally, we show the resulting variance in load. Since the heuristics limit each host to be in no more than $L$ cores, the maximum load equals $L$. The variance indicates how fairly the load is spread among the hosts. As expected, **Random** does well, having the lowest variance among all the algorithms and for all values of $L$. Ordering the greedy

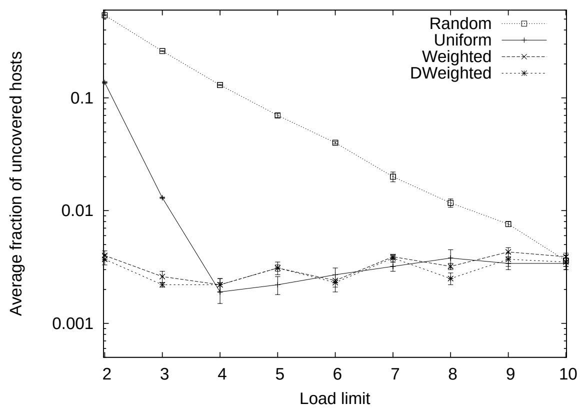

**Figure 6.7:** Average fraction of uncovered hosts

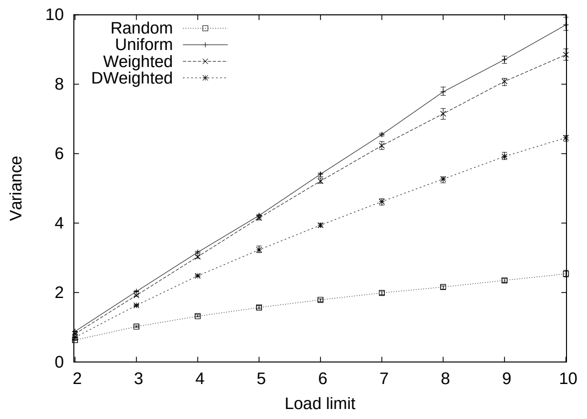

**Figure 6.8:** Average load variance

heuristics by their variance in load, we have **Uniform** $\succ$ **Weighted** $\succ$ **DWeighted**. This is not surprising since we introduced the weighted selection exactly to better balance the load. It is interesting to observe that for every value of $L$, the load variance obtained for **Uniform** is close to $L$. This means that there were several hosts not participating in any core and several other hosts participating in $L$ cores.

A larger variance in load may not be objectionable in practice as long as a maximum load is enforced. Given the extra work of maintaining the functions $N_s$ and $N_c$, the heuristic **Uniform** with small $L$ ($L > 2$) is the best choice for our application. However, should load variance be an issue, we can use one of the other heuristics.

### 6.4.3 Translating to real pathogens

In this section, we discuss why we have chosen to tolerate exploits of vulnerabilities on a single attribute at a time. We do so based on information about past worms to support our choices and assumptions.

Worms such as the ones in Table 6.1 used services that have vulnerabilities as vectors for propagation. Code Red, for example, used a vulnerability in the IIS Web server to infect hosts. In this example, a vulnerability on a single attribute (Web server listening on port 80) was exploited. In other instances, such as with the Nimda worm, more than one vulnerability was exploited during propagation, such as via e-mail messages and Web browsing. Although these cases could be modeled as exploits to vulnerabilities on multiple attributes, we observe that previous worms did not propagate across operating system platforms: in fact, the worms targeted services on various versions of Windows.

By covering classes of operating systems in our cores, we guarantee that pathogens that exploit vulnerabilities on a single platform are not able to compromise all the members of a core $C$ of a particular host $h$, assuming that $C$ covers all attributes of $h$. Even if $\mathit{Core}(h)$ leaves some attributes uncovered, $h$ is still protected against attacks targeting covered attributes. Referring back to Figure 6.7, the majority of the cores have maximum coverage. We also observed in the previous section that, for cores that do not have maximum coverage, usually it is only a single uncovered attribute.

Under our assumptions, informed replication mitigates the effects of a worm that exploits vulnerabilities on a service that exists across multiple operating systems, and of a worm that exploits vulnerabilities on services in a single operating system. Figure 6.7 presents a conservative estimate on the percentage of the population that is unprotected in the case of an outbreak of such a pathogen. Assuming conservatively that every host that is not fully covered has the same uncovered attribute, the numbers in the graph give the fraction of the population that can be affected in the case of an outbreak. As can be seen, this fraction is very small.

With our current use of attributes to represent software heterogeneity, a worm can be effective only if it can exploit vulnerabilities in services that run across operating systems, or if it exploits vulnerabilities in multiple operating systems. To the best of our knowledge, there has been no large-scale outbreak of such a worm. Of course, such a worm could be written. In the next section, we discuss how to modify our heuristics to cope with exploits of vulnerabilities on multiple attributes.

### 6.4.4 Exploits of multiple attributes

To tolerate exploits of multiple attributes, we need to construct cores such that, for subsets of attributes possessed by members of a core, there must be a core member that does not have these attributes. We call a $k$*-resilient core* $C$ a group of hosts in $\mathcal{H}$ such that, for every $k$ attributes of members of $C$, there is at least one host in $C$ that does not contain any of these attributes. In this terminology, the cores we have been considering up to this point have been 1-resilient cores.

To illustrate this idea, consider the following example. Hosts run *Windows*, *Linux*, and *Solaris* as operating systems, and *IIS*, *Apache*, and *Zeus* as Web servers. An example of a 2-resilient core is a subset composed of hosts $h_1, h_2, h_3$ with configurations: $h_1 = \{\text{Linux, Apache}\}$; $h_2 = \{\text{Windows, IIS}\}$; $h_3 = \{\text{Solaris, Zeus}\}$. In this core, for every pair of attributes, there is at least one host that contains none of them.

As before, every host $h$ builds a $k$-resilient core *Core*$(h)$. To build *Core*$(h)$, host $h$ uses the following heuristic:

**Step 1** Select randomly $k-1$ hosts, $h_1$ through $h_{k-1}$, such that $h_i.os \neq h.os$, for every $i \in \{1, \ldots, k-1\}$;

**Step 2** Use **Uniform** to search for a 1-resilient core $C$ for $h$;

**Step 3** For each $i \in \{1, \ldots, k-1\}$, use **Uniform** to search for a 1-resilient core $C_i$ for $h_i$;

**Step 4** *Core*$(h) \leftarrow C \cup C_1 \cup \ldots \cup C_{k-1}$.

Intuitively, to form a $k$-resilient core we need to gather enough hosts such that we can split these hosts into $k$ subsets, where at least one subset is a 1-resilient core. Moreover, if there are two of these subsets where, for each subset, all of the members

**Table 6.4:** Summary of simulation results for $k = 2$ for $8$ different runs

| $L$ | **Avg. 2–coverage** | **Avg. 1–coverage** | **Avg. Core size** |
|---|---|---|---|
| 5 | 0.829 (0.002) | 0.855 (0.002) | 4.19 (0.004) |
| 6 | 0.902 (0.002) | 0.917 (0.002) | 4.59 (0.005) |
| 7 | 0.981 (0.001) | 0.987 (0.001) | 5.00 (0.005) |
| 8 | 0.995 (0.0) | 1.0 (0.0) | 5.11 (0.005) |
| 9 | 0.996 (0.0) | 1.0 (0.0) | 5.14 (0.005) |
| 10 | 0.997 (0.0) | 1.0 (0.0) | 5.17 (0.003) |

of that subset share some attribute, then the shared attribute of one set must be different from the shared attribute of the other set. Our heuristic is conservative in searching independently for 1-resilient cores because the problem does not require all such sets to be 1-resilient cores. In doing so, we protect clients and at the same time avoid the complexity of optimally determining such sets. The sets output by the heuristic, however, may not be minimal, and therefore they are approximations of theoretical cores.

In Table 6.4, we show simulation results for this heuristic for $k = 2$. The first column shows the values of load limit ($L$) used by the **Uniform** heuristic to compute cores. We chose values of $L \geq 5$ based on an argument generalized from the one given in Section 6.4.2 giving the lower bound of $L$. In the second and third columns, we present our measurements for coverage with standard error in parentheses. For each computed core $\mathit{Core}(h)$, we calculate the fraction of pairs of attributes such that at least one host $h' \in \mathit{Core}(h)$ contains none of the attributes of the pair. We name this metric **2-coverage**, and in the table we present the average across all hosts and across all eight runs of the simulator. **1-coverage** is the same as the average coverage metric defined in Section 6.4.2. Finally, the last column shows average core size.

According to the coverage results, the heuristic does well in finding cores that protect hosts against potential pathogens that exploit vulnerabilities in at most two attributes. A beneficial side-effect of protecting against exploits on two attributes is that the amount of diversity in a 2-resilient core permits better protection to its client against pathogens that exploit vulnerabilities on single attributes. For values of $L$ greater than seven, all clients have all their attributes covered (the average 1-coverage metric is one and the standard error is zero).

Having a system that more broadly protects its hosts requires more resources: core sizes are larger to obtain sufficiently high degrees of coverage. Compared to the results in Section 6.4.2, we observe that we need to double the load limit to obtain similar values for coverage. This is not surprising. In our heuristic, for each host, we search for two 1-resilient cores. We therefore need to roughly double the amount of resources used.

Of course, there is a limit to what can be done with informed replication. As $k$ increases, the demand on resources continues to grow, and a point will be reached in which there is not enough diversity to withstand an attack that targets $k + 1$ attributes. Using our diversity study results in Table 6.2, if a worm were able to simultaneously infect machines that run one of the first four operating systems in this table, the worm could potentially infect $84\%$ of the population. The release of such a worm would most likely cause the Internet to collapse. An approach beyond informed replication would be needed to combat an act of cyberterrorism of this magnitude.

## 6.5 The Phoenix Recovery System

A cooperative recovery service is an attractive architecture for tolerating Internet catastrophes. It is attractive for both individual Internet users, like home broadband users, who do not wish to pay for commercial backup service or deal with the inconvenience of making manual backups, as well as corporate environments, which often have a significant amount of unused disk space per machine. If Phoenix were deployed, users would not need to exert significant effort to backup their data, and they would not require local backup systems. Phoenix makes specifying what data to protect as straightforward as specifying what data to share on file-sharing peer-to-peer systems. Further, a cooperative architecture has little cost in terms of time and money; instead, users relinquish

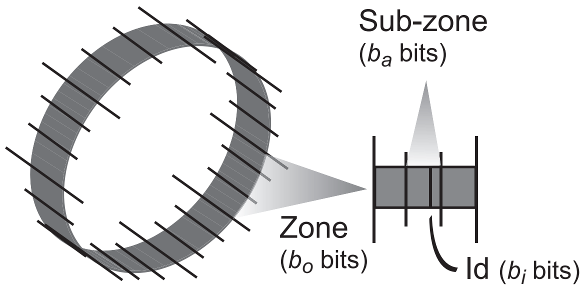

**Figure 6.9:** Phoenix ring

a small fraction of their disk, CPU, and network resources to gain access to a highly resilient backup service.

As with Pastiche [CN02], we envision using Phoenix as a cooperative recovery service for user data. However, rather than exploiting redundant data on similar hosts to reduce backup costs for operating system and application software, we envision Phoenix users only backing up user-generated data and relying upon installation media to recover the operating system and application software. With this usage model, broadband users of Phoenix can recover 10 GB of user-generated data in a day. Given the relatively low capacity utilization of disks in desktop machines [BJZH04], 10 GB should be sufficient for a wide range of users. Further, users can choose to be more selective in the data backed up to reduce their recovery time. We return to the issue of bandwidth consumption and recovery time in Section 6.6.3.

### 6.5.1 System overview

A Phoenix host selects a subset of hosts to store backup data, expecting that at least one host in the subset survives an Internet catastrophe. This subset is a core, chosen using the **Uniform** heuristic described in Section 6.4.

Choosing cores requires knowledge of host software configurations. As described in Section 6.4, we use the container mechanism for advertising configurations. In our prototype, we implement containers using the Pastry [RD01] distributed hash table (DHT). Pastry is an overlay of nodes that have identifiers arranged in a ring. This overlay provides a scalable mechanism for routing requests to appropriate nodes.

Phoenix structures the DHT identifier space hierarchically. It splits the identifier space into *zones*, mapping containers to zones. It further splits zones into *sub-zones*, mapping sub-containers to equally-sized sub-zones. Figure 6.9 illustrates this hierarchy. Corresponding to the hierarchy, Phoenix creates host identifiers out of three parts. To generate its identifier, a host concatenates the hash representing its operating system $h.os$, the hash representing an attribute $a \in h.apps$, and the hash representing its IP address. As Figure 6.9 illustrates, each part has $b_o$, $b_a$, and $b_i$ bits, respectively. To advertise its configuration, a host creates a hash for each one of its attributes. It therefore generates as many identifiers as the number of attributes in $h.apps$. It then joins the DHT at multiple points, each point being characterized by one of these identifiers. Since the hash of the operating system is the initial, "most significant" part of all the host's identifiers, all identifiers of a host lie within the same zone.

To build *Core*($h$) using **Uniform**, host $h$ selects hosts at random. When trying to cover an attribute $a$, $h$ first selects a container at random, which corresponds to choosing a number $c$ randomly from $[0, 2^{b_o} - 1]$. The next step is to select a sub-container and a host within this sub-container both at random. This corresponds to choosing a random number $sc$ within $[0, 2^{b_a} - 1]$ and another random number $id$ within $[0, 2^{b_i} - 1]$, respectively. Host $h$ creates a Phoenix identifier by concatenating these various components as $(c \circ sc \circ id)$. It then performs a lookup on the Pastry DHT for this identifier. The host $h'$ that satisfies this lookup informs $h$ of its own configuration. If this configuration covers attribute $a$, $h$ adds $h'$ to its core. If not, $h$ repeats this process.

The hosts in $h$'s core maintain backups of its data. These hosts periodically send announcements to $h$. In the event of a catastrophe, if $h$ loses its data, it waits for one of these announcements from a host in its core, say $h'$. After receiving such a message, $h$ requests its data from $h'$. Since recovery is not time-critical, the period between consecutive announcements that a host sends can be large, from hours to a day.

A host may permanently leave the system after having backed up its files. In this situation, other hosts need not hold any backups for this host and can use garbage collection to retrieve storage used for the departed host's files. Thus, Phoenix hosts assume that if they do not receive an acknowledgment for any announcement sent for a large period of time (*e.g.*, a week), then this host has left the system and its files can be discarded.

Since many hosts share the same operating systems, Phoenix identifiers are not mapped in a completely random fashion into the DHT identifier space. This could lead to some hosts receiving a disproportionate number of requests. For example, consider a host $h$ that is either the first of a populated zone that follows an empty zone or is the last host of a populated zone that precedes an empty zone. Host $h$ receives requests sent to the empty zone because, by the construction of the ring, its address space includes addresses of the empty zone. In our design, however, once a host reaches its load limit, it can simply discard new requests by the Phoenix protocol.

Experimenting with the Phoenix prototype, we found that constructing cores performed well even with an unbalanced ID space. But a simple optimization can improve core construction further. The system can maintain an OS hint list that contains canonical names of operating systems represented in the system. When constructing a core, a host then uses hashes of these names instead of generating a random number. Such a list could be maintained externally or generated by sampling. We present results for both approaches in Section 6.6.

### 6.5.2 Service design

The Phoenix software a host runs is composed of two mechanisms, an *agent* and a *server*. The Phoenix agent is responsible for interacting with a user application on top of it and with a Pastry agent underneath. Figure 6.10 is a state machine description of the behavior of the Phoenix agent. The agent begins in the `Init` state, and changes to `Joining` when the user application requests it to join the Phoenix ring. In the `Joining` state, it creates a session for each Phoenix address of the host. Each of these sessions has its own routing table and leaf set, and thus participates in the DHT as an independent Pastry agent. After joining all the sessions, a Phoenix agent change its state

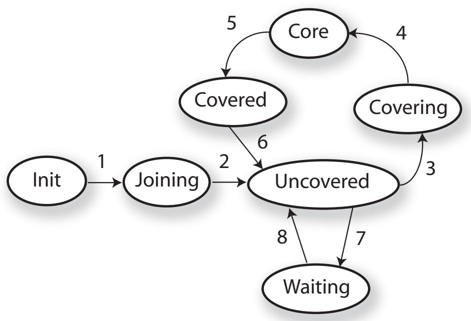

**Figure 6.10:** State machine for a Phoenix agent

to `Uncovered`. At this point, the agent requires input from the user application. If the application specifies that the host needs a core to backup data, the agent undergoes transition “3”, changing to state to `Covering`. If, on the other hand, the application specifies that the host has lost data, it generates a request that causes the agent to use transition “7”, changing its state to `Waiting`.

In state `Covering`, the agent uses heuristic **Uniform** to select a core. Note that containers and sub-containers in the original specification of **Uniform** map to zones and sub-zones, respectively. After selecting a core, the agent notifies the application and changes its state to `Core`. The application then has to decide if the core satisfies its expectations, or if the agent should try again. If it decides to accept the core, then the agent sends `Data` messages containing the data to be backed up to the Phoenix servers of the core components. When the data backup completes on all the hosts in the core, the host transitions to state `Covered`.

If, while in state `Uncovered`, the Phoenix agent undergoes a transition to `Waiting`, then it waits until it receives an announcement from a host in its core. Hosts holding data on behalf of other hosts send these messages periodically so that hosts learn of the members of their core in the event of a catastrophe. Upon reception of an announcement, a host in state `Waiting` sends a request to the core member that made the announcement to restore its data. The Phoenix server of the core member receives the request and replies with the content requested.

The two main responsibilities of the Phoenix server of a host are managing storage, and sending announcements and responding to requests to restore data. When a Phoenix server of a host $h$ participates in a core, it commits to store the data of the requester. If it receives data from the requester, then $h$ stores this data, and starts sending announcements to this host. The requester, however, may decide not to include $h$ in its core. In this case, the requester may ignore the reply or send a release message. If the requester sends a release message, then $h$ removes this host from its list of served clients. Otherwise, the acceptance eventually times out, and $h$ again rejects data from the requester. Release messages also serve the purpose of releasing data stored on core members. This happens in the case that a user decides to select another core or if a core partially fails.

Since recovery is not time-critical, the period between consecutive announcements sent to the same host can be relatively large, from hours to a day. For this reason, we assume that hosts send such announcements once a day, although the parameter is configurable and can be changed according to the demands on the system. These messages are acknowledged by the receiver. Note that not getting a reply to an announcement does not necessarily mean that the host left the system ungracefully, since the particular host that did not reply might have failed. At the same time, it is necessary to garbage collect backups of hosts that are not part of the system anymore. For this reason, we assume that if a host $h$ does not receive a reply to announcement messages it sent to $h'$ within a large period of time, say a week, then it garbage collects the data it holds on behalf of $h'$. A week should be sufficient time for users to notice that they lost their data and request a restore.

We now turn our attention to the protocol used by Phoenix to communicate among servers. Phoenix implements this protocol using the following message types:

- `request`: requests participation in a core;
- `reply`: in response to a request to participate in a core, a host $h$ replies indicating whether it agrees to participate or not. This decision depends on the number of other hosts already being serviced by this host. If $h$ decides to accept, then it sends

its own configuration along with the reply message;

- `release`: if a host $h'$ decides not to use $h$ as a core member for its data, it sends this message to $h$ so that $h$ is notified that it is not a core member for $h'$.
- `announcement`: a host $h$ periodically sends this message to host $h'$ if $h$ is in $h'$s core and stores a copy of $h'$s data;
- `data`: a host $h$ sends this message containing its data to be backed up to a host $h'$ if $h'$ has agreed to participate in the core constructed by $h$;
- `request_restore`: after a catastrophe, a host sends this message as soon as it discovers a member of its core storing its data, *i.e.*, as soon as it receives an `announcement` message;
- `restore`: once a host receives a `restore_request`, it replies with the data it stored on behalf of the requesting host.

### 6.5.3 Phoenix application

The Phoenix application is responsible for all the interaction with a Phoenix user. To start operating, this application needs to determine the data to be backed up, the amount of storage available for other hosts, and the configuration of the current host. Currently, the input to the application is composed of a tar file of data to be backed up and a host configuration. Although in our prototype users manually specify the host configuration, we believe it is necessary to add procedures for extracting the host configuration automatically. The idea for these procedures, however, is not to eliminate the user from the process of deciding the attributes of the host. Instead, the goal is provide the user with hints of what the attributes should be. We expect that, from a practical point of view, a user will want to have some say in which attributes are important. Designing such procedures for determining the attributes of a host is part of future work.

### 6.5.4 Attacks on Phoenix

Phoenix uses informed replication to survive wide-spread failures due to exploits of vulnerabilities in unrelated software on hosts. However, Phoenix itself can also be the target of attacks mounted against the system, as well as attacks from within by misbehaving peers.

The most effective way to attack the Phoenix system as a whole is to unleash a pathogen that exploits a vulnerability in the Phoenix software. In other words, Phoenix itself represents a shared vulnerability for all hosts running the service. This shared vulnerability is not a covered attribute, hence an attack that exploits a vulnerability in the Phoenix software would make it possible for data to be lost as a pathogen spreads unchecked through the Phoenix system. To the extent possible, Phoenix relies on good programming practices and techniques to prevent common attacks such as buffer overflows. However, this kind of attack is not unique to Phoenix or the use of informed replication. Such an attack is a general problem for any distributed system designed to protect data, even those that use approaches other than informed replication [HMD05]. A single system fundamentally represents a shared vulnerability; if an attacker can exploit a vulnerability in system software and compromise the system, the system cannot easily protect itself.

Alternatively, hosts participating in Phoenix can attack the system by trying to access private data, tamper with data, or mount denial-of-service attacks. To prevent malicious servers from accessing data without authorization or from tampering with data, we can use standard cryptographic techniques [JBH$^+$05]. In particular, we can guarantee the following: (1) the privacy and integrity of any data saved by any host is preserved, and (2) if a client host contacts an honest server host for a backup operation, then the client is able to recover its data after a catastrophe. From a security perspective, the most relevant part of the system is the interaction process between a host client and a host server which has agreed to participate in the host's core.

Malicious servers can mount a denial-of-service attack against a client by agreeing to hold a replica copy of the client's data, and subsequently dropping the data or refusing recovery requests. One technique to identify such misbehavers is to issue *signed receipts* [JBH$^+$05]. Clients can use such receipts to claim that servers are misbehaving. As we mentioned before, servers cannot corrupt data assuming robustness of the security primitives.

Hosts could also advertise false configurations in an attempt to free-ride in the system. By advertising attributes that make a host appear more unreliable, the system will consider the host for fewer cores than otherwise. As a result, a host may be able to have its data backed up without having to back up its share of data.

To provide a disincentive against free-riders, members of a core can maintain the configuration of hosts they serve, and serve a particular client only if their own configuration covers at least one client attribute. By sampling servers randomly, it is possible to reconstruct cores and eventually find misbehaving clients.

An important feature of our heuristic that constrains the impact of malicious hosts on the system is the load limit: if only a small percentage of hosts is malicious at any given time, then only a small fraction of hosts are impacted by the maliciousness. Hosts not respecting the limit can also be detected by random sampling.

## 6.6 Phoenix evaluation

In this section, we evaluate our Phoenix prototype on the PlanetLab testbed using the metrics discussed in Section 6.4. We also simulate a catastrophic event — the simultaneous failure of all Windows hosts — to experiment with Phoenix's ability to recover from large failures. Finally, we discuss the time and bandwidth required to recover from catastrophes.

### 6.6.1 Prototype evaluation

We tested our prototype on 63 hosts across the Internet: 62 PlanetLab hosts and one UCSD host. To simulate the diversity we obtained in the study presented in

Section 6.3, we selected 63 configurations at random from our set of 2,963 configurations of general-purpose hosts, and made each of these configurations an input to the Phoenix service on a host. In the population we have chosen randomly, out of the 63 configurations 38 have Windows as their operating system. Thus, in our setting roughly 60% of the hosts represent Windows hosts. From Section 6.4.2, the load limit has to be at least three.

For the results we present in this section, we use an OS hint list while searching for cores. Varying $L$, we obtained the values in Table 6.5 for coverage, core size, and load variance for a representative run of our prototype. For comparison, we also present results from our simulations with the same set of configurations used for the PlanetLab experiment. From the results in the table, coverage is perfect in all cases, and the average core size is less than 3 (less than 2 replica copies).

The major difference in increasing the value of $L$ is the respective increase in load variance. As $L$ increases, load balance worsens. We also counted the number of requests issued by each host in its search for a core. Different from our simulations, we set a large upper bound on the number of request messages (`diff_OS + same_OS` $= 100$) to verify the average number of requests necessary to build a core, and we had hosts searching for other hosts only outside their own zones (`same_OS` = 0). The averages for number of requests are $14.6$, $5.2$, and $4.1$ for values of $L$ of $3$, $5$, and $7$, respectively. Hence, we can tradeoff load balance and message complexity.

We also ran experiments without using an OS hint list. The results are very good, although worse than the implementation that uses hint lists. We observed two main consequences in not using a hint list. First, the average number of requests is considerably higher (over 2x). Second, for small values of $L$ ($L = 3, 5$), some hosts did not obtain perfect coverage.

### 6.6.2 Simulating catastrophes

Next we examine how the Phoenix prototype behaves in a severe catastrophe: the exploitation and failure of all Windows hosts in the system. This scenario corre-

**Table 6.5:** Implementation results on PlanetLab (“Imp”) with simulation results for comparison (“Sim”)

| **Load limit (L)** | **Core size** | | **Coverage** | | **Load var.** | |
|---|---|---|---|---|---|---|
| | Imp. | Sim. | Imp. | Sim. | Imp. | Sim. |
| 3 | 2.12 | 2.23 | 1.0 | 1.0 | 1.65 | 1.88 |
| 5 | 2.10 | 2.25 | 1.0 | 1.0 | 2.88 | 3.31 |
| 7 | 2.10 | 2.12 | 1.0 | 1.0 | 4.44 | 3.56 |

sponds to a situation in which a worm exploits a vulnerability present in *all* versions of Windows, and corrupts the data on the compromised hosts. Note that this scenario is far more catastrophic than what we have experienced with worms to date. The worms listed in Table 6.1, for example, exploit only particular services on Windows.

The simulation proceeded as follows. Using the same experimental setting as above, hosts backed up their data under a load limit constraint of $L = 3$. We then triggered a failure in all Windows hosts, causing the loss of data stored on them. Next we restarted the Phoenix service on the hosts, causing them to wait for announcements from other hosts in their cores (Section 6.5.1). We then observed which Windows hosts received announcements and successfully recovered their data.

All 38 hosts recovered their data in a reasonable amount of time. For 35 of these hosts, it took on average 100 seconds to recover their data. For the other three machines, it took several minutes due to intermittent network connectivity (these machines were in fact at the same site). Two important parameters that determine the time for a host to recover are the frequency of announcements and the backup file size (transfer time). We used an interval between two consecutive announcements to the same client of 120 seconds, and a total data size of 5 MB per host. The announcement frequency depends on the user expectation on recovery speed. In our case, we wanted to finish each experiment in a reasonable amount of time. Yet, we did not want to have hosts sending a large number of announcement messages unnecessarily. For the backup file size, we chose an arbitrary value since we are not concerned about transfer time in this experiment. On the other hand, this size was large enough to hinder recovery when connectivity between client and server was intermittent.

It is important to observe that we stressed our prototype by causing the failure of these hosts almost simultaneously. Although the number of nodes we used is small compared to the potential number of nodes that Phoenix can have as participants, we did not observe any obvious scalability problems. On the contrary, the use of a load limit helped in constraining the amount of work a host does for the system, independent of system size.

### 6.6.3 Recovering from a catastrophe

We now examine the bandwidth requirements for recovering from an Internet catastrophe. In a catastrophe, many hosts will lose their data. When the failed hosts come online again, they will want to recover their data from the remaining hosts that survived the catastrophe. With a large fraction of the hosts recovering simultaneously, a key question is what bandwidth demands the recovering hosts will place on the system.

The aggregate bandwidth required to recover from a catastrophe is a function of the amount of data stored by the failed hosts, the time window for recovery, and the fraction of hosts that fail. Consider a system of 10,000 hosts that have software configurations analogous to those presented in Section 6.3, where $54.1\%$ of the hosts run Windows and the remaining run some other operating system. Next consider a catastrophe similar to the one above in which all Windows hosts, independent of version, lose the data they store. Table 6.6 shows the bandwidth required to recover the Windows hosts for various storage capacities and recovery periods. The first column shows the average amount of data a host stores in the system. The remaining columns show the bandwidth required to recover that data for different periods.

The first four rows show the aggregate system bandwidth required to recover the failed hosts: the total amount of data to recover divided by the recovery time. This bandwidth reflects the load on the network during recovery. Assuming a deployment over the Internet, even for relatively large backup sizes and short recovery periods, this load is small. Note that these results are for a system with 10,000 hosts and that, for an

**Table 6.6:** Bandwidth consumption after a catastrophe

| Size (GB) | 1 hour | 1 day | 1 week |
|---|---|---|---|
| Aggregate bandwidth | | | |
| 0.1 | 1.2 Gb/s | 50 Mb/s | 7.1 Mb/s |
| 1 | 12 Gb/s | 0.50 Gb/s | 71 Mb/s |
| 10 | 120 Gb/s | 5.0 Gb/s | 710 Mb/s |
| 100 | 1.2 Tb/s | 50 Gb/s | 7.1 Gb/s |
| Per-host bandwidth ($L = 3$) | | | |
| 0.1 | 0.7 Mb/s | 28 Kb/s | 4.0 Kb/s |
| 1 | 6.7 Mb/s | 280 Kb/s | 40 Kb/s |
| 10 | 66.7 Mb/s | 2.8 Mb/s | 400 Kb/s |
| 100 | 667 Mb/s | 28 Mb/s | 4.0 Mb/s |

equivalent catastrophe, the aggregate bandwidth requirements will scale linearly with the number of hosts in the system and the amount of data backed up.

The second four rows show the average per-host bandwidth required by the hosts in the system responding to recovery requests. Recall that the system imposes a load limit $L$ that caps the number of replicas any host will store. As a result, a host recovers at most $L$ other hosts. Note that, because of the load limit, per-host bandwidth requirements for hosts involved in recovery are independent of both the number of hosts in the system and the number of hosts that fail.

The results in the table show the per-host bandwidth requirements with a load limit $L = 3$, where each host responds to at most three recovery requests. The results indicate that Phoenix can recover from a severe catastrophe in reasonable time periods for useful backup sizes. As with other cooperative backup systems like Pastiche [CN02], per-host recovery time will depend significantly on the connectivity of hosts in the system. For example, hosts connected by modems can serve as recovery hosts for a modest amount of backed up data (28 Kb/s for 100 MB of data recovered in a day). Such backup amounts would only be useful for recovering particularly critical data, or recovering frequent incremental backups stored in Phoenix relative to infrequent full backups using other methods (*e.g.*, for users who take monthly full backups on media but use Phoenix for storing and recovering daily incrementals). Broadband hosts can recover failed hosts storing orders of magnitude more data (1–10 GB) in a day, and high-bandwidth hosts can recover either an order magnitude more quickly (hours) or even an order of magnitude more data (100 GB). Further, Phoenix could potentially exploit the parallelism of recovering from all surviving hosts in a core to further reduce recovery time.

Although there is no design constraint on the amount of data hosts back up on Phoenix, for current disk usage patterns, disk capacities, and host bandwidth connectivity, we envision users typically storing 1–10 GB in Phoenix and waiting a day to recover their data. According to a recent study, desktops with substantial disks ($> 40$ GB) use less than 10% of their local disk capacity, and operating system and temporary user files consume up to 4 GB [BJZH04]. Recovery times on the order of a day are also practical. For example, previous worm catastrophes took longer than a day for organizations to recover, and recovery using organization backup services can take a day for an administrator to respond to a request.

## 6.7 The complexity of finding cores

In this section, we discuss the complexity of searching for cores in a set of hosts. The goal is to show that the problem of selecting optimally cores (cores that contain the fewest possible number of hosts) is intractable even when hosts know all the other hosts in the system and their configurations. Throughout this section, we first show that the problem of selecting one core is NP-complete. Then we show that selecting optimally a $k+1$-resilient core is at least as hard as selecting a $k$-resilient core. We begin by providing a few important definitions.

### 6.7.1 Definitions

Recall that, informally, a core is a subset of hosts that is diverse enough for a given task. What "enough" means depends on the application. In the case of the Phoenix system, the members of a core must have sufficiently different software configurations so that not all members share a non-empty set of exploitable vulnerabilities.

We show that finding such cores is an intractable problem. Let the definition of a system be as follows:

**Definition 6.7.1** A system is a triple $\langle \mathcal{H}, \mathcal{A}, \alpha \rangle$, where $\mathcal{H}$ is a finite set of hosts, $\mathcal{A}$ is a finite set of attributes, and $\alpha$ a mapping from hosts to attributes ($\alpha : h \in \mathcal{H} \rightarrow A \subseteq \mathcal{A}$).

We define a $k$-resilient core as follows:

**Definition 6.7.2** Given a system $\langle \mathcal{H}, \mathcal{A}, \alpha \rangle$, a set of hosts $C \subseteq \mathcal{H}$ is a $k$-resilient core, $k > 0 \wedge k \in \mathbb{N}$, if and only if there is no $A \subseteq \mathcal{A}$, $|A| \leq k$, such that for every $h_c \in C$, $\alpha(h_c) \cap A \neq \emptyset$. Such a subset must be also minimal: $\forall h_c \in C, \exists a \in \alpha(h) : \forall h'_c \in C \setminus \{h_c\}, a \notin \alpha(h'_c)$.

Recall from Section 6.4 that a core serves a particular host $h$ and it is constrained to contain $h$. We hence make the same assumption here with the following definition:

**Definition 6.7.3** A $k$-resilient core $C$ is a $k$-resilient core for $h \in \mathcal{H}$ if and only if $h \in C$.

We now present two important problems for the purposes of this section. The Set Cover problem is a well-known NP-complete problem, often used in mapping reductions [GJ79]. We repeat its definition here for the sake of clarity:

**Problem** : SC decision problem

**Instance** : Collection $\mathcal{T}$ of subsets of a finite set $U$, positive integer $k \leq |\mathcal{T}|$;

**Question** : Does $\mathcal{T}$ contain a cover for $U$ of size at most $k$?

Now the problem we are interested in:

**Problem** : $k$-Core decision problem

**Instance** : A system $\langle \mathcal{H}, \mathcal{A}, \alpha \rangle$, a host $h \in \mathcal{H}$, a positive integer $s > 0$;

**Question** : Is there a $k$-resilient core $C \subseteq \mathcal{H}$ for $h$ of size at most $s$?

### 6.7.2 NP-completeness of 1-Core

We show with the following two claims that the 1-Core decision problem is NP-complete. By doing so, we later argue in this section that there cannot be a polynomial-time algorithm that outputs a minimal 1-resilient core, unless P = NP. In other words, the correspondent search problem cannot be easier to solve than the decision problem. In the next section, we discuss in more detail the problem of searching for a minimal $k$-resilient core for a value of $k$ greater than one. Intuitively, such a problem is at least as hard as searching for a minimal 1-resilient core. As such, if there is no polynomial-time algorithm that outputs a minimal 1-resilient core given a system $\langle \mathcal{H}, \mathcal{A}, \alpha \rangle$, there cannot be a polynomial-time algorithm that outputs a minimal $k$-resilient core given a system $\langle \mathcal{H}, \mathcal{A}, \alpha \rangle$, for values of $k$ greater than one.

**Claim 6.7.4** *SC* $\leq_m$ 1-*Core*

**Proof:**
We need to provide a polynomial-time algorithm that, given an instance $\langle U, \mathcal{T}, k \rangle$ of the SC problem, returns an instance $\langle \langle \mathcal{H}, \mathcal{A}, \alpha \rangle, h, s \rangle$ of the 1-Core Problem, such that the following holds:

i. If $\langle U, \mathcal{T}, k \rangle \in$ SC, then $\langle \langle \mathcal{H}, \mathcal{A}, \alpha \rangle, h, s \rangle \in$ 1-Core;

ii. If $\langle \langle \mathcal{H}, \mathcal{A}, \alpha \rangle, h, s \rangle \in$ 1-Core, then $\langle U, \mathcal{T}, k \rangle \in$ SC.

An algorithm that achieves this goal is the following:

**Algorithm SCtoC**: $\langle U, \mathcal{T}, k\rangle$

```
    $\mathcal{A} \leftarrow \emptyset$; $\mathcal{H} \leftarrow \{h\}$;
    $A \leftarrow \emptyset$;
    For every element $u$ of $U$,
        $\mathcal{A} \leftarrow \mathcal{A} \cup \{a_u, \widehat{a_u}\}$;
        $A \leftarrow A \cup \{a_u\}$;
    $\alpha \leftarrow \alpha \cup [h \rightarrow A]$;
    For every element $T$ of $\mathcal{T}$,
        $\mathcal{H} \leftarrow \mathcal{H} \cup \{h_T\}$;
        $A \leftarrow \emptyset$;
        $\forall a_u \in A$, if ($u \in T$) then $A \leftarrow A \cup \{\widehat{a_u}\}$;
                                     else $A \leftarrow A \cup \{a_u\}$;
        $\alpha \leftarrow \alpha \cup [h_T \rightarrow A]$;
    $s \leftarrow k$;
```

Every step of the algorithm runs in polynomial time. Consequently, time complexity is given by the sum of the complexities of the individual steps. This is clearly polynomial.

It remains to show that Properties (i) and (ii) hold for **SCtoC**. First we show (i). For an instance $\langle U, \mathcal{T}, k\rangle$ of the SC problem, suppose there is a subset $\mathcal{T}'$ of $\mathcal{T}$ such that $|\mathcal{T}'| \leq k$ and $\mathcal{T}'$ is a cover for $U$. We construct a 1-resilient core $\Omega$ for the instance of the Core Problem returned by our algorithm as follows:

1. $\forall T \in \mathcal{T}' : \Omega \leftarrow \Omega \cup \{h_T\}$;

2. $\Omega \leftarrow \Omega \cup \{h\}$.

By construction, for every attribute $a \in \alpha(h)$, there is in $\Omega$ at least one host $h_T$ such that $a \notin \alpha(h_T)$. According to the description of **SCtoC**, a host $h_T$ only covers an attribute $a_u$ of $h$ if $u \in T$. Because $\mathcal{T}'$ is a cover for $U$, $\Omega$ must be a 1-resilient core for $h$. Moreover, $\Omega$ must have size at most $s = k$.

Now we show ii). Given an instance $\langle\langle\mathcal{H},\mathcal{A},\alpha\rangle,h,s\rangle$ of the 1-Core problem, suppose there is a 1-resilient core $\Omega$ for $h$ of size at most $s$. From the definition of a core, we have that a host $h_T$ is in $\Omega$ only if it covers at least one attribute of $h$. By the construction of **SCtoC**, if a host $h_T$ covers an attribute $a_u$ of $h$, then $u \in T$. Thus, we can construct a cover $\mathcal{T}'$ for $U$ by including in $\mathcal{T}'$ all the sets $T \in \mathcal{T}$ such that $h_T \in \Omega$. Again by construction, $\mathcal{T}'$ must cover $U$, and $|\mathcal{T}'| \leq k$. This completes our proof.

□

Now we show that 1-Core is in NP.

**Claim 6.7.5** 1-*Core* $\in$ *NP.*

**Proof:**

We need to provide a polynomial-time verifier for 1-Core. The verifier takes as input an instance $\langle\langle\mathcal{H},\mathcal{A},\alpha\rangle,h,s\rangle$ of the 1-Core problem and a certificate *Cert*. This certificate consists of a subset of $\mathcal{H}$. Thus, the verifier has to check whether the subset provided as a certificate is an 1-resilient core of size at most $s$ for the instance provided. We now describe such a verifier as follows:

```
Verifier V: $\langle\langle\mathcal{H},\mathcal{A},\alpha\rangle,h,s\rangle$, Cert
    Parse Cert into a subset $\Omega$ of processes;
    Check if $\Omega \subseteq \mathcal{H}$;
    Check if $|\Omega| \leq s$;
    Check if $h \in \Omega$;
    For every attribute $a \in \mathcal{A}$:
        Check if there are at least one host in $\Omega$ that does not contain $a$;
    If any of these checks fail, then reject, otherwise accept.
```

Each step of the verifier executes in polynomial time on the size of the input. Thus, the total execution time has to be polynomial on the size of the input. This concludes the proof of our claim.

□

With these two claims, we have shown that there is no polynomial-time algorithm for 1-Core if P $\neq$ NP. From [BG94], we have that search reduces to decision for NP-complete problems. The search problem for $k$-Core is as follows:

**Problem:** $k$-Core search problem

**Instance:** A system $\langle \mathcal{H}, \mathcal{A}, \alpha \rangle$ a host $h \in \mathcal{H}$, a positive integer $s > 0$;

**Search for $k$-Core:** Find a subset $H \subseteq \mathcal{H}$ such that $H$ is a $k$-resilient core, $h \in H$, and $|H| \leq s$, or output $\perp$.

Thus, we conclude that there is a polynomial-time algorithm for the 1-Core search problem if and only if there is a polynomial-time algorithm that solves the 1-Core decision problem. Furthermore, the following optimization problem clearly reduces to the $k$-Core search problem:

**Problem:** $k$-MinCore

**Input:** A system $\langle \mathcal{H}, \mathcal{A}, \alpha \rangle$ a host $h \in \mathcal{H}$;

**Output:** a subset $H \subseteq \mathcal{H}$ such that $H$ is a $k$-resilient core and $h \in H$;

**Cost function:** $f(H) = |H|$;

**Goal:** Minimize.

Given an oracle $\text{OS}_{k\text{-Core}}$ that solves the $k$-Core search problem in polynomial time on the size of the input, we can solve the optimization problem by calling the oracle with increasing values of $s$, until the oracle outputs a $k$-resilient core. Note that we need to call the oracle at most $|\mathcal{H}|$ times in the worst case. Thus, running such an algorithm still takes polynomial time on the size of the input. Note also that this argument is valid for any $k > 0$, and consequently it is valid for the case $k = 1$, which is the one we discussed above.

### 6.7.3 Searching for $k$–resilient cores

In the previous section, we showed that the 1-Core problem is NP-complete. In this section we show that computing $k$-resilient core is also a hard problem, $k > 1$.

Recall that the problem of searching for a 1-resilient core consists in determining a subset $H$ of hosts such that the intersection of the set of attributes across all hosts of $H$ is empty. Searching for a $k$-resilient core, for any positive integer $k$, consists in searching for a subset of hosts $H$ such that there is no subset $A$ of $k$ attributes in which every host in $H$ contains at least one attribute of $A$ in their configuration. This constraint can be expressed as follows:

**Definition 6.7.6** A subset $H \subseteq \mathcal{H}$ of hosts is a $k$-resilient core if and only if the following holds:

$\exists\, H_1, \ldots, H_k \subseteq H$ :
$\quad \wedge \exists i \in [1 \ldots k]$ : $H_i$ is a 1-resilient core
$\quad \wedge \forall i, j \in [1 \ldots k], i \neq j : (\cap_{h \in (H_i \cup H_j)} \alpha(h)) = \emptyset$

Note that if a set of processes $H$ satisfies the properties above, then it is necessary a set $A$ formed of at least $k+1$ attributes so that every process in $H$ contains at least one of the attributes in $A$.

**Claim 6.7.7** *$k$-Core $\leq_m$ $(k+1)$-Core , $k \geq 1$*

**Proof:**
We have to show that there is a polynomial-time algorithm **KtoK+1** such that, given an instance of the $K$-Core problem, it outputs an instance of the $(k+1)$-Core. Such an instance must be such that:

i. If $\langle\langle \mathcal{H}, \mathcal{A}, \alpha\rangle, h, s\rangle \in k$-Core, then $\langle\langle \mathcal{H}', \mathcal{A}', \alpha'\rangle, h', s'\rangle \in (k+1)$-Core;

ii. If $\langle\langle \mathcal{H}', \mathcal{A}', \alpha'\rangle, h', s'\rangle \in (k+1)$-Core, then $\langle\langle \mathcal{H}, \mathcal{A}, \alpha\rangle, h, s\rangle \in k$-Core.

We now describe **KtoK+1**:

```
Algorithm KtoK+1:$\langle\langle\mathcal{H},\mathcal{A},\alpha\rangle,h,s\rangle$
    $\mathcal{H}' \leftarrow \mathcal{H} \cup \{h^*\}$, $h^* \notin \mathcal{H}$;
    $\mathcal{A}' \leftarrow \mathcal{A} \cup \{a^*\}$, $a^* \notin \mathcal{A}$;
    $\alpha \leftarrow [\alpha : h^* \rightarrow \{a^*\}]$;
    $h' \leftarrow h$;
    $s' \leftarrow s + 1$;
    output $\langle\langle\mathcal{H},\mathcal{A},\alpha\rangle,h',s'\rangle$;
```

The algorithm clearly runs in polynomial time, since every step is executed in polynomial time on the size of the input. It remains to show (i) and (ii). First, we show (i). Let $C \subseteq \mathcal{H}$ be a $k$-resilient core of size at most $s$. We then have that $C' = C \cup \{h^*\}$ is a $(k+1)$-resilient core. By assumption, $C$ is a $k$-resilient core. Consequently, there is no subset $A$ of $k$ or less attributes such that for all $h \in C$, $\alpha(h) \cap A$ is not empty. The host we add to $C$ to form $C'$ has a single attribute that is not shared by any other host. There are two cases to analyze. First, let $A'$ be a subset of $k+1$ attributes that does not include $a^*$. Such a subset of attributes cannot intersect every host in $C'$ because it does not intersect at least $h^*$. Second, let $A''$ be a subset of $k+1$ attributes such that $A''$ includes $a^*$. Such subset cannot intersect the configuration of every host in $C'$ either. Otherwise, there is a subset of $k$ attributes in $A''$ that intersects the configuration of every host in $C$, thereby contradicting our assumption that $C$ is a $k$-resilient core. We thus have that there is no subset of $k+1$ or less attributes such that for all $h \in C$, $\alpha(h) \cap A$ is not empty, and $C'$ has size at most $s' = s + 1$.

We now show (ii). Let $C'$ be a $(k+1)$-resilient core of size at most $s'$ for the instance of $(k+1)$-Core output by **KtoK+1**. If $C'$ does not contain $h^*$, then there is a host $h'$ in $C'$ such that $C' \setminus \{h'\}$ is a $k$-resilient core. This must be true, otherwise there is a subset of at $k+1$ that intersects the configurations of all the hosts in $C'$. Now, if $C'$ does contain $h^*$, then $C' \setminus \{h^*\}$ is a $k$-resilient core of size at most $s$. To see why $C' \setminus \{h^*\}$ must be a $k$-resilient core observe that if it is not, then there exists a set $A'$ of

$k$ attributes of $\mathcal{A}$ such that they intersect the configurations of all the hosts in $C' \setminus \{h^*\}$. In this case, $C'$ cannot be a $(k+1)$-resilient core either because $A' \cup \{a^*\}$ intersects all the configurations of $C'$. In both cases, we have that the resulting $k$-resilient core has size at most $s' - 1 = s$. This concludes our proof.

□

Using a simple recursive argument, we have that $1$-Core reduces to $k$-Core for any $k > 1$. We therefore have that $k$-Core cannot be solved in polynomial time, unless $1$-Core has a polynomial-time solution.

## 6.8 Conclusions

In this chapter, we proposed a new approach called informed replication for designing distributed systems to survive Internet epidemics that cause catastrophic damage. Informed replication uses a model of correlated failures to exploit software diversity, providing high reliability with low replication overhead. Using host diversity characteristics derived from a measurement study of hosts on the UCSD campus, we developed and evaluated heuristics for determining the number and placement of replicas that have a number of attractive features. Our heuristics provide excellent reliability guarantees (over $0.99$ probability that user data survives attacks of single- and double-exploit pathogens), result in low degree of replication (less than 3 copies for single-exploit pathogens; less than 5 copies for double-exploit pathogens), limit the storage burden on each host in the system, and lend themselves to a fully distributed implementation. We then used this approach in the design and implementation of a cooperative backup system called the Phoenix Recovery Service. Based upon our evaluation results, we conclude that our approach is a viable and attractive method for surviving Internet catastrophes.

The results of this chapter also illustrate an important advantage of incorporating dependent failure information explicitly into a failure model. By using attributes to determine the cores of the system, the amount of replication necessary to protect hosts against Internet attacks is much smaller compared to considering the set of hosts as homogeneous and selecting hosts at random.

# Chapter 7

# Conclusion

As our reliance upon computer systems increases, failures of computers can be highly disruptive. As a result, we use fault-tolerant techniques to build more reliable and more available systems. When building fault-tolerant computer systems, it is common to assume that parts of the system fail independently. In particular, when building replicated systems, it is common to assume that processes fail independently. In practice, however, failures are often not independent due to dependencies of groups of processes upon resources.

To model sets of processes that can fail in the execution of an algorithm, a typical assumption is that is possible to determine a threshold $t$ on the number of process failures. Such an assumption is ideal when processes fail independently and with identical probability distribution because all subsets of processes have the same probability of failure.

To enable the design of algorithms for fault-tolerant systems when failures are not independent, we have proposed in this dissertation a model of dependent failures based on two abstractions: cores and survivor sets. Cores and survivor sets generalize abstractions typically used when designing fault-tolerant algorithms, namely $t + 1$ and $n - t$, where $n$ is the number of processes. Modeling failures with cores and survivor sets has at least three immediate advantages. First, it enables a more expressive characterization of failures. As cores and survivor sets do not reference a value of $t$, it is possible to have heterogeneous sets with respect to size. Second, it is simple to use compared to complex probability models that accurately capture the details of a system. Although probability models may be able to capture more accurately the behavior of a system, it is often more complex to design algorithms using such models, and applying algorithms designed with probability models to different systems is often more difficult. Third, it enables solutions using fewer processes. As our model enables the use of sets of different sizes meeting a particular reliability/availability goal, it is possible to solve problems with fewer processes compared to the number necessary when using a single threshold.

We applied this model to the traditional consensus problem, and derived results for different types of failure and system models. For synchronous systems with crash process failures, we showed that it is sufficient to have a single core in the system. In fact, our algorithm SyncCrash assumes that only the processes in one of the smallest cores are active. For synchronous systems with Byzantine failures, we showed equivalent properties that are necessary and sufficient to enable a solution to consensus under such a model. These properties are Byzantine Partition and Byzantine Intersection. In general, partition properties are useful in showing the minimal requirement on process replication, whereas intersection properties are more useful when designing algorithms. Proofs of lower bound on process replication often refer to partitions and cores, as ours for synchronous Byzantine consensus that concludes that Byzantine Partition must hold. In designing algorithms, however, partition properties are less useful. Intersection properties are more useful in this case because algorithms often refer to survivor sets when waiting for messages from processes and proofs of correctness often refer to the intersection of survivor sets. Our algorithm SyncByz uses survivor sets when constructing messages to send in a round, and assumes Byzantine Intersection for correctness.

A surprising result is the difference on the minimum number of rounds necessary to solve consensus when considering crash and Byzantine failures. As the proof we provided shows, it depends upon the maximum number of failures in the set of processes allowed to send messages (active processes). It is a well-known result that the minimum number of rounds is the same for both crash and Byzantine failures in the threshold model. Thus, considering more expressive models enables the design of algorithms that are, at least theoretically, more efficient.

For asynchronous systems, we also considered crash and Byzantine failures. For crash process failures, there are two equivalent properties, similar to Byzantine Partition and Byzantine Intersection, that determine a necessary and sufficient requirement on the number of processes to enable solutions to consensus. These properties are Crash Partition and Crash Intersection. Similarly to the synchronous case for Byzantine failures, Crash Partition is useful when deriving the necessary requirement on process replication, whereas Crash Intersection is more useful when designing algorithms, *e.g.*, AsyncCrash. For Byzantine failures, we have also shown that Byzantine Partition and Byzantine Intersection are necessary and sufficient to solve consensus in such a model. To show that these properties are sufficient, we described an algorithm called AsyncByz. This algorithm is interesting because it works in environments with both weak failure assumptions and weak timing constraints.

As a natural extension of the set of partition and intersection properties used for consensus, we considered parameterized versions of these properties, which we called $k$–Partition and $k$–Intersection, $k > 1$. The general idea is that $k$–Partition and $k$–Intersection are equivalent, and correspond in our model to the constraint $n > k \cdot t$ in the threshold model, for some value of $k$. We have also shown that there is value in exploring the space of properties even further. A set of parameterized properties ($k$,$k-1$)-Partition and ($k$,$k-1$)-Intersection are important, for example, when solving the weak leader election problem, which arises in the context of the primary-backup approach to replication in systems in which processes can fail by omitting to receive messages. We presented an algorithm that solves this problem in our model. As a corollary of this result, we have also shown that the known lower bound of $n > 3 \cdot t/2$ on process replication for the threshold model is actually tight, which is a result of theoretical value.

Although the space of partition and intersection properties is infinite, there are two major reasons for not having explored it any further. First, it is not clear what form the partition and intersection properties take, as there are different possibilities and it is not clear which one is more useful, or even if any of the forms is useful. Second, we could not find problems, as we did for the other properties, to motivate new definitions. A further expansion of the space of intersection and partition properties is therefore part of future work.

One reason for pursuing a space of partition and intersection properties was to invent a technique for translating automatically algorithms designed for the threshold model to our model of cores and survivor sets. This is an important goal because with such a mechanism we can adapt all previous work done under the threshold model to this new model, without having to recreate all the algorithms designed and all the lower bound results proved for the threshold model. The existence of such a general technique, however, is a conjecture at this point.

In the last two chapters of this dissertation, we considered two practical applications of our model. In multi-site systems, we have shown that it is possible to achieve higher availability for quorum systems when incorporating dependent failure information. In particular, we show that majority quorums do not have necessarily optimal availability, which is a traditional result for when failures of processes are independent and identically distributed. The main observation leading to this conclusion was that multi-site systems experience site failures, where a site failure implies the simultaneous unavailability of all the hosts (or nodes) in a site. Based on this observation, we discussed problems with traditional availability metrics for quorum systems, presenting a new metric that consists in counting survivor sets, and new quorum constructions that are optimal with respect to this new metric. Results from an experiment conducted on PlanetLab showed that our quorum construction not only achieve higher availability, but also has better average case behavior due to the use of fewer replicas per quorum.

To tolerate threats that exploit shared software vulnerability, such as outbreaks of worms and viruses, we proposed to replicate data using a technique called informed replication. This technique consists in selecting replica sets using attributes, where different attributes indicate different vulnerabilities. In the case of large-scale Internet attacks, these attributes are the software systems hosts run. We then used the core abstraction to represent subsets of participating hosts such that at least one survives such an attack.

As building such cores requires a population that is diverse in the software systems they run, we conducted a study of the UCSD network to determine whether there is sufficient diversity in a large Internet setting to make this approach feasible. We concluded that, although the popularity of software systems is highly skewed (*e.g.*, over 50% of the hosts run Windows), the amount of diversity is sufficient to enable efficient replication mechanisms. In particular, we have designed heuristics that enable hosts to select replica sets that often comprise the host itself and one single extra replica. Our simulations and experimental results from a deployment on PlanetLab show that it is possible for a host to survive such attacks with high probability, using a small amount of replication (less than two extra replicas on average) and committing a small amount of resources (a host participating in at most three cores from other hosts). Compared to a technique that selects hosts at random, ignoring the diversity of the population of participating hosts, our techniques use three times less storage, thus reducing significantly the storage overhead.

To conclude, incorporating dependent failures into the failure model when designing fault-tolerant algorithms has both theoretical and practical benefits. We have shown these benefits by considering important theoretical problems, such as consensus and leader election, as well as practical applications, such as replication in multi-site systems and cooperative systems that tolerate large-scale Internet attacks. As current systems increase in size and extent, more realistic failure models will have a fundamental role in system design, and incorporating dependent failures is certainly an important part of such failure models. Our results on modeling with cores and survivor sets therefore constitute an important step toward more practical designs.

# Bibliography

[ACT97] Marcos Aguilera, Wei Chen, and Sam Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG)*, volume 1320 of *LNCS*, pages 126–140, Saarbrùcken, Germany, September 1997. Springer-Verlag.

[AFM99] Bernd Altmann, Matthias Fitzi, and Ueli Maurer. Byzantine agreement secure against general adversaries in the dual failure model. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC)*, volume 1693 of *LNCS*, pages 123–139. Springer-Verlag, Sep 1999.

[AW96] Y. Amir and A. Wool. Evaluating quorum systems over the Internet. In *Proceedings of the IEEE 26th International Symposium on Fault-Tolerant Computing (FTCS)*, pages 26–37, Sendai, Japan, June 1996.

[AW98a] Yair Amir and Avishai Wool. Optimal availability quorum systems: Theory and practice. *Information Processing Letters*, 65(5):223–228, March 1998.

[AW98b] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*, chapter 3. McGraw-Hill, 1998.

[AW98c] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*, chapter 5. McGraw-Hill, 1998.

[AWL05] Mike Afergan, Joel Wein, and Amy LaMeyer. Experience with some principles for building an Internet-scale reliable system. In *Proceedings of the 2nd Workshop on Real, Large Distributed Systems (WORLDS)*, pages 1–6, San Francisco, CA, December 2005.

[BAF$^+$03] Elena Barrantes, David Ackley, Stephanie Forrest, Trek Palmer, Darko Stefanović, and Dino Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 281–289, Washington D.C., USA, October 2003.

[BBB$^+$04] Jean-Michel Busca, Marin Bertier, Fatima Belkouch, Pierre Sens, and Luciana Arantes. A performance evaluation of a quorum-based state-machine replication algorithm for computing grids. In *Proceedings of the 16th IEEE SBAC-PAD'04*, Foz do Iguaçú, PR, Brazil, October 2004.

[BBST01] C. Batten, K. Barr, A. Saraf, and S. Treptin. pStore: A secure peer-to-peer backup system. Technical Report MIT-LCS-TM-632, MIT, December 2001.

[BG94] Mihir Bellare and S. Goldwasser. The complexity of decision versus search. *SIAM Journal on Computing*, 23(1):97–119, February 1994.

[BGM86] Daniel Barbara and Hector Garcia-Molina. The vulnerability of vote assignments. *ACM Transactions on Computer Systems*, 4(3):187–213, August 1986.

[BI95] Jan Bioch and Toshihide Ibaraki. Generating and approximating nondominated coteries. *IEEE Transactions on Parallel and Distributed Systems*, 6(9):905–914, September 1995.

[Bir99] Kenneth Birman. A review of experiences with reliable multicast. *Software: Practice and Experience*, 29(9):741–774, July 1999.

[bir06] The Biomedical Informatics Research Network (BIRN), April 2006. `http://www.nbirn.net`.

[BJZH04] Ali Raza Butt, Troy A. Johnson, Yili Zheng, and Y. Charlie Hu. Kosha: A peer-to-peer enhancement for the network file system. In *Proceedings of ACM/IEEE Supercomputing*, Pittsburgh, PA, Nov 2004.

[BMST92] Navin Budhiraja, Keith Marzullo, Fred Schneider, and Sam Toueg. Optimal primary-backup protocols. In *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG)*, volume 647 of *LNCS*, pages 362–378, Haifa, Israel, Nov 1992. Springer-Verlag.

[Bura] U.S. Census Bureau. Computer and Internet use in the United States: 2003. P23-208.

[Burb] U.S. Census Bureau. Computers and office and accounting machines: 2004. MA334R(04)-1.

[Burc] U.S. Census Bureau. Quarterly retail *e*-commerce sales: 4th quarter 2005. CB06-19.

[BWWG02] M. Bakkaloglu, J. J. Wylie, C. Wang, and G. R. Ganger. On correlated failures in survivable storage systems. Technical Report CMU-CS-02-129, Carnegie-Mellon University, May 2002.

[CBS00] Bernardette Charron-Bost and Andre Schiper. Uniform consensus is harder than consensus. Technical Report DSC/2000/028, École Polytechnique Fédérale de Lausanne, Switzerland, May 2000.

[Che99] Yuan Chen. A prototype implementation of archival intermemory. In *Proceedings of the Fourth ACM International Conference on Digital Libraries*, pages 28–37, Berkeley, CA, August 1999.

[Chr90] Flaviu Christian. Synchronous atomic broadcast for redundant broadcast channels. *Journal of Real-Time Systems*, 2:195–212, Sep 1990.

[CHT96] Tushar Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, Jul 1996.

[Cis] Cisco Systems, Inc. Using Network-Based Application Recognition and Access Control Lists for Blocking the "Code Red" Worm at Network Ingress Points. Cisco Technical Documentation.

[CL02] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[CN02] L. P. Cox and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th ACM/USENIX OSDI*, pages 285–298, Boston, MA, December 2002.

[cod01] Codegreen, September 2001. `http://www.winnetmag.com/Article/ArticleID/22381/22381.html`.

[CPM+98] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, January 1998.

[crc01] CRclean, September 2001. `http://www.winnetmag.com/Article/ArticleID/22381/22381.html`.

[CRL03] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. BASE: Using Abstraction to Improve Fault Tolerance. *ACM Transactions on Computer Systems*, 21(3):236–269, Aug 2003.

[CT96] Tushar Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar 1996.

[DBB93] Joanne Dugan, Salvatore Bavuso, and Mark Boyd. Fault trees and Markov models for reliability analysis of fault-tolerant systems. *Reliability engineering and system safety*, 39(3):291–307, 1993.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the Minimal Synchronism Needed for Distributed Consensus. *Journal of the ACM*, 1(34):77–97, January 1987.

[DS83] D. Dolev and H. R. Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal of Computing*, 12(4):656–666, November 1983.

[DS98] A. Doudou and A. Schiper. Muteness Detectors for Consensus with Byzantine Processes. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, page 315, Puerto Vallarta, Mexico, July 1998. (Brief Announcement).

[FK99] Ian Foster and Carl Kesselman, editors. *The Grid Blueprint for a New Computing Infrastructure*, chapter 1. Morgan Kauffman, 1st edition, 1999.

[FL82] M. Fischer and N. Lynch. A lower bound on the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, June 1982.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

[Gif79] David Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating System Principles (SOSP)*, pages 150–162, Pacific Grove, CA, December 1979.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A guide to the theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[GM04] Seth Gilbert and Grzegorz Malewicz. The Quorum Deployment Problem. In *Proceedings of the International Conference on Principles of Distributed Computing (OPODIS)*, pages 218–228, Grenoble, France, April 2004.

[GMB85] Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, October 1985.

[GR03] Rachid Guerraoui and Michel Raynal. A generic framework for indulgent consensus. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 88–95, Providence, Rhode Island, May 2003.

[GS91] Jim Gray and Daniel Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9):39–48, Sep 1991.

[GS96] R. Guerraoui and A. Schiper. Consensus Service: A modular approach for building fault-tolerant agreement protocols in distributed systems. In

*Proceedings of the IEEE 26th International Symposium on Fault-Tolerant Computing (FTCS)*, Sendai, Japan, June 1996.

[Hal02] Eran Halperin. Improved approximation algorithms for the vertex cover problem in graphs and hypergraphs. *SIAM Journal on Computing*, 31(5):1608–1623, February 2002.

[HM97] Martin Hirt and Ueli Maurer. Complete characterization of adversaries tolerable in secure multi-party computation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 25–34, Santa Barbara, California, August 1997.

[HMD05] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 143–158, Boston, MA, May 2005.

[Hya04] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium*, pages 271–286, August 2004.

[Ins90] The Institute of Electrical and Electronics Engineers (IEEE). *IEEE Standard Glossary of Software Engineering Terminology*, September 1990. IEEE Std 610.12-1990, ISBN 1-55937-067-X.

[Ins04] Insecure.org. The nmap tool, January 2004. `http://www.insecure.org/nmap`.

[JBH$^+$05] Flavio Junqueira, Ranjita Bhagwan, Alejandro Hevia, Keith Marzullo, and Geoffrey M. Voelker. Coping with Internet catastrophes. Technical Report CS2005–0816, UCSD, Feb 2005.

[JBM$^+$03] Flavio Junqueira, Ranjita Bhagwan, Keith Marzullo, Stefan Savage, and Geoffrey M. Voelker. The Phoenix Recovery System: Rebuilding from the ashes of an Internet catastrophe. In *Proceedings of HotOS-IX*, pages 73–78, Lihue, HI, May 2003.

[JM03a] Flavio Junqueira and Keith Marzullo. Designing algorithms for dependent process failures. In *Proceedings of FuDiCo*, volume 2584 of *LNCS*, pages 24–28. Springer-Verlag, Jan 2003.

[JM03b] Flavio Junqueira and Keith Marzullo. Synchronous consensus for dependent process failures. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 274–283, Providence, Rhode Island, May 2003.

[JM05a] Flavio Junqueira and Keith Marzullo. Replication predicates for dependent-failure algorithms. In *Proceedings of the 11th Euro-Par Conference*, LNCS 3648, pages 617–632, Lisbon, Portugal, August 2005.

[JM05b] Flavio Junqueira and Keith Marzullo. The virtue of dependent failures in multi-site systems. In *Proceedings of the IEEE Workshop on Hot Topics in System Dependability*, Supplemental volume of DSN'05, pages 242–247, Yokohama, Japan, June 2005.

[KA94] Jeffrey O. Kephart and William C. Arnold. Automatic extraction of computer virus signatures. In *Proceedings of the 4th Virus Bulletin International Conference*, pages 178–184, Abingdon, England, 1994.

[KBC$^+$00] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*, pages 190–201, Cambridge, MA, 2000.

[KC03] Christian Kreibich and Jon Crowcroft. Honeycomb – Creating intrusion detection signatures using honeypots. In *Proceedings of HotNets-II*, pages 51–56, Cambridge, MA, November 2003.

[KF05] Klaus Kursawe and Felix Freiling. Byzantine fault tolerance on general hybrid adversary structures. Technical Report AIB-2005-09, Aachen University, Germany, Jan 2005.

[KMMS97] Kim Kihlstrom, L. Moser, and P. M. Melliar-Smith. Solving Consensus in a Byzantine Environment using an Unreliable Failure Detector. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, pages 61–76, Chantilly, France, December 1997.

[KR01] Idith Keidar and Sergio Rajsbaum. On the Cost of Fault-Tolerant Consensus When There Are No Faults - A Tutorial. Technical Report MIT-LCS-TR-821, MIT, May 2001.

[Kum91] Akhil Kumar. Hierarchical Quorum Consensus: A new algorithm for managing replicated data. *IEEE Transactions on Computers*, 40(9):996–1004, September 1991.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[Lam02] Leslie Lamport. *Specifying systems: The TLA$^+$ language and tools for hardware and software engineers*. Addison-Wesley, 2002.

[Lam05] Leslie Lamport. Fast Paxos. Technical Report MSR-TR-2005-112, Microsoft Research, July 2005.

[LEB$^+$03] Mark Lillibridge, Sameh Elnikety, Andrew Birrell, Mike Burrows, and Michael Isard. A cooperative Internet backup scheme. In *Proceedings of USENIX Annual Technical Conference*, pages 29–42, San Antonio, TX, 2003.

[LF82] Leslie Lamport and Michel Fischer. Byzantine Generals and Transaction Commit Protocols. Technical report, SRI International, April 1982.

[Lis06] Tom Liston. LaBrea: "Sticky" Honeypot and IDS. Technical report, April 2006. `http://labrea.sourceforge.net/`.

[LLO$^+$03] John Levin, Richard LaBella, Henry Owen, Didier Contis, and Brian Culver. The use of honeynets to detect exploited systems across large enterprise networks. In *Proceedings of the IEEE Information Assurance Workshop*, pages 92–99, Atlanta, GA, June 2003.

[LMK$^+$03] John W. Lockwood, James Moscola, Matthew Kulig, David Reddick, and Tim Brooks. Internet worm and virus protection in dynamically reconfigurable hardware. In *Proceedings of MAPLD*, September 2003.

[LS02] Nancy Lynch and Alex Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of International Symposium on Distributed Computing (DISC)*, pages 173–190, Toulouse, France, October 2002.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, Jul 1982.

[MA05] Jean-Philippe Martin and Lorenzo Alvisi. Fast Byzantine Consensus. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, pages 402–411, Yokohama, Japan, June 2005.

[Mae85] Mamoru Maekawa. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.

[Mic] Microsoft Corporation. Microsoft windows update. `http://windowsupdate.microsoft.com`.

[Mos91] Ali Mosleh. Dependent failure analysis. *Reliability engineering and system safety*, 34(4):243–248, 1991.

[MPS$^+$03] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the Slammer worm. *IEEE Privacy & Security*, 1(4):33–39, Jul 2003.

[MR97a] Dahlia Malkhi and Michael Reiter. Byzantine Quorum Systems. In *Proceedings of the 29th ACM STOC*, pages 569–578, May 1997.

[MR97b] Dahlia Malkhi and Michael Reiter. Unreliable Intrusion Detection in Distributed Computations. In *Proceedings of the 10th Computer Security Foundations Workshop(CSFW97)*, pages 116–124, Rockport, MA, June 1997.

[MS04] David Moore and Colleen Shannon. The spread of the Witty worm, April 2004. http://www.caida.org/analysis/security/witty/.

[MSB02] David Moore, Colleen Shannon, and Jeffrey Brown. Code Red: A case study on the spread and victims of an Internet worm. In *Proceedings of ACM Internet Measurement Workshop (IMW)*, pages 273–284, Marseille, France, November 2002.

[MSVS03] David Moore, Colleen Shannon, Geoffrey M. Voelker, and Stefan Savage. Internet quarantine: Requirements for containing self-propagating code. In *Proceedings of IEEE Infocom Conference*, pages 1901–1910, San Francisco, CA, April 2003.

[Mul95a] Sape Mullender, editor. *Distributed Systems*, chapter 5. Addison-Wesley, 2nd edition, 1995.

[Mul95b] Sape Mullender, editor. *Distributed Systems*, chapter 8. Addison-Wesley, 2nd edition, 1995.

[MVS01] David Moore, Geoffrey M. Voelker, and Stefan Savage. Inferring Internet denial of service activity. In *Proceedings of the USENIX Security Symposium*, Washington, D.C., August 2001.

[myd04] Lurhq. mydoom word advisory, January 2004. `http://www.lurhq.com/mydoomadvisory.html`.

[NW98] Moni Naor and Avishai Wool. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, April 1998.

[Pat02] David Patterson. A simple way to estimate the cost of downtime. In *Proceedings of the 16th USENIX Large Installation System Administration Conference*, pages 185–188, Philadelphia, PA, November 2002.

[pla06] The Planetlab testbed, April 2006. `http://www.planet-lab.org/`.

[PR04] Philippe Raopin Parvèdy and Michel Raynal. Optimal early stopping uniform consensus in synchronous systems with process omission failures. In *Proceedings of the 16th ACM SPAA*, pages 302–310, Jun 2004.

[PSL80] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):pp. 228–234, April 1980.

[PW95a] David Peleg and Avishai Wool. The availability of quorum systems. *Information and Computation*, 123(2):210–223, December 1995.

[PW95b] David Peleg and Avishai Wool. Crumbling Walls: A class of practical and efficient quorum systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 120–129, Ontario, Canada, August 1995.

[RD01] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of ACM Middleware*, pages 329–350, Heidelberg, Germany, November 2001.

[Res] Infonetics Research. The costs of enterprise downtime: North American vertical markets 2005. Jan 2005.

[RKB$^+$04] Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostic, and Amin Vahdat. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proceedings of the 1st ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 267–280, San Francisco, CA, March 2004.

[Ros00] Sheldon Ross. *Introduction to probability models*. Harcourt Academic Press, 2000.

[Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys, December 1990.

[Sch97] Andre Schiper. Early Consensus in a Asynchronous System with a Weak Failure Detector. *Distributed Computing*, 10(4):149–157, April 1997.

[SEVS04] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated Worm Fingerprinting. In *Proceedings of the 6th ACM/USENIX OSDI*, pages 45–60, San Francisco, CA, December 2004.

[SK03] Stelios Sidiroglou and Angelos D. Keromytis. A network worm vaccine architecture. In *Proceedings of IEEE Workshop on Enterprise Security*, pages 220–225, Linz, Austria, June 2003.

[sob] Lurhq. sobig.a and the spam you received today. `http://www.lurhq.com/sobig.html`.

[Sop04] Sophos anti-virus. W32/Sasser-A worm analysis. `http://www.sophos.com/virusinfo/analyses/w32sassera.html`, May 2004.

[Sym] Symantec. Symantec Security Response. `http://securityresponse.symantec.com/`.

[TK02] Thomas Toth and Christopher Kruegel. Connection-history based anomaly detection. Technical Report TUV-1841-2002-34, Technical University of Vienna, June 2002.

[TW03] Jamie Twycross and Matthew M. Williamson. Implementing and testing a virus throttle. In *Proceedings of the 12th USENIX Security Symposium*, pages 285–294, Washington, D.C., August 2003.

[WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of NDSS*, pages 3–17, San Diego, CA, February 2000.

[WGSZ04] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of ACM SIGCOMM*, pages 193– 204, Portland, Oregon, August 2004.

[Wil02] Matthew Williamson. Throttling Viruses: Restricting propagation to defeat malicious mobile code. Technical Report HPL-2002-172, HP Laboratories Bristol, June 2002.

[WMK02] H. Weatherspoon, T. Moscovitz, and J. Kubiatowicz. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *Proceedings of International Workshop on Reliable Peer-to-Peer Distributed Systems*, October 2002.

[Won04] Cynthia Wong et al. Dynamic quarantine of Internet worms. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, pages 73–82, Florence, Italy, June 2004.

[WSP04] Nicholas Weaver, Stuart Staniford, and Vern Paxson. Very fast containment of scanning worms. In *Proceedings of the USENIX Security Symposium*, pages 29–44, San Diego, CA, August 2004.

[Yu04] Haifeng Yu. Signed Quorum Systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 246–255, St. John's, Newfoundland, Canada, July 2004.

[ZGGT03] Cliff C. Zou, Lixin Gao, Weibo Gong, and Don Towsley. Monitoring and early warning for Internet worms. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 190–199, Washington D.C., USA, October 2003.

# Appendix A

# Weak Leader Election in the receive-omission failure model

Leader Election is an important primitive in fault-tolerant distributed computing because it enables the solution of problems broadly applicable in real systems such as consensus, as illustrated by the Paxos algorithm [Lam98], and primary-backup protocols, as in [BMST92].

The particular version of the Leader Election problem we develop upon first appeared in the context of primary-backup protocols. In the primary-backup approach for fault-tolerant services, clients issue requests that the primary is responsible for handling and replying to. When the primary fails, one of the backup replicas takes over as the new primary. Thus, a primary-backup protocol embeds a Leader Election algorithm that can infinitely often select a primary.

In [BMST92], Budhiraja and Marzullo show a lower bound of $n > \lfloor 3t/2 \rfloor$ for such algorithms when processes can fail to receive messages, where $n$ is the number of processes and $t$ is the maximum number of process failures in an execution. The basic idea of the lower bound proof is that multiple primaries can be elected if fewer than $\lfloor 3t/2 \rfloor + 1$ processes compose the system. In a later section, we repeat this result for exposition purposes.

Still on the early work by Budhiraja and Marzullo on primary-backup protocols, the degree of replication necessary for the algorithm they designed is higher because they require faulty processes not to be elected [BMST92]. According to their statement of the problem, if a process does not crash but it commits receive-omission failures, then it cannot be elected. This is due to the assumption that responses to client requests are bounded in time. Failure detection for receive-omission failures, however, requires at least twofold replication. In our statement of the leader election problem, we allow faulty processes to be elected, and this is the reason for naming the problem as a weaker version of the traditional leader election problem.

When implementing a system based on the primary-backup approach, servers are often connected by a local area network to bound response time to client requests, which implies bounded fail-over time. For such settings, partitions are unlikely to occur if processors operate at a reasonable speed. Messages, however, can be lost due to, for example, buffer overflows at the receiver. One can imagine using retransmissions to cope with such failures. A retransmission mechanism, however, only guarantees eventual delivery; bounded response is not possible with eventual delivery of messages. Because of the requirements on bounded response and failure-over time, primary-backup protocols are usually synchronous.

In this chapter, we describe a synchronous algorithm for Leader Election under receive-omission process failures and prove its correctness. The novelty in this algorithm is fourfold: 1) it proves tight a lower bound that has been known for over 10 years; 2) by permitting faulty (but not crashed) processes to be elected, it requires fewer replicas; 3) it is based on cores and survivor sets which are abstractions that enable one to more expressively represent failure scenarios by considering failures that are not independent or not identically distributed; 4) although it allows for faulty processes to be elected, correct processes are able to detect this situation, enabling the use of alarms to indicate failures in the system. Relating to our discussion on primary-backup protocols, by assuming that faulty processes can be elected, we cannot bound response time for a primary-backup protocol. We can guarantee, however, that there is at most one primary at any time, and that response is bounded whenever a correct process emerges as the primary. We further discuss this and other issues with primary-backup protocols later in the chapter.

The remainder of this chapter is organized as follows. We describe in detail the system model in Section A.1. We then introduce the problem by stating the properties an algorithm must fulfill (Section A.2). Still in Section A.2, we repeat the lower bound proof for process replication, and generalize this bound to our model of dependent failures. Section A.3 describes our FFS-WLE algorithm for Leader Election. As we shall see, the algorithm depends on a primitive that we call *RO consensus*. The properties for RO consensus resemble the ones for the traditional uniform consensus primitive. The differences, however, are significant enough for naming the problem differently. In Section A.3, we also provide an algorithm for RO consensus. Sections A.4 and A.5 provide proofs of correctness for FFS-ROC and FFS-WLE, respectively. In Section A.6, we strengthen the definition of Leader Election to disable executions in which different leaders are elected infinitely often, and provide a simple modification of the algorithm that enables it. Before concluding, we provide a discussion on the implications of the properties of our algorithm in a primary-backup protocol in Section A.7. We finally conclude in Section A.8.

## A.1 System model

A system is a collection of processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ that communicate through messages.[1] For every pair of processes $p_i, p_j \in \Pi$, there is a channel that $p_i$ uses to send messages to $p_j$.

In such a system, an algorithm *alg* is a collection of state machines, one for each process. *alg* then proceeds in steps of processes. In a step, a process $p_i$ executes atomically the following:

$$\bigwedge \begin{array}{ll} \bigvee & \text{receives a message from a process } p_j \\ \bigvee & \text{sends a message to a process } p_j \end{array}$$

[1]We use $p_i$ to denote a process and $i$ to denote the identifier of this process.

$\bigvee$ executes a local operation

$\bigwedge$ undergoes a state transition

The definition of an execution is as in Chapter 2. We now review it here. An execution $E$ of *alg* is a tuple $\langle \mathit{Init}, \mathit{Steps}, \mathit{Time}, \mathit{Faulty} \rangle$, where: *Faulty* is a mapping from step to set of faulty processes; *Init* is the set of initial states, one for each process; *Steps* is a set of steps; *Time* is a mapping from step to integer, such an integer corresponding to global elapsed time. We use global time in proofs, but we do not assume that such a clock that produces global time is available to processes. The assumption of such a virtual clock is useful for defining executions, more specifically to order steps of processes with respect to time. We also use $\mathit{Correct}(E)$ for the set of processes that are correct in $E$. Finally, $\mathcal{E}$ is the set of executions of *alg*.

We assume that processes can fail by crashing or by omitting to receive messages. If a process $p_i$ crashes in an execution $E$, then there is step $s$ of $p_i$ such that $p_i$ executes no further steps after $\mathit{Time}(s)$. We call this step a *crash step*. A faulty process, however, does not necessarily crash: it can selectively fail to receive messages. To characterize failure scenarios, we use our model of dependent process failures based on the abstractions of cores and survivor sets. We assume that systems are synchronous: the steps of every execution of some algorithm $\mathcal{A}$ can be split into rounds. That is, there is a mapping $\mathit{Round} : S \rightarrow \mathcal{R}$ from steps of processes to round numbers, where $\mathcal{R} = \mathbb{Z}^*$ and round numbers monotonically increase with time. We then have the following properties for rounds:

**P-Liveness** : If a process executes all the steps of a round $r$, then every process that does not crash by $r$ executes at least one step of $r$.

**C-Liveness** : If a process $p_i$ sends a message $m$ to a correct process $p_j$ in round $r$ and $p_i$ does not crash by round $r$, then $p_j$ receives $m$ in round $r$.

**Integrity** : If $p_i$ receives a message $m$ from $p_j$, then $p_j$ sent $m$ to $p_i$.

**No duplicates** : No message $m$ is received more than once.

## A.2 Problem specification

For the following description of the problem, we assume that each process $p_i$ in $\Pi$ has a boolean variable $p_i$*.elected* that is set to true if the process elects itself, and to false otherwise. We then define the *weak leader election* problem with the following three properties:

**Safety** $\Box|\{p_i \in \Pi : p_i.\mathit{elected}\}| < 2.$

**LE-Liveness** $\Box\Diamond(|\{p_i \in \Pi : p_i.\mathit{elected}\}| > 0).$

**FF-Stability** In a failure-free execution, only one process ever has *elected* set to true.

In words, infinitely often some process elects itself, and no more than one process is elected at any time. The third property eliminates the possibility of an algorithm that, for example, elects processes in a round-robin fashion (which can be implemented with *no* communications given that the system is synchronous). It does not rule out, however, executions in which two or more processes are elected infinitely often when there is at least one process failure in the execution. For this reason, we propose another property called *E-stability* stated as follows:

**E-Stability** $\exists p_i \in \Pi : \Diamond\Box(\forall p_j \in \Pi : p_j.\mathit{elected} \Rightarrow (\, j \;=\; i\,)\,)$

An algorithm satisfying this property eventually elects the same process forever in every execution. Note that with E-stability only, failure-free executions are allowed to have multiple leaders elected (at different times, to not violate safety), and hence does not render FF-stability unnecessary.

In the following sections, we first derive an algorithm that satisfies the first three properties. Later we modify this algorithm to also satisfy E-stability. We discuss a lower bound for this problem in Chapter 4.

## A.3 The algorithm

In this section, we describe an algorithm FFS-WLE that satisfies safety, LE-liveness, and FF-stability (Figure A.2). It assumes a system profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ that satisfies (3,2)-Intersection and uses as a building block an algorithm FFS-ROC that implements a weak version of uniform consensus that we call *RO consensus*. We call it RO consensus because its definition resembles the one of consensus. It is tailored, however, to fulfill the requirements of FFS-WLE. Note that we use the prefix "FFS-" (FF-stability) to distinguish the algorithms in this section from the ones of Section A.6. Recall that in Section A.6 we present an algorithm that also satisfies E-stability.

We assume that each process $p_i$ has an initial value $p_i.a \in V \cup \{\perp\}$, where $V$ is the set of initial values and $\perp$ is a default value, and a decision value $p_i.d\,[1 \ldots n]$, where $p_i.d[j] \in V \cup \{\perp\}$. We use $v \in p_i.d$ to denote that there is some $p_\ell \in \Pi$ such that $p_i.d[\ell] = v$. If a process $p_i$ crashes, then we assume that its decision value $p_i.d$ is $\mathcal{N}$, where $\mathcal{N}$ stands for the $n$ element list $[\perp, \ldots, \perp]$. To avoid repetition throughout the discussion of our algorithm, we say that a process $p$ decides in an execution $E$ if $p.d$ is different than $\mathcal{N}$.

As we describe later, we execute FFS-ROC multiple times in electing a leader. We then have that processes may crash before starting an execution $E$ of FFS-ROC. Such processes consequently have initial value undefined in $E$. We therefore use $\perp$ to denote the initial value of crashed processes. That is, if $p_i.a = \perp$, then $p_i$ has crashed.

Let the relation $x \subseteq y$ for $x$ and $y$ lists of $n$ elements be that, for all $i : 1 \leq i \leq n$, $(x[i] \neq \perp) \Rightarrow (x[i] = y[i])$.

The specification of RO consensus is given by four properties as follows:

*Termination*: Every process that does not crash eventually decides on some value.

*Agreement*: If $p_i.d[\ell] \neq \perp$, then for every non-faulty $p_c$, $p_i.d[\ell] = p_c.d[\ell]$;

*RO Uniformity*: Let *vals* be $\{d : \exists p_i \in \Pi \textit{ s.t. } (p_i.d = d)\} \setminus \mathcal{N}$. Then,

$\bigwedge \; 1 \leq |vals| \leq 2$

$\bigwedge\ \forall d, d' \in \mathit{vals} : d \subseteq d' \vee d' \subseteq d$

$\bigwedge\ \forall d_f, d_c \in \mathit{vals}, d_f \subseteq d_c : \exists S_f, S_c \in \mathcal{S}_\Pi :$

$\quad \wedge\ \forall p \in S_f : \vee\ p$ crashes

$\qquad\qquad\qquad \vee\ p.d = d_f$

$\quad \wedge\ \forall p \in S_c : \wedge\ p.d = d_c$

$\qquad\qquad\qquad \wedge\ p$ is not faulty

That is, there can be no more than two non-$\mathcal{N}$ decision values, and if there are two then one is a subset of the other. Furthermore, if there are two different decision values, then these are the values that processes in two disjoint survivor sets decide upon, one for the processes of each survivor set.

*Validity*:

$\bigwedge$ If $p_j \in \Pi$ does not crash, then for all non-faulty $p_i$, $p_i.d[j] = p_j.a$

$\bigwedge$ If $p_j \in \Pi$ does crash, then for all non-faulty $p_i$, $p_i.d[j] \in \{\perp, p_j.a\}$

$\bigwedge$ If there are survivor sets $S_f, S_c \in \mathcal{S}_\Pi$

and values $v_f, v_c \in V$, $v_f \neq v_c$, such that

$\quad \bigwedge\ \forall p \in S_f : p.a \in \{v_f, \perp\}$

$\quad \bigwedge\ \forall p \in S_c : \bigwedge\ p.a = v_c$

$\qquad\qquad\qquad \bigwedge\ p$ is not faulty

$\quad \bigwedge\ \exists p_i, p_\ell \in \Pi : p_i.d[\ell] = v_f$

then for all $p_j$ that does not crash, $v_f \in p_j.d$

That is, if a process $p_i$ is not faulty and $p_i.d[j] \neq \perp$, then the value of $p_i.d[j]$ must be $p_j.a$. The value of $p_i.d[j]$ can be $\perp$ only if $p_j$ crashes. The third case exists because we use the decision values of an execution as the initial values for another execution. From RO uniformity, there can be two different non-$\mathcal{N}$ values $d_f$ and $d_c$. If this is the case, then there is a survivor set $S_c$ containing only correct processes such that all processes in $S_c$ decide upon $d_c$, and another survivor set $S_f$ containing only faulty processes such that all the processes in $S_f$ either crash or decide upon $d_f$. Let $d_f$ be $v_f$ and $d_c$ be $v_c$. By the third case, if some process that decides includes $v_f$ in its decision value, then every process that does not crash also includes $v_f$ in its decision value.

We now describe our algorithm FFS-ROC for RO consensus. Figure A.1 shows the pseudocode for a single process. A TLA+ specification of the algorithm appears in Appendix F. From the figure, the algorithm FFS-ROC executes exactly in $t + 1$ rounds, where $t = \min_s\{s = |S_i| \wedge S_i \in \mathcal{S}_\Pi\}$ or alternatively $t + 1 = \max_c\{c = |C_i| \wedge C_i \in \mathcal{C}_\Pi\}$.[2] For the proof of correctness we present in the next section, we assume that the value of $t$ is at least one ($t \geq 1$). Note that for $t = 0$ there is a trivial, much simpler algorithm.

In every round $r$ of FFS-ROC, a process $p_i$ sends its list $p_i.A$ of values to a subset of the processes $\Pi'$ in $\Pi$. If process $p_i$ does not crash or stop[3] in round $r$, then $\Pi = \Pi'$. Otherwise, this subset is arbitrary. Before the end of round $r$, every process $p_i$ that does not execute a crash step at $r$ receives all the messages sent to it in round $r$. Note that if a process $p_i$ crashes in round $r$, but sends a message $m_i$ to process $p_j$, then $p_j$ does not necessarily receive $m_i$ by C-liveness. We then use the following, where $0 \leq r \leq t$:

- $p_i.M(r)$ denotes the set of messages of round $r$ that $p_i$ receives;
- $p_i.s(r)$ denotes the set of processes from which process $p_i$ receives messages of round $r$. That is, $p_i.s(r) = \{p : m \in p_i.M(r) \wedge m.\mathit{from} = p\}$;
- $p_i.sr(r)$ denotes the set $p_i.s(r)$ removed the processes $p_i$ detects to be faulty in round $r$.

Processes send no messages at the last round. Note that, by the algorithm, messages received by the end of round $r$ are available for processing at the beginning of round $r + 1$.

[2]The first and the last rounds in the algorithms are actually half rounds, and we then consider that both together constitute a single round. Put another way, we can easily rearrange the order of sending and receiving messages to make it fit into $t + 1$. We have chosen the former for expositional convenience.

[3]A *stop* instruction is equivalent to a crash in that a process does not execute any further steps after executing a stop instruction. A process, however, executes a stop instruction according to its own state machine, and hence it is not in any arbitrary step.

If a process detects that it has failed to receive messages, then it stops by deciding on $\mathcal{N}$. In the discussion that follows, we treat processes that crash and processes that stop in the same manner. If a distinction is necessary, then we clearly state it. There are two ways a processes $p_i$ can determine that it is faulty:

1. By receiving messages from a set of processes in round $r$ such that $p_i.s(r) \not\subseteq p_i.s(r-1)$, $r > 1$; 2) By determining that in its set of messages of round $r$, there is no survivor set possibly containing only correct processes.

2. The second form of detection relies on the set of values $p_i$ receives from another process $p_j$. If $p_i$ notices that $p_j$ did not receive a previous message from $p_i$, then $p_i$ declares $p_j$ faulty. By removing the obviously faulty processes and looking at the remaining set, if there is no survivor set in the remaining set, then $p_i$ must be faulty as well. More specifically, $p_i$ checks in round $r > 1$ whether $p_i.sr(r)$ contains a survivor set. To decide upon membership for $p_i.sr(r)$, $p_i$ uses the value of $p_i.A$ from round $r-2$. We use $p_i.A_p(r)$ as the value of $p_i.A$ in round $r$ after $p_i$ updates $p_i.A$ with the values received in the messages from round $r$. Note that the value of $p_i.A_p(r)$, $0 \leq r \leq t-1$, is used only in round $r+2$.

Figure A.2 shows the pseudocode for an algorithm that implements safety, LE-liveness, and FF-stability. It proceeds in iterations of an infinite repeat loop. In each iteration, a process executes FFS-ROC twice, and decides if it has to elect itself by the end of the second phase.

In the following sections, we prove the correctness of both FFS-ROC and FFS-WLE.

## A.4 Correctness of FFS-ROC

We provide a proof of correctness for the FFS-ROC algorithm. We say that a process $p_i$ is alive in round $r$ if either one of the following happens:

```
**Algorithm** FFS-ROC on input $p_i.a$
**round** $0$:
    $p_i.s(0) \leftarrow \Pi$; $p_i.sr(0) \leftarrow p_i.s(0)$
    $p_i.A\,[i] \leftarrow p_i.a$
    **for all** $p_k \in \Pi$, $p_k \neq p_i : p_i.A\,[i] \leftarrow \bot$
    $p_i.A_p(0) \leftarrow p_i.A$
    **send** $p_i.A$ to all

**round** $1$:
    $p_i.sr(1) \leftarrow p_i.s(1)$
    **if** $\nexists S \in \mathcal{S}_\Pi : S \subseteq p_i.sr(1)$
    **then decide** $[\bot, \ldots, \bot]$
    **else for** each message $m \in p_i.M(1)$, $p_k \in \Pi$:
            if $(p_i.A\,[k] = \bot)$ $p_i.A\,[k] \leftarrow m.A\,[k]$
    $p_i.A_p(1) \leftarrow p_i.A$
    **send** $p_i.A$ to all

**round** $r$: $2 \leq r \leq t$:
    $p_i.sr(r) \leftarrow p_i.s(r) \setminus \{p_j : \exists m \in p_i.M(r)$ :
                $(m.from = p_j) \wedge (p_i.A_p(r-2) \nsubseteq m.A)\}$
    **if** $\vee p_i.s(r) \nsubseteq p_i.s(r-1)$
      $\vee$ $\nexists S \in \mathcal{S}_\Pi : S \subseteq p_i.sr(r)$
    **then decide** $[\bot, ..., \bot]$
    **else** for each message $m \in p_i.M(r)$, $p_k \in \Pi$:
            **if** $(p_i.A\,[k] = \bot)$ $p_i.A\,[k] \leftarrow m.A\,[k]$
    $p_i.A_p(r) \leftarrow p_i.A$
    **send** $p_i.A$ to all

**round** $t+1$:
    $p_i.sr(t+1) \leftarrow p_i.s(t+1) \setminus \{p_j : \exists m \in p_i.M(t+1)$ :
                $(m.from = p_j) \wedge (p_i.A_p(t-1) \nsubseteq m.A)\}$
    **if** $\vee$ $p_i.s(t+1) \nsubseteq p_i.s(t)$
      $\vee$ $\nexists S \in \mathcal{S}_\Pi : S \subseteq p_i.sr(t+1)$
    **then decide** $[\bot, ..., \bot]$
    **else** for each message $m \in p_i.M(t+1)$, $p_k \in \Pi$:
            **if** $(p_i.A\,[k] = \bot)$ $p_i.A[k] \leftarrow m.A[k]$
    $p_i.A_p(t+1) \leftarrow p_i.A$
    **decide** $p_i.A$
```

**Figure A.1:** FFS-ROC: Algorithm run by process $p_i$.

```
Algorithm FFS-WLE
repeat {
$p_i$.elected $\leftarrow$ FALSE
Phase 1:
   Run FFS-ROC with
         $p_i.a \leftarrow i.$
   if ($p_i.d = [\perp, \ldots, \perp]$) then stop
Phase 2:
   Run FFS-ROC with
         $p_i.a \leftarrow p_i.d$ from Phase 1.
   if ($p_i.d = [\perp, \ldots, \perp]$) then stop
   let $x$ be a value of $p_i.d\,[1 \ldots n]$
         such that $p_i.d\,[x] \neq [\perp, \ldots, \perp]$
         and it has the least number of non-$\perp$ values
   if ($p_i$ is the first index of $x$ such that $x[i] \neq \perp$)
         then $p_i$.elected $\leftarrow$ TRUE
}
```

**Figure A.2:** FFS-WLE: Algorithm run by process $p_i$.

- if $p_i$ sends at least one message $m_i$ to some process $p_j$ in round $r$, $0 \leq r \leq t$, and $p_j$ receives $m_i$ by the end of round $r$;

- if $p_i$ decides in round $r$, $r = t + 1$.

We use *Alive*$(r)$ to denote the processes that are alive in round $r$. For an execution $E$ of FFS-ROC we define the following to use in the proofs of this section:

- $T^i_\phi(i, r) \in T$ denotes the value of *Time*$(s_f)$, where $p_i \in$ *Alive*$(r)$ and $s_f \in S$ is the first step $p_i$ executes of round $r$, $r \in \{0, \ldots, t + 1\}$. If $p_i$ executes no steps in $r$, then $T^i_\phi(i, r)$ is undefined;

- $T^u_\phi(i, r) \in T$ denotes the value of *Time*$(s_m)$, where $p_i \in$ *Alive*$(r)$ and $s_m \in S$ is the first step of round $r$ in which $p_i$ sends a message, $0 \leq r \leq t$. If $p_i \notin$ *Alive*$(r)$, then $T^u_\phi(i, r)$ is undefined;

- $T^u_\phi(i, t+1) \in T$ denotes *Time*$(s_d)$, where $p_i \in$ *Alive*$(t+1)$ and $s_d \in S$ is the step in which $p_i$ decides in round $t+1$. If $p_i \notin$ *Alive*$(r)$, then $T^u_\phi(i, t+1)$ is undefined;

- $M_i^r$ is a list of $n$ values, one for each process in $\Pi$ such that the following holds:

$$M_i^r[j] \begin{cases} p_j.a, & \exists m \in p_i.M(r) : m.A[j] \neq \perp \\ \perp, & \text{otherwise} \end{cases}$$

Processes that are alive in a round $r$ may send messages to a strict subset of $\Pi$ when they fail in $r$. Thus, in executions in which processes fail, we have that processes may have a different knowledge of the initial values. For example, if in round zero, a process $p_\ell$ sends a message to a non-faulty process $p_i$, but it crashes before sending a message to a non-faulty process $p_j$, then $p_i.A[\ell] = p_\ell.a$ and $p_j.A[\ell] = \perp$. For the purpose of analyzing these cases, we define a *process chain* (or simply a chain) $\omega_\ell = (i_0 \circ i_1 \circ \ldots \circ i_k)_\ell$, $k \leq t+1$, to be a string over the set of process identifiers. Let $\omega_\ell[x]$ be the process identifier at position $x$ of the chain $\omega_\ell$. The following holds for a process chain $\omega_\ell$:

1. $\omega_\ell[r] \neq \omega_\ell[r']$, if $r \neq r'$;
2. If $\omega_\ell[r] = i$, then $(p_i.A_p(r)[\ell] \neq \perp) \wedge (\forall r' \in \{x \in \mathbb{Z}^* : x \leq r-1\} : p_i.A_p(r')[\ell] = \perp)$;
3. If $\omega_\ell[r] = i$, $r > 0$, then $\exists m \in p_i.M(r) : (m.A[\ell] \neq \perp) \wedge (\omega_\ell[r-1] = j) \wedge (m.\mathit{from} = p_j)$;
4. If $\omega_\ell[0] = i$, then $i = \ell$.

We say that a process $p_i$ is in $\omega_\ell$ ($i \in \omega_\ell$) if and only if there exists an index $r$ such that $\omega_\ell[r] = i$. We use process chains in the proofs below to represent the propagation of knowledge in executions with failures.

We show the correctness of FFS-ROC with Proposition A.4.1. We divide the proof of this proposition into several statements (lemmas and theorems). Each of these statements may depend upon others. Thus, we summarize in Figure A.3 the structure of the proof. For each statement, we show the other statements it depends upon.

**Proposition A.4.1** FFS-ROC implements RO consensus.

Proposition
A.4.1: {A.4.20, A.4.21, A.4.22, A.4.23}

Theorems
A.4.20: {}
A.4.21: {A.4.16}
A.4.22: {A.4.5, A.4.15, A.4.17, A.4.18}
A.4.23: {A.4.2, A.4.5, A.4.11, A.4.16, A.4.19}

Lemmas
A.4.2: {}
A.4.3: {A.4.2}
A.4.4: {A.4.2, A.4.3}
A.4.5: {A.4.3, A.4.4}
A.4.6: {A.4.5}
A.4.7: {A.4.4, A.4.6}
A.4.8: {A.4.6}
A.4.9: {}
A.4.10: {A.4.9}
A.4.11: {A.4.2, A.4.4, A.4.5}
A.4.12: {A.4.6, A.4.7, A.4.10, A.4.11}
A.4.13: {A.4.2, A.4.4, A.4.6, A.4.7, A.4.10, A.4.11, A.4.12}
A.4.14: {A.4.2, A.4.4, A.4.12}
A.4.15: {A.4.13}
A.4.16: {A.4.2, A.4.4, A.4.5, A.4.11, A.4.12}
A.4.17: {A.4.13, A.4.16}
A.4.18: {A.4.5, A.4.11, A.4.13, A.4.14, A.4.15, A.4.16, A.4.17}
A.4.19: {A.4.3, A.4.4, A.4.13, A.4.16}

**Figure A.3:** FFS-ROC proof hierarchy.

We prove Proposition A.4.1 with the following lemmas.

**Lemma A.4.2** *Let $E$ be an execution of FFS-ROC, $r$ be a round of $E$, $0 \leq r \leq t+1$, $p_i$ be a process in Alive$(r)$, and $p_j$ be a process in $\Pi$. If $p_i.A_p(r)[j] \neq \perp$, then $p_i.A_p(r) = p_j.a$.*

**Proof:**

We show with an induction on the round numbers $\rho$, $0 \leq \rho \leq t+1$, that for every round $r' \leq r$, if $p_\ell \in$ *Alive*$(\rho)$ and $p_\ell.A_p(r')[j] \neq \perp$, then $p_\ell.A_p(\rho)[j] = p_j.a$. The base case is $\rho = 0$. From the algorithm, in round $0$ a process $p_\ell$ has $p_\ell.A_p(0)[j] = \perp$, if $\ell \neq j$, and $p_\ell.A_p(0)[\ell] = p_\ell.a$. Thus, $p_\ell.A_p(0)[j] \neq \perp$ only if $\ell = j$, and $p_\ell.A_p(0)[\ell] = p_\ell.a$.

Now suppose that for every $p_\ell \in$ *Alive*$(\rho)$ if $p_\ell.A_p(\rho)[j] \neq \perp$, then $p_\ell.A_p(\rho)[j] = p_j.a$. We show that for every $p_\ell \in$ *Alive*$(\rho+1)$, if $p_\ell.A_p(\rho+1)[j] \neq \perp$, then $p_\ell.A_p(\rho+1)[j] = p_j.a$. By the algorithm, if $p_\ell \in$ *Alive*$(\rho)$ is such that

$p_\ell.A_p(\rho)[j] = p_j.a$, then for every message $m$ it sends at round $\rho$, $m.A[j] = p_j.a$. For $p_{\ell'} \in$ *Alive*$(\rho+1)$, if $p_{\ell'}.A[j] \neq\perp$ at $T^i_\phi(\ell', \rho+1)$, then $p_{\ell'}.A_p(\rho)[j] \neq\perp$ and must be equal to $p_j.a$ by the induction hypothesis and the algorithm. Otherwise, for every message $m$ it receives such that $m.A[j] \neq\perp$, $m.A[j] = p_j.a$. Thus, by the algorithm, if $p_{\ell'}$ receives at least one such a message, it sets $p_{\ell'}.A[j]$ to $p_j.a$, and we have that $p_{\ell'}.A_p(\rho+1)[j] = p_j.a$.

From the previous induction, we conclude that if $p_i.A_p(r)[j] \neq\perp$, then $p_i.A_p(r)[j] = p_j.a$.

□

**Lemma A.4.3** *Let $E$ be an execution of* **FFS-ROC**, *$r$ be a round of $E$, $0 < r \leq t+1$, and $p_i$ be a process in Alive$(r)$. For every message $m \in p_i.M(r)$ such that $m.A[\ell] \neq\perp$, for some $p_\ell \in \Pi$, $m.A[\ell] = p_\ell.a$.*

**Proof:**

By Lemma A.4.2, for every $p_j \in$ *Alive*$(r-1)$, if $p_j.A_p(r-1)[\ell] \neq\perp$, then $p_j.A_p(r-1)[\ell] = p_\ell.a$. If $p_j$ sends a message $m_j$ to $p_i$ in round $r-1$, then $m.A[\ell] = p_\ell.a$. We conclude that for every $m \in p_i.M(r)$ such that $m.A[\ell] \neq\perp$, $m.A[\ell] = p_\ell.a$.

□

**Lemma A.4.4** *Let $E$ be an execution of* **FFS-ROC** *and $r$ be a round of $E$, $0 < r \leq t+1$. For every $p_i \in$ Alive$(r)$, $M^r_i = p_i.A_p(r)$.*

**Proof:**

By the algorithm, if $p_i \in$ *Alive*$(r)$, then for every $j \in [1 \ldots n]$, such that $p_i.A[j] =\perp$ at $T^i_\phi(i, r)$, $p_i$ sets $p_i.A[j]$ to a value $v \in V = \{v : v = m.A[j] \wedge m \in p_i.M(r) \wedge m.A[j] \neq\perp \ \}$, $j \in [1 \ldots n]$ at $t$ if $V \neq \emptyset$, where $T^i_\phi(i, r) \leq t \leq T^u_\phi(i, r)$, $t =$ *Time*$(s)$, and $s$ is a step of $p_i$ that updates $p_i.A[j]$ in round $r$. Otherwise, if $p_i.A[j] \neq\perp$ at $T^i_\phi(i, r)$, then no step of $p_i$ in round $r$ modifies the value of $p_i.A[j]$, and $p_i.A_p(r)[j] = p_i.A_p(r-1)[j]$.

Let $p_j$ be a process of $\Pi$. By Lemma A.4.3, we have that $V = \{v : v = m.A[j] \wedge m \in p_i.M(r) \wedge m.A[j] \neq\perp\}$, $j \in [1 \ldots n]$, is either empty or contains a single value. If $|V| = 1$, then $V = \{p_j.a\}$. There are two cases to consider: 1) $p_i.A_p(r-1)[j] \neq \perp$; 2) $p_i.A_p(r-1) = \perp$.

If $p_i.A_p(r-1)[j] \neq \perp$, then, by the algorithm, $p_i$ does not modify the value of $p_i.A[j]$ in round $r$. By Lemma A.4.2, $p_i.A_p(r-1)[j] = p_j.a$. By the algorithm, $p_i$ sends $p_i.A_p(r-1)$ to itself. Finally, by Lemma A.4.3, every message $m \in p_i.M(r)$, $m.A[j] \neq \perp$, is such that $m.A[j] = p_j.a$. We then have that $p_i.A_p(r)[j] = M_i^r[j]$.

If $p_i.A_p(r-1) = \perp$ and $V = \{p_j.a\}$, then $p_i.A_p(r)[j] = p_j.a$. If $V = \emptyset$, then $p_i.A_p(r)[j] = \perp$. In both cases, $p_i.A_p(r)[j] = M_i^r[j]$.

We conclude that $p_i.A_p(r)$ must be equal to $M_i^r$.

□

**Lemma A.4.5** *Let $E$ be an execution of FFS-ROC. For every $r \in \{z \in \mathcal{R} : z \leq t+1\}$, Correct$(E) \subseteq$ Alive$(r)$.*

**Proof:**

By definition, a process is alive in round $t+1$ of $E$ if it neither crashes nor stops before deciding in this round. We show this claim by showing that for every $p_c \in$ *Correct*$(E)$ and every $r \in \{x \in \mathcal{R} : x \leq t+1\}$, $p_c \in$ *Alive*$(r)$. Let $p_c$ be a process in *Correct*$(E)$. By definition, $p_c$ does not crash in any round. It remains to show that, for every $p_c \in$ *Correct*$(E)$ and every $r \in \{z \in \mathcal{R} : z \leq t+1\}$, $p_c$ does not stop in $r$. We show this with an induction on the round numbers $\rho$, $\rho \in \{z \in \mathcal{R} : z \leq t+1\}$.

By the algorithm, no process stops in round $0$. At round $1$, a process only stops if it does not receive messages from a survivor set. Let $p_c$ be a process in *Correct*$(E)$. By definition, there is a survivor set $S_c$ containing only correct processes, and every process $p_{c'} \in S_c$ sends a message to $p_c$ in round $0$. By C-liveness, $p_c$ must have messages in $p_c.M(1)$ at least from the processes in $S_c$. That is, $S_c \subseteq p_c.s(1)$. Consequently, $p_c$ does not stop in round $1$.

Now suppose that the claim holds for every $\rho$, $\rho \in \{z \in \mathcal{R} : 1 \leq z \leq t\}$, and we show for $\rho+1$. Let $p_c$ be a process in *Correct*$(E)$. By assumption, if $p_c$ does not receive a message from some process $p_i$ in round $\rho$, then $p_i$ must have crashed by round $\rho$. This implies that all processes in $\Pi \setminus p_c.s(\rho)$ crashed by round $\rho$, and $p_c.s(\rho+1)$ therefore cannot contain a process $p_i$ that is not in $p_c.s(\rho)$. Consequently, $p_c.s(\rho+1) \subseteq p_c.s(\rho)$.

For the induction step, it remains to show that $p_c.sr(\rho+1)$ contains some survivor set. From the algorithm, a process $p_i$ is in $p_c.sr(\rho+1)$ if there is a message $m_i \in p_c.M(\rho+1)$ and $p_c.A_p(\rho-1) \subseteq m_i.A$. By assumption, no correct process stops by round $\rho$. Thus, for every $p_{c'} \in \mathit{Correct}(E)$, there is a message $m_c \in p_{c'}.M(\rho)$ from $p_c$. By Lemma A.4.3, for every $p_\ell \in \Pi$ such that $m_c.A[\ell] \neq \perp$, we have that $m_c.A[\ell] = p_c.A_p(\rho-1)[\ell] = p_\ell.a$ and $M_{c'}^{\rho}[\ell] = p_\ell.a$. By Lemma A.4.4, we then have that for every $p_{c'} \in \mathit{Correct}(E)$, $p_c.A_p(\rho-1) \subseteq p_{c'}.A_p(\rho)$. By the algorithm and by the assumption that no correct process stops in round $\rho$, for every $p_{c'} \in \mathit{Correct}(E)$, $p_{c'}$ sends a message to $p_c$. By the observation that for every $p_{c'}$, $p_c.A_p(\rho-1) \subseteq p_{c'}.A_p(\rho)$, we have that $\mathit{Correct}(E) \subseteq p_c.sr(\rho+1)$. By assumption, there is a survivor set $S_c \in \mathcal{S}_\Pi$ such that $S_c \subseteq \mathit{Correct}(E)$. We conclude that $S_c \subseteq p_c.sr(\rho+1)$.

This concludes the proof of the lemma.

□

**Lemma A.4.6** *Let $E$ be an execution of* **FFS-ROC** *such that $\omega_\ell$ is a chain in $E$, $1 \leq |\omega_\ell| \leq t+1$. If $\omega_\ell[r]$ is the identifier of a correct process for some $r$, then $M_j^{r+1}[\ell] \neq \perp$ for every process $p_j \in \mathit{Correct}(E)$.*

**Proof:**

Let $r$ be an index such that $\omega_\ell[r]$ is the identifier of a correct process in $E$ and $i$ be the process identifier in $\omega_\ell[r]$. By the definition of a process chain, we have that $(p_i.A_p(r)[\ell] \neq \perp) \wedge (\forall r' \in \{x \in \mathcal{R} : x \leq r-1\} : p_i.A_p(r')[\ell] = \perp)$. By the algorithm, process $p_i$ sends $p_i.A_p(r)$ to all the processes in $\Pi$. By Lemma A.4.5, every correct process decides in $E$ ($\mathit{Correct}(E) \subseteq \mathit{Alive}(t+1)$). By C-liveness, for every $p_j \in \mathit{Correct}(E)$, there is $m_i \in p_j.M(r+1)$ such that $m_i.A[\ell] \neq \perp$. Again by the algorithm, $M_j^{r+1}[\ell]$ must be different than $\perp$ for every correct process $p_j$ in $E$.

□

**Lemma A.4.7** *Let $E$ be an execution of* **FFS-ROC** *such that there is a chain $\omega_\ell$ of length at least three in $E$. There are no three correct processes $p_{c_1}, p_{c_2}, p_{c_3}$ such that $c_1, c_2, c_3 \in \omega_\ell$.*

**Proof:**

Proof by contradiction. Suppose that there are three processes $p_{c_1}, p_{c_2}, p_{c_3} \in \mathit{Correct}(E)$ such that $c_1, c_2, c_3 \in \omega_\ell$, and that $r$ is the smallest index such that $\omega_\ell[r]$ is the identifier of a correct process. Observe that $|\omega_\ell|$ must be at least as large as $r+3$ ($|\omega_\ell| \geq r+3$), otherwise the claim is vacuously true.

Without loss of generality, let $\omega_\ell[r] = c_1$. By Lemma A.4.6, for every correct process $p_c$ in $\mathit{Correct}(E)$ we have that $M_c^{r+1}[\ell]$ different than $\perp$ and by Lemma A.4.4 $p_c.A_p(r+1)[\ell] = M_c^{r+1}[\ell]$. Consequently, we have that $p_{c_2}.A_p(r+1)[\ell] \neq \perp$ and $p_{c_3}.A_p(r+1)[\ell] \neq \perp$. By the definition of a process chain, $c_2$ and $c_3$ cannot be both in $\omega_\ell$, a contradiction.

□

**Lemma A.4.8** *Let $E$ be an execution of* **FFS-ROC** *such that there is a chain $\omega_\ell$ of length at least three. There is no two correct processes $p_i$, $p_j$ such that $i, j \in \omega_\ell$ and $\omega_\ell = (\omega' \circ i \circ \omega \circ j \circ \omega'')_\ell$, where $\omega, \omega', \omega''$ are substrings of $\omega_\ell$ and $\omega$ is not the empty string.*

**Proof:**

Proof by contradiction. Suppose that there are two correct processes $p_i$ and $p_j$ in $E$ such that $\omega_\ell[r] = i$ and $\omega_\ell[r'] = j$, $r+1 < r'$. By Lemma A.4.6 and by the definition of a process chain, for every correct process $p_j$ in $\mathit{Correct}(E)$, $M_j^{r+1}[\ell] \neq \perp$. We hence have that $r'$ must be equal to $r+1$, and $\omega$ must be empty, contradicting out initial assumption that $r+1 < r'$.

□

**Lemma A.4.9** *Let $E$ be an execution of* **FFS-ROC** *and $p_i$ be a process in $\mathit{Alive}(r)$, $r \geq 2$, such that $p_i.A_p(r)[\ell] \neq \perp$ and for all $r' \in \{x \in \mathcal{R} : x \leq r-1\}$, $p_i.A_p(r')[\ell] = \perp$.*

*For every round $\rho \in \{x \in \mathcal{R} : 1 \leq x \leq r\}$, there are processes $p_{j_1} \in$ Alive$(\rho)$ and $p_{j_2} \in$ Alive$(\rho - 1)$ such that the following holds:*

1. $p_{j_1}.A_p(\rho)[\ell] \neq \perp$, *and for all* $\rho' \in \{x \in \mathcal{R} : x \leq \rho - 1\}$, $p_{j_1}.A_p(\rho')[\ell] = \perp$;
2. $p_{j_2}.A_p(\rho - 1)[\ell] \neq \perp$, *and for all* $\rho' \in \{x \in \mathcal{R} : x \leq \rho - 2\}$, $p_{j_2}.A_p(\rho')[\ell] = \perp$;
3. $\exists m_{j_2} \in p_{j_1}.M(\rho) : (m_{j_2}.A[\ell] \neq \perp)$.

**Proof:**

We now show with an induction on the values of $\psi$, $0 \leq \psi \leq r - 1$, that for every round $\rho = r - \psi$, the claim holds.

The base case is $\psi = 0$, $\rho = r$. By assumption, $p_i$ is such that $p_i.A_p(r)[\ell] \neq \perp$ and for all $r' \in \{x \in \mathcal{R} : x \leq r - 1\}$, $p_i.A_p(r')[\ell] = \perp$. This implies that $M_i^r[\ell] \neq \perp$. From the algorithm, we have that $p_i.s(\rho) \subseteq p_i.s(\rho - 1) \subseteq \ldots \subseteq p_i.s(0)$. Consequently, there must be some process $p_j \in$ *Alive*$(\rho - 1)$ such that the following holds: A) $p_j.A_p(r - 1) \neq \perp$; B) $p_j.A_p(\rho') = \perp$ for all $\rho' \in \{x \in \mathcal{R} : x \leq r - 2\}$; C) $\exists m \in p_i.M(r) : (m.A[\ell] \neq \perp) \wedge (m.\mathit{from} = p_j)$. If there is no such a process $p_j$ that satisfies both A) and C), then $p_i.A_p(r) = \perp$, contradicting our initial assumption. By the algorithm, once $p_j$ sets the value of $p_j.A[\ell]$ to a value different than $\perp$, then the value of $p_j.A[\ell]$ does not change in subsequent rounds. This implies that for all $\rho'$, $0 \leq \rho' < r - 1$, $p_j.A_p(\rho')$ must be equal to $\perp$, because by the algorithm $p_i$ receives a message from $p_j$ in every round ($p_i.s(\varrho) \subseteq p_i.s(\varrho - 1)$, $r \geq \varrho > 0$) and $p_i.A_p(\rho'')$ is different than $\perp$ otherwise, for some $\rho'' < r$.

Suppose the claim is true for $\psi < r - 1$. We show for $\psi + 1$. If it is true for $\psi$, then there is a process $p_{j_1}$ alive in $\rho = r - (\psi + 1)$ such that $p_{j_1}.A_p(\rho)[\ell] \neq \perp$, and for all $\rho' \in \{x \in \mathcal{R} : x \leq \rho - 1\}$, $p_{j_1}.A_p(\rho')[\ell] = \perp$. From the algorithm, we have that $p_{j_1}.s(\rho) \subseteq p_{j_1}.s(\rho - 1) \subseteq \ldots \subseteq p_{j_1}.s(0)$. Consequently, there must be some process $p_{j_2} \in$ *Alive*$(\rho - 1)$ such that the following holds: A) $p_{j_2}.A_p(\rho - 1) \neq \perp$; B) $p_{j_2}.A_p(\rho') = \perp$ for all $\rho' \in \{x \in \mathcal{R} : x \leq \rho - 2\}$; C) $\exists m \in p_{j_1}.M(\rho) : (m.A[\ell] \neq \perp) \wedge (m.\mathit{from} = p_{j_2})$. If there is no such a process $p_{j_2}$ that satisfies both A) and C), then $p_{j_1}.A_p(\rho) = \perp$, contradicting our assumption that the hypothesis hold for $\psi$. By the algorithm, once $p_{j_2}$ sets the value of $p_{j_2}.A[\ell]$ to a value different than $\perp$, then the value of $p_{j_2}.A[\ell]$ does not change in subsequent rounds. This implies that for all $\rho'$, $0 \leq \rho' < \rho - 1$, $p_{j_2}.A_p(\rho')$ must be equal to $\perp$, because by the algorithm $p_{j_1}$ receives a message from $p_{j_2}$ at every round ($p_{j_1}.s(\varrho) \subseteq p_{j_1}.s(\varrho - 1)$, $\rho \geq \varrho > 0$) and $p_i.A_p(\rho'')$ is different than $\perp$ otherwise, for some $\rho'' < \rho$.

This concludes the proof of the lemma.

□

**Lemma A.4.10** *Let $E$ be an execution of FFS-ROC and $p_i$ be a process that is alive in round $r$ of $E$, $r \geq 0$, such that $p_i.A_p(r)[\ell] \neq \perp$ and for all $r' \in \{x \in \mathcal{R} : x \leq r - 1\}$, $p_i.A_p(r')[\ell] = \perp$. There is a chain $\omega_\ell$ such that $|\omega_\ell| = r + 1$, and $\omega_\ell[r] = i$.*

**Proof:**

We have to build a chain $\omega_\ell$ such that $|\omega_\ell| = r + 1$, and $\omega_\ell[r] = i$.

We build such a chain $\omega_\ell$ as follows:

$$\begin{aligned}
&\omega_\ell[0] = \ell \\
&\omega_\ell[\rho] = j \quad , \quad \wedge(0 < \rho < r) \\
&\qquad\qquad\qquad \wedge(p_j.A_p(\rho) \neq \perp) \\
&\qquad\qquad\qquad \wedge(\forall \rho' \in \{x \in \mathcal{R} : x \leq \rho - 1\} : p_j.A_p(\rho') = \perp) \\
&\qquad\qquad\qquad \wedge \exists m_j \in p_{\omega_\ell[\rho+1]}.M(\rho + 1) \text{ from } p_j \\
&\omega_\ell[r] = i
\end{aligned}$$

We can easily verify that $\omega_\ell$ satisfies the properties of a process chain. It remains to show that it is a valid construction.

By the algorithm, we have that $\omega_\ell[0] = \ell$. By Lemma A.4.9, we have that for every $\rho$, $0 < \rho < r$, there is a process $p_j$ that satisfies the properties we stated above. Finally, by assumption, $p_i$ is such that $p_i.A_p(r)[\ell] \neq \perp$ and for all $r' \in \{x \in \mathcal{R} : x \leq r - 1\}$, $p_i.A_p(r')[\ell] = \perp$.

This concludes the proof of the lemma.

□

**Lemma A.4.11** *Let $E$ be an execution of FFS-ROC. If $p_i, p_j \in$ Correct$(E)$, then $p_i.d = p_j.d$ in $E$.*

**Proof:**

By Lemma A.4.5, every correct process decides in $E$ (no correct process stops). Now let $r$, $0 \leq r \leq t$, be a round in which no process crashes. Such a round exists in $E$ by assumption (no more than $t$ processes can fail in an execution, where $t$ is $|\Pi|$ subtracted the size of the smallest survivor set).

We first show by induction on the values of $\rho$, $r+1 \leq \rho \leq t+1$, the following proposition:

$$\bigwedge \ \forall p_{c_1}, p_{c_2} \in \mathit{Correct}(E) : p_{c_1}.A_p(\rho) = p_{c_2}.A_p(\rho)$$
$$\bigwedge \ \forall p_\ell \in \mathit{Alive}(\rho), p_c \in \mathit{Correct}(E) : p_\ell.A_p(\rho) \subseteq p_c.A_p(\rho)$$

The base case is $\rho = r + 1$. According to the algorithm, every process $p_i$ that is alive in round $r$ sends a message containing $p_i.A_p(r)$ to every other process. According to C-liveness and the assumption that no process crashes in round $r$, for every process $p_c \in \mathit{Correct}(E)$, $p_c.s(r+1) = \mathit{Alive}(r)$. This implies that for every $p_{c_1}, p_{c_2} \in \mathit{Correct}(E)$, $M_{c_1}^{r+1} = M_{c_2}^{r+1}$. By Lemma A.4.4, $M_c^{r+1} = p_c.A_p(r+1)$ for every $p_c \in \mathit{Correct}(E)$. This implies that for every $p_{c_1}, p_{c_2} \in \mathit{Correct}(E)$, $p_{c_1}.A_p(r+1) = p_{c_2}.A_p(r+1)$.

It remains to show the second part of the proposition for the base case. Let $p_\ell$ be a process in $\mathit{Alive}(r+1)$ and $p_c$ be a process in $\mathit{Correct}(E)$. By the failure assumptions, we have that $p_\ell.s(r+1) \subseteq \mathit{Alive}(r)$. This implies that $p_\ell.s(r+1) \subseteq p_c.s(r+1)$. If $p_\ell.s(r+1) \subseteq p_c.s(r+1)$, then $M_\ell^{r+1} \subseteq M_c^{r+1}$. By Lemma A.4.4, $M_\ell^{r+1} = p_\ell.A_p(r+1)$ and $M_c^{r+1} = p_c.A_p(r+1)$, which implies that $p_\ell.A_p(r+1) \subseteq p_c.A_p(r+1)$. This concludes the proof of the base case.

Suppose that the proposition holds for every $\rho < t+1$. We show for $\rho + 1$. By the induction hypothesis, the algorithm, and Lemma A.4.2, for every process

$p_c \in \textit{Correct}(E)$, $p_c.A_p(\rho) = M_c^{\rho+1}$. By Lemma A.4.4, for every $p_c \in \textit{Correct}(E)$, $p_c.A_p(\rho+1) = M_c^{\rho+1}$. We conclude that for every $p_{c_1}, p_{c_2} \in \textit{Correct}(E)$, $p_{c_1}.A_p(\rho+1) = p_{c_2}.A_p(\rho+1)$.

By our failure assumptions, a faulty process may receive an arbitrary subset of the messages sent to it in a round. Let $p_\ell$ be a process in *Alive*$(\rho+1)$ and $p_c$ be a process in *Correct*$(E)$. By the induction hypothesis, the algorithm, and Lemma A.4.2, $M_\ell^{\rho+1} \subseteq M_c^{\rho+1}$. By Lemma A.4.4, $p_\ell.A_p(\rho+1) = M_\ell^{\rho+1}$ and $p_c.A_p(\rho+1) = M_c^{\rho+1}$. We conclude that $p_\ell.A_p(\rho+1) \subseteq p_c.A_p(\rho+1)$. This concludes the proof of the induction step.

From the previous proposition, we have that $p_i.A_p(t+1) = p_j.A_p(t+1)$. By the algorithm, $p_i$ decides upon $p_i.A_p(t+1)$ and $p_j$ decides upon $p_j.A_p(t+1)$. Consequently, $p_i.d = p_j.d$. This concludes the proof of the lemma.

□

**Lemma A.4.12** *Let $E$ be an execution of FFS-ROC and $p_i, p_j$ be two processes in Alive$(t+1)$, and $p_\ell$ be a process in $\Pi$. If $(p_i.A_p(t+1)[\ell] \neq \perp)$ and for all $r \in \{x \in \mathcal{R} : x \leq t\}$, $(p_i.A_p(r)[\ell] = \perp)$, then $p_j.d[\ell] \neq \perp$.*

**Proof:**

By the algorithm, once $p_j$ sets the value of $p_j.A[\ell]$ to a value different than $\perp$, $p_j$ does not change it in subsequent rounds. Thus, we only need to show that there is some round $r$ in which $p_j$ sets $p_j[\ell]$ to a value different than $\perp$.

Suppose that:

$$(p_i.A_p(t+1)[\ell] \neq \perp) \wedge (\forall r \in \{z \in \mathcal{R} : z \leq t\} : p_i.A_p(r)[\ell] = \perp)$$

Assuming that $p_i$ and $p_j$ can be either correct or faulty, there are four possible cases, and we analyze each case separately as follows:

- $p_i$ and $p_j$ are correct in $E$. By Lemma A.4.11, we have that $p_i.d = p_j.d$;
- $p_i$ is faulty and $p_j$ is correct in $E$. If $p_i$ decides in $E$, then $p_i$ is in *Alive*$(t+1)$. By Lemma A.4.10, there is a chain $\omega_\ell$ such that $|\omega_\ell| = t+2$, and $\omega_\ell[t+1] = i$. Since

there are at most $t$ failures by assumption, there is at least one correct process in $\omega_\ell$. Moreover, such correct process must be in a position $r$ of the chain such that $0 \leq r \leq t$. Thus, $p_j.A_p(t+1)[\ell]$ must be different than $\perp$, by Lemma A.4.6 and the algorithm;

- $p_i$ is correct and $p_j$ is faulty in $E$. If $p_i$ is correct, then, by Lemma A.4.10, there is a chain $\omega_\ell$ such that $|\omega_\ell| = t+2$, and $\omega_\ell[t+1] = i$. Because $p_i$ is correct, one of the following two must happen: 1) for every $r$, $0 \leq r \leq t-1$ and $x = \omega_\ell[r]$, $p_x$ crashes in round $r$ of $E$; 2) $p_i.A_p(r)[\ell] \neq \perp$ for some $r < t+1$ by Lemma A.4.6 (there is a correct process in the chain). Case 1 cannot happen because $p_j \in$ *Alive*$(t+1)$ by assumption, and there are at least $t+1$ faulty processes, violating our assumptions for survivor sets. In case 2, $p_i$ learns the initial value of $p_\ell$ in an earlier round, contradicting our initial assumption. We conclude that this case is hence not possible;

- $p_i$ and $p_j$ are faulty in $E$. By Lemma A.4.10, there is a chain $\omega_\ell$ such that $|\omega_\ell| = t+2$, and $\omega_\ell[t+1] = i$. By Lemma A.4.7, there are at most two correct processes in any chain. Thus, $\omega_\ell$ contains $t$ faulty processes. Consequently, there must be an $r$, $0 \leq r \leq t$, such that $j = \omega_\ell[r]$, and $p_j.A_p(r) \neq \perp$.

From the previous analysis, we have that either $p_i.d = p_j.d$ or $p_j.A_p(t+1) \neq \perp$. By the algorithm, we have $p_j$ decides upon $p_j.A_p(t+1)$, and in both cases $p_j.d[\ell] \neq \perp$. This concludes the proof of the lemma.

□

**Lemma A.4.13** *Let $E$ be an execution of **FFS-ROC**, $p_i$, $p_j$ be two processes in Alive$(t+1)$, and $S_i$, $S_j$ be two survivor sets in $\mathcal{S}_\Pi$ such that for all $r \in \{z \in \mathcal{R} : z \leq t+1\}$, $S_i \subseteq p_i.sr(r)$, $S_j \subseteq p_j.sr(r)$, and $S_i \cap S_j \neq \emptyset$. $p_i.d = p_j.d$ in $E$.*

**Proof:**

By the algorithm, once a process $p_i$ sets the value of $p_i.A[\ell]$ in round $r \in \{z \in \mathcal{R} : z \leq t\}$ to a value different than $\perp$, it does not change it in subsequent rounds. We then have to show that if $p_i$ learns about the initial value of $p_\ell$ in round $r$ (that is, $p_i.A_p(r)[\ell] \neq \perp$ and for all $r' \in \{z \in \mathcal{R} : z \leq r-1\}$, $p_i.A_p(r')[\ell] \neq \perp$), then there is a round $r'$ such that $p_j$ learns the initial value of $p_\ell$ at round $r'$ (that is, $p_j.A_p(r')[\ell] \neq \perp$ and for all $r'' \in \{z \in \mathcal{R} : z \leq r'-1\}$, $p_j.A_p(r'')[\ell] \neq \perp$). By Lemma A.4.2, if $p_i.A_p(r)[\ell] = p_j.A_p(r')[\ell] \neq \perp$, then $p_i.A_p(r)[\ell] = p_j.A_p(r')[\ell] = p_\ell.A_p(0)[\ell]$. We now analyze each case separately.

First, suppose that $((p_i.A_p(t+1)[\ell] \neq \perp) \wedge (\forall r \in \{z \in \mathcal{R} : z \leq t\} : p_i.A_p(r)[\ell] = \perp))$. This follows directly from Lemma A.4.12.

Now, suppose that $(p_i.A_p(t)[\ell] \neq \perp) \wedge (\forall r \in \{z \in \mathcal{R} : z \leq t-1\} : p_i.A_p(r)[\ell] = \perp)$:

- $p_i$ and $p_j$ are correct in $E$. By Lemma A.4.11, we have that $p_i.d = p_j.d$.

- $p_i$ is faulty and $p_j$ is correct in $E$. From Lemma A.4.10, there is a chain $\omega_\ell$ such that $|\omega_\ell| = t+1$, and $\omega_\ell[t] = i$. Because there are at most $t$ faulty processes by assumption, there must be a correct process in $\omega_\ell$. That is, there must be some $r$, $0 \leq r \leq t-1$, such that $\omega_\ell[r]$ is the identifier of a correct process in $E$. It follows that $p_j.A_p(r+1)$ must be different than $\perp$, by Lemma A.4.6.

- $p_i$ is correct and $p_j$ is faulty in $E$. From Lemma A.4.10, there is a chain $\omega_\ell$ such that $|\omega_\ell| = t+1$, and $\omega_\ell[t] = i$. Because $p_j$ is faulty, either there is some $r$ such that $\omega_\ell[r] = j$ or there are at most $t-1$ faulty processes in $\omega_\ell$. If the former holds, then we are done. If the latter holds, then $\omega_\ell[t-1]$ must be the identifier of a correct process, and there is no $r < t-1$ such that $\omega_\ell[r]$ is the identifier of a correct process. Otherwise there is some $r \in \{z \in \mathcal{R} : z \leq t-1\}$ such that $p_i.A_p(r)[\ell] \neq \perp$ (by Lemma A.4.6). In addition, because $\omega_\ell$ contains $t-1$ faulty processes and $p_j$ is faulty, any $p_x \in (S_i \cap S_j)$ is either correct or is in the chain $\omega_\ell$. Thus, $p_x.A_p(t) \neq \perp$, for every $p_x \in (S_i \cap S_j)$. Since by assumption $S_j \subseteq p_j.sr(r)$ for every $r \in \{z \in \mathcal{R} : z \leq t+1\}$, we have that $p_j.A_p(t+1) \neq \perp$, by the algorithm and Lemma A.4.4;

- $p_i$ and $p_j$ are faulty in $E$. From Lemma A.4.10, there is a chain $\omega_\ell$ such that $|\omega_\ell| = t+1$, and $\omega_\ell[t] = i$. Because $\omega_\ell$ contains exactly $t+1$ process identifiers, at least one must be correct, and by Lemma A.4.7, at most two correct processes. We then have that either there is some $r$ such that $\omega_\ell[r] = j$ or $\omega_\ell[r] \neq j$ for every $r$. In the former case, we have that $p_j.A_p(r)[\ell] \neq\perp$ for some $r < t$. In the latter, $\omega_\ell$ must contain $t-1$ faulty processes (at most two correct processes and $p_j$ is not in $\omega_\ell$). Let $p_x$ be a process in $S_i \cap S_j$. $p_x$ is either correct or faulty. Suppose $p_x$ is correct. Because there is some correct process in $\omega_\ell[r]$, for $r \in \{z \in \mathcal{R} : z \leq t-1\}$, by Lemma A.4.6, it must be the case that $p_x.A_p(t-1)[\ell] \neq\perp$ and consequently $p_j.A_p(t)[\ell] \neq\perp$, by the algorithm and Lemma A.4.4.

Now suppose that $p_x$ is faulty. In this case, either there is $r \in \{z \in \mathcal{R} : z \leq t\}$ such that $\omega_\ell[r] = x$ or $x = j$. The case that $\omega_\ell[r] = x$ is straightforward. If $x = j$, then either $\{p_j\} = S_j$ or $\{p_j\} \subset S_j$. Suppose the former. If $S_j$ is a singleton set, then $t = |\Pi| - 1$ and $t + 1 = |\Pi|$. In this case, all the processes in $\Pi$ must be in the chain $\omega_\ell$, and hence it is must be the case that $p_j \in \omega_\ell$. Thus, $S_j$ must contain at least two processes. Let $p_{j'}$ be a process in $S_j$ such that $j \neq j'$. We then have that either $j' \in \omega_\ell$ or $p_{j'} \in \mathit{Correct}(E)$. In either case, there must be some round $r \in \{z \in \mathcal{R} : z \leq t\}$ such that $p_j.A_p(r)[\ell] \neq\perp$, and $p_j.A_p(t+1)[\ell] \neq\perp$ by the algorithm.

Finally, suppose that $\exists r \in \{z \in \mathcal{R} : z \leq t-1\} : (p_i.A_p(r)[\ell] \neq\perp) \wedge (\forall r' \in \{z \in \mathcal{R} : z \leq r-1\} : p_i.A_p(r')[\ell] =\perp)$. By assumption $S_i \subseteq p_i.sr(\rho)$ for all $\rho \in \{z \in \mathcal{R} : z \leq t{+}1\}$. This implies by the algorithm that $p_x.A_p(r{+}1) \subseteq p_i.A_p(r{+}2)$, $p_x \in S_i \cap S_j$. Because $S_j \subseteq p_j.sr(\rho)$, for all $\rho \in \{z \in \mathcal{R} : z \leq t+1\}$, and $p_x \in S_j$, we then have by the algorithm and Lemma A.4.4 that $p_j.A_p(r+2)[\ell]$ must be different than $\perp$, and equal to $p_i.A_p(r)[\ell]$ by Lemma A.4.2.

From the previous argument, we conclude that $p_i.A_p(t{+}1) = p_j.A_p(t{+}1)$. By the algorithm, we have that $p_i$ decides upon $p_i.A_p(t{+}1)$ and $p_j$ decides upon $p_j.A_p(t{+}1)$. Again by the algorithm, we have that $p_i.d = p_j.d$.

□

**Lemma A.4.14** *Let $E$ be an execution of* ***FFS-ROC*** *and $p_i, p_j$ be two processes in*

*Alive*$(t+1)$ *such that* $p_j \in p_i.sr(r)$ *for every* $r \in \{z \in \mathcal{R} : z \leq t+1\}$. $p_j.d \subseteq p_i.d$ *in* $E$.

**Proof:**

By the algorithm, once a process $p_j$ sets the value of $p_j.A[\ell]$ to a value different than $\perp$ in a round $r$, for some $p_\ell \in \Pi$ and some $0 \leq r \leq t+1$, it does not change it in subsequent rounds. If $p_j.A_p(t+1)[\ell] \neq \perp$, then there is some round $\rho$, $0 \leq \rho \leq t+1$, such that $p_j.A_p(\rho)[\ell] \neq \perp$ and for all $\rho' \in \{z \in \mathcal{R} : z \leq \rho - 1\}$, $p_j.A_p(\rho')[\ell] = \perp$. We then have to show that for every $p_\ell \in \Pi$ such that $p_j.A_p(t+1)[\ell] \neq \perp$, there is some $\varrho$ such that $p_i.A_p(\varrho)[\ell] \neq \perp$, $\varrho \in \{z \in \mathcal{R} : z \leq t+1\}$.

Let $p_\ell$ be a process such that $p_j.A_p(\rho)[\ell] \neq \perp$ and for all $\rho' \in \{x : 0 \leq x < \rho\}$, $p_j.A_p(\rho')[\ell] = \perp$. Suppose that $\rho = t+1$. This case follows directly from Lemma A.4.12. Now suppose that $\rho \leq t$. Because $p_j$ sends a message to $p_i$ in every round by assumption, $M_i^{\rho+1}[\ell]$ must be different than $\perp$, and $p_i.A_p(\rho+1)[\ell] = M_i^{\rho+1}[\ell]$ by Lemma A.4.4. We conclude that if $p_j.A_p(t+1)[\ell] \neq \perp$, for some $p_\ell \in \Pi$, then $p_i.A_p(t+1)[\ell] \neq \perp$. By Lemma A.4.2, $p_i.A_p(t+1)[\ell] = p_j.A_p(t+1)[\ell] = p_\ell.a$. By the algorithm, $p_i$ decides upon $p_i.A_p(t+1)$ and $p_j$ decides upon $p_j.A_p(t+1)$. Consequently, $p_j.d \subseteq p_i.d$.

□

**Lemma A.4.15** *Let* $E$ *be an execution of* FFS-ROC. *If* $p_i$, $p_j$, *and* $p_\ell$ *decide in* $E$, *then either* $p_i.d = p_j.d$, $p_i.d = p_\ell.d$, *or* $p_j.d = p_\ell.d$.

**Proof:**

If $p_i$, $p_j$, and $p_\ell$ decide in $E$, then there are survivor sets $S_i$, $S_j$, and $S_\ell$ such that $S_i \subseteq p_i.sr(r)$, $S_j \subseteq p_j.sr(r)$, and $S_\ell \subseteq p_\ell.sr(r)$, for all $r$, $0 \leq r \leq t+1$. By the (3,2)-Intersection property, either $S_i \cap S_j \neq \emptyset$, $S_i \cap S_\ell \neq \emptyset$, or $S_j \cap S_\ell \neq \emptyset$. By Lemma A.4.13, we then have that either $p_i.d = p_j.d$, $p_i.d = p_\ell.d$, or $p_j.d = p_\ell.d$.

□

**Lemma A.4.16** *Let* $E$ *be an execution of* FFS-ROC *and* $p_i$ *be a correct process in* $E$. *If* $p_j$ *decides in* $E$, *then* $p_j.d \subseteq p_i.d$.

**Proof:**

By Lemma A.4.5, $p_i \in \mathit{Alive}(t+1)$ ($p_i$ decides in $E$). If $p_j$ is correct, then the lemma follows from Lemma A.4.11. Now suppose that $p_j$ fails to receive at least one message in $E$. By assumption, both $p_i$ and $p_j$ decide in $E$. By the algorithm, $p_j \in p_i.s(r)$ for every $r$, $0 \leq r \leq t+1$. Because $p_i$ is correct, we have that there is $m$ from $p_j$ in $p_i.M(r)$ for every $r$, $0 \leq r \leq t+1$. By Lemma A.4.4 and the algorithm, we then have that if $p_j.A_p(r)[\ell] \neq \perp$, for some $p_\ell \in \Pi$ and $0 \leq r \leq t$, then $p_i.A_p(r+1)[\ell] = p_j.A_p(r)[\ell]$. It remains to show that if $p_j.A_p(t+1)[\ell] \neq \perp$, and $p_j.A_p(r)[\ell] = \perp$, $p_\ell \in \Pi$, for every $r \in \{z \in \mathcal{R} : z \leq t\}$, then $p_i.A_p(t+1)[\ell] = p_j.A_p(t+1)[\ell]$. By Lemma A.4.12, we have that if $p_j.A_p(t+1)[\ell] \neq \perp$, then $p_i.A_p(t+1)[\ell] \neq \perp$. By Lemma A.4.2, $p_i.A_p(t+1)[\ell] = p_j.A_p(t+1)[\ell] = p_\ell.a$. We conclude that $p_j.d \subseteq p_i.d$.

□

**Lemma A.4.17** *Let $E$ be an execution of* FFS-ROC*. If there are two processes $p_i$ and $p_j$, $p_i, p_j \in \mathit{Alive}(t+1)$, then either $p_i.d \subseteq p_j.d$ or $p_j.d \subseteq p_i.d$.*

**Proof:**

If at least one of $p_i$ and $p_j$ is correct, then the proof follows from Lemma A.4.16. Now suppose both $p_i$ and $p_j$ are faulty. Because both $p_i$ and $p_j$ decide in $E$ by assumption, there are survivor sets $S_i$ and $S_j$ such that $(S_i \subseteq p_i.sr(r)) \wedge (S_j \subseteq p_j.sr(r))$ for every $r$, $0 \leq r \leq t+1$. If $S_i \cap S_j \neq \emptyset$, then the lemma follows because $p_i.d = p_j.d$ by Lemma A.4.13. Suppose now the contrary: $S_i \cap S_j = \emptyset$. By assumption, there must be a survivor set $S_c$ containing only correct processes. By the (3,2)-Intersection property, either $S_i \cap S_c \neq \emptyset$ or $S_j \cap S_c \neq \emptyset$. Let $p_c$ be a process in $S_c$. We then have by Lemma A.4.13 that either $p_i$ and $p_c$ decide upon the same value or $p_j$ and $p_c$ decide upon the same value. Suppose without loss of generality that $p_i$ and $p_c$ decide upon the same value. We hence have from Lemma A.4.16 that $p_j.d \subseteq p_i.d$. This concludes the proof of the lemma.

□

**Lemma A.4.18** *Let $E$ be an execution and* vals *be $\{d : \exists p_i \in \Pi : (p_i.d = d)\} \setminus \mathcal{N}$. For* *every $d_f, d_c \in$ vals, $d_f \subseteq d_c$, there are survivor sets $S_f, S_c \in \mathcal{S}_\Pi$ such that the following properties hold:*

$$\begin{array}{lll} \bigwedge \ \forall p \in S_f : & \vee & p \textit{ crashes} \\ & \vee & p.d = d_f \\ \bigwedge \ \forall p \in S_c : & \wedge & p.d = d_c \\ & \wedge & p \textit{ is not faulty} \end{array}$$

**Proof:**

By Lemma A.4.5, *Correct*$(E) \subseteq$ *Alive*$(t+1)$. By the algorithm, every non-faulty process $p_c$ is such that $p_c.d[c] = p_c.a$. We then have that *vals* contains at least one value. By Lemma A.4.15, there cannot be three different decision values, and if there are two values $d$ and $d'$, then either $d \subseteq d'$ or $d' \subseteq d$ by Lemma A.4.17. We analyze these two cases separately.

**Case 1.** Suppose that *vals* contains a single value, say $d$, and $d_f = d_c = d$. By assumption, there is a survivor set $S'_c$ such that $S'_c$ contains only non-faulty processes. By Lemma A.4.11, every process $p_i \in S'_c$ is such that $p_i.d = d$. If we make $S_c = S_f = S'_c$, then our claim holds.

**Case 2.** Suppose that *vals* contains two distinct values $d_f$ and $d_c$, $d_f \subseteq d_c$. Let $p_i$ be a process such that $p_i \in$ *Alive*$(t+1)$ and $p_i.d = d_f$. By the algorithm, there is survivor set $S_i$ such that $S_i \subseteq p_i.sr(r)$, for every $r \in \{z \in \mathcal{R} : z \leq t+1\}$. Let $p_j$ be a process in $S_i$. If $p_j \in$ *Alive*$(t+1)$, then there is an $S_j \in \mathcal{S}_\Pi$ such that $S_j \subseteq p_j.sr(r)$, for every $r \in \{z \in \mathcal{R} : z \leq t+1\}$. Now let $S'_c$ be a survivor set such that $S'_c \subseteq$ *Correct*$(E)$. By the (3,2)-Intersection property, either $S_j \cap S'_c \neq \emptyset$ or $S_j \cap S_i \neq \emptyset$. Note that $S_i \cap S'_c$ must be empty, otherwise $p_i.d = d_c$ according to Lemma A.4.13, contradicting our initial assumption.

If $S_j \cap S'_c \neq \emptyset$, then by Lemma A.4.13 we have that $p_j.d = d_c$ because $p_j.d = p_c.d$ for every $p_c \in S'_c$ (Lemma A.4.13) and $p_c.d$, $p_c \in S'_c$, must be equal to $d_c$ (Lemma A.4.16). By Lemma A.4.14, however, we have that $p_j.d \subseteq p_i.d$. This implies that $p_i.d = d_c$, again contradicting our initial assumption. It therefore must be the case that $S_i \cap S_j$ is not empty. By Lemma A.4.13, we have that $p_i.d = p_j.d = d_f$.

Now suppose that $p_j \notin \mathit{Alive}(t+1)$. We then have that $p_j$ crashes in $E$, and $p_j.d = \mathcal{N}$. We therefore have that $p_j \in S_i$ either decides upon $d_f$ or crashes in $E$.

It remains to show the second part of the properties in the statement of the lemma. By Lemma A.4.16, every correct process must decide upon $d_c$. Thus, every process $p_c$ in $S$ is such that $p_c.d = d_c$ in $E$.

To conclude, if we make $S_f = S_i$ and $S_c = S'_c$, then our claim holds.

□

**Lemma A.4.19** *Let $E$ be an execution of* FFS-ROC *such that there are survivor sets $S_f, S_c \in \mathcal{S}_\Pi$ and values $v_f, v_c \in V$, $v_f \neq v_c$, such that the following holds:*

$$\begin{array}{l} \bigwedge \ \forall p \in S_f : p.a \in \{v_f, \perp\} \\ \bigwedge \ \forall p \in S_c : p.a = v_c \\ \bigwedge \ \forall p \in S_c : p \textit{ is not faulty} \end{array}$$

*If exists $p_i, p_\ell \in \Pi$ such that $p_i.d[\ell] = v_f$, then for all $p_j \in \mathit{Alive}(t+1)$, $v_f \in p_j.d$.*

**Proof:**

Suppose that $p_i.d[\ell] = v_f$, for some $p_i, p_\ell \in \Pi$. By the algorithm, if a process $p_j$ does not crash or stop in an execution of FFS-ROC, then there is some survivor set $S_j$ such that $S_j \subseteq p_j.sr(r)$ for every $r \in \{z \in \mathcal{R} : z \leq t+1\}$. $S_f$ and $S_c$ must be disjoint, otherwise there is some process with two different initial values. By the (3,2)-Intersection property, either $S_j \cap S_f \neq \emptyset$ or $S_j \cap S_c \neq \emptyset$. If $S_j \cap S_f \neq \emptyset$, then $p_j.A_p(r)[\ell] = M_j^r[\ell] = p_\ell.a = v_f$ (Lemma A.4.4, Lemma A.4.3, and the algorithm). We then have by the algorithm that $p_j.A_p(t+1)[\ell] = p_j.d[\ell] = p_\ell.a = v_f$.

If $S_j \cap S_c \neq \emptyset$, then $p_j.d = p_c.d$ by Lemma A.4.13, for every $p_c \in S_c$. By Lemma A.4.16, $p_i.d \subseteq p_c.d$ for every non-faulty $p_c$. That is, we have that $p_c.d[\ell] = p_i.d[\ell] = v_f$. We then have that $p_j.d[\ell] = p_c.d[\ell] = p_\ell.a = v_f$. This concludes the proof of the lemma.

□

**Theorem A.4.20** *Algorithm* FFS-ROC *satisfies Termination.*

**Proof:**

This is straightforward from the algorithm: every process that does not crash in an execution of FFS-ROC decides in round $t + 1$.

□

**Theorem A.4.21** *Algorithm* FFS-ROC *satisfies Agreement.*

**Proof:**

By Lemma A.4.16, if a process $p_i$ decides in $E$, then $p_i.d \subseteq p_c.d$ for every non-faulty $p_c$. This implies that for every $p_\ell$ such that $p_i.d[\ell] \neq \perp$, we have that $p_c.d[\ell] = p_i.d[\ell]$ for every non-faulty $p_c$.

□

**Theorem A.4.22** *Algorithm* FFS-ROC *satisfies RO Uniformity.*

**Proof:**

By Lemma A.4.5, every correct process decides in $E$. By the algorithm, for every non-faulty process $p_c$, $p_c.d[c] = p_c.a$. Thus, there must be at least one non-$\mathcal{N}$ decision value. By Lemma A.4.15, there cannot be three processes in an execution of FFS-ROC such that each process decides upon a different value. This shows the first statement of the property: $1 \leq |\mathit{vals}| \leq 2$, where $\mathit{vals} = \{d : \exists p_i \in \Pi \text{ s.t. } (p_i.d = d)\} \setminus \mathcal{N}$ in any execution of FFS-ROC. The second statement follows directly from Lemma A.4.17. The third statement follows directly from Lemma A.4.18.

□

**Theorem A.4.23** *Algorithm FFS-ROC satisfies Validity.*

**Proof:**

If $p_i \in Alive(t+1)$ in some execution $E$ of FFS-ROC, then by Lemmas A.4.5 and A.4.16 $p_i.d \subseteq p_c.d$, for every $p_c \in Correct(E)$. By the algorithm, $p_i.d[i]$ must be equal to $p_i.a$. We consequently have that $p_c.d[i]$ must be equal to $p_i.a$. This proves the first statement in the specification of validity.

If $p_i$ crashes in an execution $E$ of FFS-ROC, then by Lemmas A.4.2 and A.4.11 either $p_c.d[i] = \perp$ or $p_c.d[i] = p_i.a$, for every $p_c \in Correct(E)$. This shows the second statement in the definition of validity. The third statement follows directly from Lemma A.4.19.

□

With Theorems A.4.20, A.4.21, A.4.22, and A.4.23, we show that FFS-ROC implements the four RO consensus properties, thereby showing Proposition A.4.1.

## A.5 Correctness of FFS-WLE

Algorithm FFS-WLE proceeds in iterations of an infinite repeat loop. In each iteration, processes execute two phases, and in each phase a process participates in the execution of an algorithm that implements RO consensus. For the following description, we assume that such an algorithm is FFS-ROC. As shown in Figure A.2, a process that does not crash in an execution of FFS-WLE executes infinitely many iterations of the repeat loop. According to our system model, we split an execution of an algorithm into rounds. We further number the iterations of an execution of FFS-WLE and assume that round numbers map to iteration numbers. That is, there is a mapping *Iteration* : $\mathcal{R} \rightarrow \mathcal{I}$, where $\mathcal{R}$ is the set of round numbers as before, and $\mathcal{I} = \mathbb{Z}^*$ is the set of iteration numbers. In addition, we assume that iteration numbers increase monotonically with round numbers, and the number of rounds executed in an iteration is fixed, being a function of the number of rounds in an execution of FFS-ROC.[4] For the purpose of the

[4]Because there are infinitely many executions of FFS-ROC in an execution of FFS-WLE and round numbers monotonically increase with time, the round numbers in the pseudocode for FFS-ROC are relative to the first round

proofs that follow, we only need to assume that each phase executes at least two rounds. Note that FFS-ROC requires $t+1$ rounds, and $t+1 \geq 2$ if $t \geq 1$. We therefore have that each phase must have at least two rounds, assuming systems in which processes can fail. In fact, because processes can fail by crashing and we assume cores and survivor sets to characterize valid sets of faulty processes, we can use the same argument of Section 3.3.2 to show that $t+1$ is a lower bound on the number of rounds.

According to the discussion in the previous paragraph, we associate an iteration number with each iteration of the algorithm in an execution. In the following, we use iteration numbers to refer to iterations of the repeat loop. In proving the correctness of FFS-WLE, we also use the following definitions:

- $\mathit{vals}_i$, $i \in \{1,2\}$ is the set $\{d : p_i.d = d \wedge p_i \in \Pi\} \setminus \mathcal{N}$ after executing FFS-ROC in phase $i$ of some iteration $\zeta$, $\zeta \in \mathcal{I}$;
- a process $p_i$ finishes a phase $\rho \in \{1,2\}$ of some iteration $\zeta$ in an execution $E$ if it neither stops nor crashes before executing the last step of that phase;
- A process $p_i$ starts phase $x \in \{1,2\}$ of an iteration $\zeta$ at time $\tau$ if $p_i$ executes at least one step of phase $x$ of $\zeta$ and the first step $s$ of $p_i$ in phase $x$ of $\zeta$ is such that $\mathit{Time}(s) = \tau$.

As in Section A.3, we assume that FFS-WLE uses a system profile $\langle \Pi, \mathcal{C}_\Pi, \mathcal{S}_\Pi \rangle$ and that this profile satisfies (3,2)-Intersection.

Now we prove the following proposition.

**Proposition A.5.1** FFS-WLE implements Safety, LE-Liveness, and FF-Stability.

We show proposition A.5.1 with the following set of theorems, each one proving a property of weak leader election.

**Theorem A.5.2** *Algorithm FFS-WLE satisfies Safety.*

in which an execution of FFS-ROC starts.

**Proof:**

Let $E$ be an execution of FFS-WLE. We have to show that $|\{p_i \in \Pi : p_i.\textit{elected}\}| < 2$ for every $\tau \in T$. First, we show that in an iteration of $E$, at most one process is elected. By the RO uniformity property of RO consensus, there is at least one decision value and at most two different decision values as a result of phase 1. That is, $1 \leq |\textit{vals}_1| \leq 2$. Suppose $|\textit{vals}_1| = 1$. By the algorithm, every process $p_i$ that finishes phase 2 uses a list $x$ in its decision value $p_i.d$, where $x$ has the minimum number of non-$\perp$ values among all lists in $p_i.d$. $x$ must be the initial value of some process by validity. By assumption, there is a single initial value in phase 2, and consequently $p_i.d[j] \in \{x, \perp\}$, for every $p_j \in \Pi$, implying that no two distinct processes that finish phase 2 can be elected.

Now suppose that $|\textit{vals}_1| = 2$. From RO uniformity, we have that there are values $d_1, d_2 \in \textit{vals}_1$ and $S_1, S_2 \in \mathcal{S}_\Pi$ such that:

$$\begin{array}{lll} \wedge\ \forall p \in S_1 : & \vee & p \text{ crashes} \\ & \vee & p.d = d_1 \\ \wedge\ \forall p \in S_2 : & \wedge & p.d = d_2 \\ & \wedge & p \text{ is not faulty} \end{array}$$

By the algorithm, a process that finishes phase 1 of an iteration executes FFS-ROC once more in phase 2 with its decision value of the previous phase as its initial value. If the above properties hold, then the only processes in $S_1$ that do not have $d_1$ as initial value are the ones that crash before phase 2 starts. Let's call this set $\textit{Crash}_1$. By validity, if some process $p_i$ decides upon a value $p_i.d$ such that $d_1 \in p_i.d$, then every process $p_j$ that finishes phase 2 is such that $d_1 \in p_j.d$. We then again have that there is a single process that can be elected because every process that finishes phase 2 has $d_1$ in its decision value and $d_1 \subseteq d_2$ by agreement.

It remains to show that if $p_i$ is elected in iteration $\zeta$, and $p_j$ is elected in iteration $\zeta'$, and $\zeta > \zeta'$, then there is no $\tau \in T$ such that both $p_i.\textit{elected}$ and $p_j.\textit{elected}$ are true at time $\tau$. By the algorithm, every process that starts the execution of phase 1 in an iteration, first sets its flag *elected* to false. If the iteration is the first of $E$, then

$p_j$ cannot be elected in a previous iteration, and the hypothesis is vacuously true. Now suppose an iteration $\zeta > 0$. By assumption, every process that executes a phase of an iteration $\zeta$ executes at least two rounds. By P-liveness, no process can start a new round $r + 1$, $r \geq 0$, without having every other alive process executing at least one step of $r$. If a process $p_i$ starts phase 2 of an iteration at time $\tau$, then every process that has not crashed by $\tau$ must have executed at least one step of round zero of FFS-ROC at phase 1 of $\zeta$. Otherwise, there is a non-crashed process $p_j$ such that $p_i$ executes the first step of a round $r + 1$ of FFS-ROC whereas $p_j$ has not executed any steps of $r$. This implies that no process can finish phase 2 of an iteration without having all non-crashed processes setting *elected* to false.

This concludes the proof of the theorem.

□

**Theorem A.5.3** *Algorithm FFS-WLE satisfies LE-Liveness.*

**Proof:**

We have to show that for every execution $E$ of FFS-WLE and for every $\tau \in T$, there is some iteration after $\tau$ such that $|\{p_i \in \Pi : p_i.\mathit{elected}\}| > 1$.

Proof by contradiction. Suppose an execution $E$ of FFS-WLE and a time $\tau \in T$ such that $p_i.\mathit{elected}$ is false forever after $\tau$ for every $p_i$. By validity and RO uniformity, in every iteration $\zeta$ of $E$, $\zeta \in \mathcal{I}$, there is a value $v \in \mathit{vals}_1$ such that $v$ is the list with the least number of non-$\perp$ values, and for every process $p_i$ that finishes phase 2 of iteration $\zeta$, $v \in p_i.d$. Every process that finishes phase 2 of iteration $\zeta$ selects the same value $i$ as the first index of $v$ mapping to a value in $v$ with a non-$\perp$ value. If process $p_i$ evaluates the last "if" statement of phase 2, then it sets $p_i.\mathit{elected}$ to true. If $p_i$ crashes, however, then it does not set $p_i.\mathit{elected}$ to true, and no process is elected in iteration $\zeta$. By the assumption that all failures are benign, crashing (or stopping which is equivalent to crashing in our model) is the only possibility for having no process elected in an iteration $\zeta$. By the assumption that $t$ ($|\Pi|$ subtracted the size of the smallest survivor set) is the largest number of processes that can crash in $E$, there can be at most $t$ iterations after $\tau$ such that $|\{p_i \in \Pi : p_i.\mathit{elected}\}| = 0$.

□

**Theorem A.5.4** *Algorithm FFS-WLE satisfies FF-Stability.*

**Proof:**

Suppose $E$ is a failure-free execution and $\zeta$ is an iteration of $E$. By agreement, every process decides upon the same value in both phases of iteration $\zeta$. We then have that every process $p_i$ uses the same value $x$ to determine whether it sets $p_i.\mathit{elected}$ to true in $E$. Moreover, we have that $x[i] = p_i$ for every $i$, by validity. Assuming that $\Pi = \{p_1, p_2, \ldots, p_n\}$, we have by the algorithm that $p_1$ sets $p_1.\mathit{elected}$ to true at phase 2 of $\zeta$.

□

## A.6 Adding E-Stability

FFS-WLE allows for executions in which two or more processes are elected infinitely often. Such behavior, however, is not desirable. As leadership moves from one process to another, the responsibility of accomplishing the tasks of a leader also moves. Recall that the original motivation is to embed such a Leader Election algorithm into a primary-backup protocol. This oscillation causes unnecessary overhead such as requests being forwarded to the correct Primary or even system instabilities if changes occur too frequently.

We now show how to modify FFS-WLE (and FFS-ROC) to also satisfy E-stability. We call WLE the modified version of FFS-WLE, and ROC the modified version of FFS-ROC to distinguish between the previous versions and the modified versions.

First, instead of initializing $p_i.s(0)$ to $\Pi$, as in FFS-ROC , $p_i.s(0)$ is initialized to a parameter $p_i.\mathit{Procs}$. We also roll forward the value of $p_i.s(t+1)$ in FFS-WLE instead of having $p_i.s(0)$ constant as in FFS-ROC. That is, in an iteration $\zeta > 0$, $p_i.s(0)$

in phase 1 is $p_i.s(t+1)$ in phase 2 of iteration $\zeta - 1$. If $\zeta = 0$, then $p_i.s(0)$ in phase 1 is $\Pi$. For the initial value of $p_i.s(0)$ in phase 2, we use $p_i.s(t+1)$ of phase 1 of the same iteration. For clarity, we repeat the pseudocode for these algorithms with the respective modifications in Figures A.4 and A.5. Note that the main modifications in ROC are: 1) ROC has two parameters instead of one; 2) in round 1, $p_i$ checks whether $p_i.s(1) \subseteq p_i.s(0)$. WLE is different from FFS-WLE by initializing $p_i.\mathit{Procs}$ to $\Pi$ and by rolling $p_i.s(t+1)$ forward.

It is straightforward to see that the proof of Section A.4 is valid for ROC if the following constraint holds for every execution $E$ of ROC: if $p_c \in \mathit{Correct}(E)$ and $p_i \in \Pi$ is a process in $p_c.s(1)$, then $p_i \in p_c.\mathit{Procs}$. That is, $p_c.\mathit{Procs}$ must contain all the processes that send messages to $p_c$ in round zero if $p_c$ is not faulty. Otherwise, $p_c$ can falsely detect that it is faulty, and stop, violating validity. Lemma A.4.5 states that correct processes do not stop, and hence the proof changes with the modification to the algorithm. The change is small, however. It consists in modifying the base case of the inductive argument to show that no correct process $p_c$ stops in round 1 if $p_c.s(0)$ contains all the processes that send messages to $p_c$ in round zero (0).

If $p_f$ is faulty, then there are no restrictions on the input $p_f.\mathit{Procs}$. Intuitively, a faulty process $p_f.\mathit{Procs}$ can receive any subset of processes sent to it. Consequently, it is not possible to impose a similar constraint as we did for correct processes. Different from correct processes, if a faulty process stops, it does not violate any of the RO consensus properties.

According to the modifications described previously, $p_i.\mathit{Procs}$ is the set of processes from which $p_i$ receives a message in the last round of the previous execution of ROC ($\Pi$ if it is the execution of ROC in phase 1 of iteration 0). By assumption and by the algorithm, once a process crashes (or stops) it never sends messages again in an execution of WLE. Thus, if $p_c$ is a correct process, then $p_c.\mathit{Procs}$ must contain all the processes that $p_c$ receives messages from in an execution of ROC, satisfying our constraint on $p_c.\mathit{Procs}$ for correct processes.

Because the proofs of Theorems A.5.2 and A.5.3 rely solely on the properties

```
**Algorithm** ROC on input $p_i.a$, $p_i.Procs$
round 0:
    $p_i.s(0) \leftarrow p_i.Procs$; $p_i.sr(0) \leftarrow p_i.s(0)$
    $p_i.A\,[i] \leftarrow p_i.a$
    for all $p_k \in \Pi$, $p_k \neq p_i$ : $p_i.A\,[i] \leftarrow \perp$
    $p_i.A_p(0) \leftarrow p_i.A$
    send $p_i.A$ to all

**round** 1:
    $p_i.sr(1) \leftarrow p_i.s(1)$
    **if** $\vee p_i.s(1) \not\subseteq p_i.s(0)$
        $\vee\ \nexists S \in \mathcal{S}_\Pi : S \subseteq p_i.sr(1)$
    **then decide** $[\perp, \ldots, \perp]$
    **else for** each message $m \in p_i.M(1)$, $p_k \in \Pi$:
            **if** $(p_i.A\,[k] = \perp)$ $p_i.A\,[k] \leftarrow m.A\,[k]$
    $p_i.A_p(1) \leftarrow p_i.A$
    **send** $p_i.A$ to all

**round** $r$: $2 \leq r \leq t$:
    $p_i.sr(r) \leftarrow p_i.s(r) \setminus \{p_j : \exists m \in p_i.M(r) :$
                $p_i.A_p(r-2) \not\subseteq m.A \wedge m.from = p_j\}$
    **if** $\vee p_i.s(r) \not\subseteq p_i.s(r-1)$
        $\vee\ \nexists S \in \mathcal{S}_\Pi : S \subseteq p_i.sr(r)$
    **then decide** $[\perp, ..., \perp]$
    **else for** each message $m \in p_i.M(r)$, $p_k \in \Pi$:
            **if** $(p_i.A\,[k] = \perp)$ $p_i.A\,[k] \leftarrow m.A\,[k]$
    $p_i.A_p(r) \leftarrow p_i.A$
    **send** $p_i.A$ to all

**round** $t+1$:
    $p_i.sr(t+1) \leftarrow p_i.s(t+1) \setminus \{p_j : \exists m \in p_i.M(t+1) :$
                $p_i.A_p(t-1) \not\subseteq m.A \wedge m.from = p_j\}$
    **if** $\vee\ p_i.s(t+1) \not\subseteq p_i.s(t)$
        $\vee\ \nexists S \in \mathcal{S}_\Pi : S \subseteq p_i.sr(t+1)$
    **then decide** $[\perp, ..., \perp]$
    **else for** each message $m \in p_i.M(t+1)$, $p_k \in \Pi$:
            **if** $(p_i.A\,[k] = \perp)$ $p_i.A[k] \leftarrow m.A[k]$
    $p_i.A_p(t+1) \leftarrow p_i.A$
    **decide** $p_i.A$
```

**Figure A.4:** ROC: Algorithm run by process $p_i$.

```
Algorithm WLE
$P \leftarrow \Pi$
repeat {
$p_i$.elected $\leftarrow$ FALSE
Phase 1:
    Run ROC with
        $p_i.a \leftarrow i$; $p_i.Procs \leftarrow P$.
    $P \leftarrow p_i.s(t+1)$
    if ($p_i.d = [\perp, \ldots, \perp]$) then stop
Phase 2:
    Run ROC with
        $p_i.a \leftarrow p_i.d$ from Phase 1; $p_i.Procs \leftarrow P$.
    $P \leftarrow p_i.s(t+1)$
    if ($p_i.d = [\perp, \ldots, \perp]$) then stop
    let $x$ be a value of $p_i.d\,[1 \ldots n]$
        such that $p_i.d\,[x] \neq [\perp, \ldots, \perp]$
        and it has the least number of non-$\perp$ values
    if ($p_i$ is the first index of $x$ such that $x[i] \neq \perp$)
        then $p_i$.elected $\leftarrow$ TRUE
}
```

**Figure A.5:** WLE: Algorithm run by process $p_i$.

of RO consensus, we also have that these proofs hold for WLE. It remains to show that WLE satisfies E-stability. First, we present a few definitions to be used in the proof of E-stability. By Theorem A.5.2, in every iteration $\zeta$ of an execution $E$ of WLE it is the case that either one process $p_i$ is such that $p_i$.*elected* evaluates to true at the end of $\zeta$ or no process $p_i$ is such that $p_i$.*elected* evaluates to true at the end of $\zeta$. We then use *Leader*$(\zeta, E)$ to denote the process $p_i$, $p_i$.*elected* evaluates to true at the end of iteration $\zeta$ of $E$, or $\perp$ if no such process exists. Finally, we need some terminology to refer to values that processes decide across iterations and in the two different phases of an iteration. We then use $\mathcal{D}^{\rho}_{\zeta}(i)$ to denote $p_i.d$ at the end of phase $\rho \in \{1, 2\}$ of iteration $\zeta$.

**Proposition A.6.1** WLE implements weak leader election.

With Theorems A.5.2, A.5.3, and A.5.4, we showed that WLE satisfies Stability, LE-liveness, and FF-stability. It remains to show E-stability. Before we show our main result of this section, we state and prove a few preliminary lemmas.

**Lemma A.6.2** *Let $E$ be an execution of WLE. For every iteration $\zeta$ of WLE, if $p_i$ finishes phase 1 of both $\zeta$ and $\zeta + 1$, then $\mathcal{D}^1_{\zeta+1}(i) \subseteq \mathcal{D}^1_{\zeta}(i)$.*

**Proof:**

Proof by contradiction. Suppose that there is an iteration $\zeta$ such that the assertion $\mathcal{D}^1_{\zeta+1}(i) \subseteq \mathcal{D}^1_{\zeta}(i)$ is false. This implies that there is some process $p_\ell$, $\ell \neq i$, for which $\mathcal{D}^1_{\zeta+1}(i)[\ell] \neq \perp$ and $\mathcal{D}^1_{\zeta}(i)[\ell] = \perp$. By Lemma A.4.10, in the execution of ROC in phase 1 of iteration $\zeta + 1$, there is a chain $\omega_\ell$ such that $\omega_\ell[r] = i$, $r > 0$. By assumption, for every process $p_j$ such that $\omega_\ell[\rho] = j$, $\rho \in \{z \in \mathcal{R} : 1 \leq z \leq r\}$, $p_{j'}$ must be in $p_j.\mathit{Procs}$, where $\omega_\ell[\rho - 1] = j'$, otherwise the process with identifier $\omega_\ell[\rho]$, $\rho \leq r$, stops in round $\rho$.

Now suppose that $\omega_\ell[1] = j$. By the algorithm, $p_j.sr(r)$ contains some survivor set $S_j$, for every round $r \in \{z \in \mathcal{R} : z \leq t + 1\}$ of the execution of ROC in iteration $\zeta$. Also, there is such a survivor set $S_i$ for $p_i$, and there is some survivor set $S_c$ such that $S_c \subseteq \mathit{Correct}(E)$. By (3,2)-Intersection, $S_i$ either intersects $S_j$ or intersects $S_c$. If $S_i$ intersects $S_j$, then by Lemma A.4.13, $\mathcal{D}^1_{\zeta}(i)[\ell] \neq \perp$, contradicting our initial assumption. If $S_i$ intersects $S_c$, then by Lemma A.4.14 $\mathcal{D}^2_{\zeta}(c) \subseteq \mathcal{D}^2_{\zeta}(i)$, for some non-faulty $p_c \in \mathit{Correct}(E)$. By agreement, $\mathcal{D}^2_{\zeta}(c)[\ell] \neq \perp$, and consequently $\mathcal{D}^1_{\zeta}(i)[\ell] \neq \perp$, again contradicting our initial assumption.

This completes the proof of the lemma.

□

**Lemma A.6.3** *Let $E$ be an execution of WLE and $\zeta$ be an iteration of $E$ . No process is elected in $\zeta$ only if some process crashes or stops in $\zeta$.*

**Proof:**

By RO uniformity, we have that $1 \leq \mathit{vals}_i \leq 2$, $i \in \{1, 2\}$. By validity and RO uniformity, there is some value $d$ in $\mathit{vals}_1$ such that $d \in p_i.d$ for every process $p_i$ that finishes phase 2 of iteration $\zeta$, and $d$ contains the least number of non-$\perp$ values. Because $d \neq \mathcal{N}$, there must be a process $p_e$ such that $e$ is the smallest index of $d$ such that $d[e] \neq \perp$. We then have that $p_e$ sets $p_e.\mathit{elected}$ to true unless $p_e$ crashes. Thus, if $\mathit{Leader}(\zeta, E) = \perp$, then $p_e$ must crash or stop in $\zeta$.

□

**Lemma A.6.4** *Let $E$ be an execution of WLE, $\zeta$ and $\zeta'$ be iterations of $E$, $\zeta + 1 < \zeta'$,* *such that* $\text{Leader}(\zeta, E) = \text{Leader}(\zeta', E) = p_e$. *If* $\text{Leader}(\zeta + 1, E) = p_{e'}$, $e \neq e'$, *then there is a process* $p_i$ *that crashes or stops in some iteration* $\zeta''$, $\zeta \leq \zeta'' \leq \zeta'$.

**Proof:**

If $p_e$ is elected in iteration $\zeta$ of $E$, then there is a value $d$ in $\mathit{vals}_1$ of $\zeta$ such that $e$ is the smallest index of $d$ with a non-$\perp$ value, and $d \in \mathcal{D}^2_\zeta(e)$. Now if $p_{e'}$ is elected in iteration $\zeta + 1$, then there is a value $d'$ in $\mathit{vals}_1$ of $\zeta + 1$ such that $e'$ is the smallest index of $d'$ with a non-$\perp$ value, and $d' \in \mathcal{D}^2_\zeta(e')$. By assumption, $p_e$ is elected again in iteration $\zeta'$. As before, there must be a value $d''$ in $\mathit{vals}_1$ of $\zeta'$ such that $e$ is the smallest index of $d''$ with a non-$\perp$ value.

There are two possibilities regarding the identifiers $e$ and $e'$: 1) $e' < e$; 2) $e < e'$. If $e' < e$, then there must be a second value $d_c$ in $\mathit{vals}_1$ of $\zeta$ such that $d \subseteq d_c$ (by assumption, $p_e$ has not crashed by iteration $\zeta' > \zeta + 1$; by validity, every non-faulty process $p_c$ is such that $p_e.a \in p_c.d$). Let $p_i$ be a process such that $\mathcal{D}^1_\zeta(i) = d$. Suppose by way of contradiction that $p_i$ finishes phase 2 of $\zeta + 1$. By Lemma A.6.2, $\mathcal{D}^1_{\zeta+1}(i) \subseteq \mathcal{D}^1_\zeta(i)$, and $\mathcal{D}^1_{\zeta+1}(i)[e'] =\perp$. By validity, there is some $p_c \in \mathit{Correct}(E)$ such that $\mathcal{D}^1_{\zeta+1}(c)[e'] \neq\perp$. Note that $p_{e'}$ does not crash or stop in iteration $\zeta+1$ or in a previous iteration. By RO uniformity, $\mathcal{D}^1_{\zeta+1}(i) \subseteq \mathcal{D}^1_{\zeta+1}(c) = d'$. By RO uniformity and validity, $\mathcal{D}^1_{\zeta+1}(i)$ must be the value used by every process that completes phase 2 of iteration $\zeta+1$ to determine whether it elects itself or not. Since $e'$ is not the smallest index of $\mathcal{D}^1_{\zeta+1}(i)$ that evaluates to a non-$\perp$ value, $p_{e'}$ is not elected in $\zeta + 1$. This contradicts our initial assumption. We hence have that $p_i$ must crash or stop by iteration $\zeta + 1$.

Now if $e < e'$, then $d'[e] =\perp$ by assumption ($e'$ is the smallest index in $d'$ with a non-$\perp$ value). We use a similar argument as in the first case. Suppose by way of contradiction that $p_{e'}$ finishes phase 2 of $\zeta'$. By Lemma A.6.2, $\mathcal{D}^1_{\zeta'}(e') \subseteq \mathcal{D}^1_{\zeta+1}(e')$, and $\mathcal{D}^1_{\zeta'}(e')[e] =\perp$. By validity, there is some $p_c \in \mathit{Correct}(E)$ such that $\mathcal{D}^1_{\zeta'}(c)[e] \neq\perp$. Note that $p_e$ does not crash or stop in iteration $\zeta'$ or in a previous iteration. By RO uniformity, $\mathcal{D}^1_{\zeta'}(e') \subseteq \mathcal{D}^1_{\zeta'}(c) = d''$. By RO uniformity and validity, $\mathcal{D}^1_{\zeta'}(e')$ must be the value used by every process that completes phase 2 of iteration $\zeta'$ to determine whether it elects itself or not. Since $e$ is not the smallest index of $\mathcal{D}^1_{\zeta'}(i)$ that evaluates to a non-$\perp$ value, $p_e$ is not elected in $\zeta'$. This contradicts our initial assumption. We conclude that $p_e$ is not elected in $\zeta'$ unless $p_{e'}$ crashes or stops by iteration $\zeta'$.

Finally, we have that either $p_{e'}$ crashes or stops by iteration $\zeta'$ or some faulty process $p_i$ crashes or stops by iteration $\zeta + 1$. This concludes the proof of the lemma.

□

**Theorem A.6.5** *WLE satisfies E-Stability.*

**Proof:**

Let $E$ be an execution of WLE. By LE-liveness, infinitely often some process is elected in $E$. By Lemma A.6.3, an iteration $\zeta$ has no leader elected only if some process crashes in $\zeta$, and by assumption there is a finite number of processes that crash or stop. Thus, there is a bounded number of iterations that have no leader elected.

Let $t$ be a time such that every iteration that starts after $t$ has a leader elected. Such a $t$ exists by the previous argument. We then have that every iteration that starts after $t$ has a leader elected, and it remains to show that there is some $t' \geq t$ and some process $p_e$ such that for every iteration $\zeta$ that starts after $t'$, $p_e$ is elected in both $\zeta$ and $\zeta + 1$. Suppose by way of contradiction that there is no such $t'$ in $E$. Let $p_e$ be a process that is elected infinitely often after $t'$. Such a process must exist because the set of processes is finite. By assumption, there is an infinite sequence of iterations $\zeta_1 < \zeta_2 < \zeta_3 \ldots$, which are not necessarily consecutive, such that $p_e$ is elected in $\zeta_i$ but not in $\zeta_i + 1$. By Lemma A.6.4, for every $i \in \mathbb{Z}^+$, there is an iteration $\zeta$, $\zeta_i \leq \zeta \leq \zeta_{i+1}$, such that some process crashes or stops in $\zeta$. By assumption, the number of processes crashing or stopping is bounded. Consequently, there cannot be such an infinite sequence. We conclude that there must be some $t' \geq t$ and some process $p_e$ such that for every iteration $\zeta$ that starts after $t'$, $p_e$ is elected in both $\zeta$ and $\zeta + 1$.

□

## A.7 Discussion

Developing a primary-backup protocol that uses WLE is future work. We can, however, make a few observations regarding the use of an algorithm as WLE to elect primaries in a primary-backup protocol. As mentioned previously, WLE enables faulty processes to be elected. In a primary-backup system, this feature impacts liveness, although not correctness. Often, there is a time bound on the replies to client requests, and it is impossible to meet such bounds if the primary can be faulty. An immediate consequence of electing faulty processes is that service time is not bounded during the period of time a faulty process remains as the primary. As discussed before, processes that commit failures (but do not stop or crash) are detected. In practice, we rely on an off-line mechanism to detect these anomalies and take the appropriate measures that can be, for example, to remove faulty processes from the system.

It is possible, however, that faulty processes go through an iteration of WLE undetected as such, and fail to reply to client requests due to receive-omission failures. To solve this problem, we can require clients to broadcast requests to all the replicas and the primary to broadcast replies to all the backup replicas as well. Correct processes are also capable of detecting failures in such cases, although they may not be able to “warn” the faulty primary that it is actually faulty. Recall that failure detection for omission failures requires twofold replication.

Finally, the iterations of the repeat loop of WLE are consecutive without any delay in between for expositional purposes. In practice, iterations should be delayed until failures are detected, they are manually triggered, or, if none of these are desirable or possible, some time threshold is reached.

## A.8 Conclusions

We presented in this chapter a weaker version of the Leader Election problem and an algorithm that solves this problem. This version of the problem, unlike the traditional definition of Leader Election, enables faulty processes to be elected. The main advantage of enabling it is requiring a lower degree of replication.

There are other interesting features of the WLE algorithm. First, it uses cores and survivor sets instead of a threshold. This approach enables a more flexible characterization of systems with a heterogeneous set of processes. Second, it uses an unusual type of Intersection property, *i.e.*, (3,2)-Intersection. This property generalizes a degree of replication of the form $n > \lfloor 3t/2 \rfloor$, where $t$ is the threshold on the number of failures in any execution. Finally, correct processes are able to detect faulty processes. By Lemma A.4.16, non-crashed faulty processes decide upon lists with fewer values, and one can build an alarm system by collecting decision values by the end of every iteration.

Although we have not thoroughly investigated using WLE to implement primary-backup protocols, we believe our algorithm provides practical benefits compared to previous solutions.

# Appendix B

# **SyncCrash** specification in TLA+

```
──────────────────────── MODULE SyncCrashConsensusED ────────────────────────
EXTENDS FiniteSets, Naturals, TLC

CONSTANTS V,                          Values
          I,                          Initial values
          P,                          Set of processes
          Cores,                      Set of cores
          NULL                        Default value

VARIABLES   CurrentRound,
            MsgsToSend,
            Faulty,
            Silent,
            PV,
            SentMsgs,
            SentDecs,
            CompletedRecv,
            Completed,
            Decision

ASSUME  ∧ ∀ C ∈ Cores : C ∈ SUBSET P
        ∧ I ∈ [P → V]
        ∧ NULL ∉ V
─────────────────────────────────────────────────────────────────────────────

MinCore ≜ CHOOSE C ∈ Cores : ∧ ∀ D ∈ Cores : Cardinality(D) ≥ Cardinality(C)

Rounds ≜ 0 . . (Cardinality(MinCore) + 1)

Correct ≜ P \ Faulty
Msgs ≜ [from : P, to : P, vals  : [P → V ∪ {NULL}], r : Rounds]
DecMsgs ≜ [from : P, to : P, v : V]

RoundMsgs(p, r) ≜ {m ∈ SentMsgs : m.to = p ∧ m.r = r}
RoundValues(p, r) ≜ [q ∈ P ↦
```

```
            IF $PV[p][q] \neq NULL$
             THEN $PV[p][q]$
             ELSE
               IF $\exists\, m \in RoundMsgs(p, r) : m.vals[q] \neq NULL$
                   THEN CHOOSE $v \in \{u \in V :$
                                    $\exists\, m \in RoundMsgs(p, r) :$
                                        $m.vals[q] = u\} :$ TRUE
                   ELSE  $NULL]$
$DecisionValue(p, Values) \triangleq$  CHOOSE $v \in V : \exists\, q \in P : Values[p][q] = v$

$E(p, r) \triangleq P \setminus \{q \in P : \exists\, m \in SentMsgs : m.r = r \land m.from = q \land m.to = p\}$
```

---

```
$SCCEDTypeOK \triangleq$   $\land\ CurrentRound \in Rounds$
                   $\land\ MsgsToSend \in [P \rightarrow$ SUBSET $Msgs]$
                   $\land\ Faulty \in$ SUBSET $P$
                   $\land\ Silent \in [P \rightarrow$ BOOLEAN $]$
                   $\land\ PV \in [P \rightarrow [P \rightarrow V \cup \{NULL\}]]$
                   $\land\ SentMsgs \in$ SUBSET $Msgs$
                   $\land\ SentDecs \in$ SUBSET $DecMsgs$
                   $\land\ CompletedRecv \in [P \rightarrow Rounds]$
                   $\land\ Completed \in [P \rightarrow Rounds]$
                   $\land\ Decision \in [P \rightarrow V \cup \{NULL\}]$

$SCCEDInit \triangleq$     $\land\ CurrentRound = 0$
                   $\land\ Faulty = \{\}$
                   $\land\ Silent = [p \in P \mapsto$ FALSE$]$
                   $\land\ PV = [p \in P \mapsto [q \in P \mapsto$ IF $q = p$ THEN $I[p]$ ELSE $NULL]]$
                   $\land\ SentMsgs = \{\}$
                   $\land\ SentDecs = \{\}$
                   $\land\ MsgsToSend = [p \in P \mapsto$
                                 $\{[from \mapsto p, to \mapsto q,$
                                     $vals \mapsto [tp \in P \mapsto$
                                                 IF $tp = p$
                                                  THEN $I[p]$
                                                  ELSE $NULL],$
                                     $r \mapsto 0] :$
                                     $q \in P\}]$
                   $\land\ CompletedRecv = [p \in P \mapsto 0]$
                   $\land\ Completed = [p \in P \mapsto 0]$
                   $\land\ Decision = [p \in P \mapsto NULL]$

$Fail(p) \triangleq$       $\land\ \forall\, C \in Cores : \exists\, q \in (Correct \setminus \{p\}) : q \in C$
                   $\land\ Faulty' = Faulty \cup \{p\}$
                   $\land$ UNCHANGED $\langle CurrentRound, MsgsToSend, Silent, PV,$
                          $SentMsgs, SentDecs, CompletedRecv, Completed,$
                          $Decision\rangle$

$NextRound \triangleq$     $\land\ \forall\, p \in P :\ Completed[p] = CurrentRound$
                   $\land\ CurrentRound' = (CurrentRound + 1)$
```

```
                  ∧ UNCHANGED ⟨MsgsToSend, Faulty, Silent, PV, SentMsgs,
                               SentDecs, CompletedRecv, Completed,
                               Decision⟩

Send(p) ≜         ∧ CurrentRound < Cardinality(MinCore)
                  ∧ p ∈ MinCore
                  ∧ CompletedRecv[p] = CurrentRound
                  ∧ MsgsToSend' = [MsgsToSend EXCEPT ![p] = {}]
                  ∧ Completed' = [Completed EXCEPT ![p] = CurrentRound]
                  ∧ IF p ∈ Correct
                      THEN      ∧ SentMsgs' = SentMsgs ∪ MsgsToSend[p]
                                ∧ Silent' = Silent
                      ELSE IF Silent[p]
                            THEN (  ∧ SentMsgs' = SentMsgs
                                    ∧ Silent' = Silent)
                            ELSE (  ∧ ∃ TMsgs ∈ SUBSET MsgsToSend[p] :
                                        ∧ ((∃ q ∈ (P \ MinCore) :
                                              ∃ m ∈ TMsgs : m.to = q) ⇒
                                              (∀ q ∈ MinCore : ∃ m ∈ TMsgs :
                                                  m.to = q))
                                        ∧ SentMsgs' = SentMsgs ∪ TMsgs
                                        ∧ Silent' = [Silent EXCEPT ![p] = TRUE])
                  ∧ UNCHANGED ⟨CurrentRound, Faulty, PV, SentDecs, CompletedRecv,
                               Decision⟩

Recv(p) ≜      LET ToSend ≜ {[from ↦ p,
                              to ↦ q,
                              vals ↦ RoundValues(p, CurrentRound − 1),
                              r ↦ CurrentRound] : q ∈ P}
               IN
                ∧ CompletedRecv[p] = CurrentRound − 1
                ∧ CompletedRecv' = [CompletedRecv EXCEPT ![p] = CurrentRound]
                ∧ IF ((p ∈ (P \ MinCore)) ∧ (CurrentRound < Cardinality(MinCore)))
                      THEN Completed' = [Completed EXCEPT ![p] = CurrentRound]
                      ELSE  Completed' = Completed
                ∧ PV' = [PV EXCEPT ![p] =
                             RoundValues(p, CurrentRound − 1)]
                ∧ MsgsToSend' = [MsgsToSend EXCEPT ![p] = ToSend]
                ∧ UNCHANGED ⟨CurrentRound, Faulty, Silent,
                             SentMsgs, SentDecs, Decision⟩

Decide(p) ≜     ∧ (CurrentRound = Cardinality(MinCore))
                ∧ p ∈ Correct
                ∧ CompletedRecv[p] = CurrentRound
                ∧ Completed' = [Completed EXCEPT ![p] = CurrentRound]
                ∧ PV' = [PV EXCEPT ![p] =
                             RoundValues(p, CurrentRound − 1)]
                ∧ Decision' = [Decision EXCEPT ![p] =
                            DecisionValue(p, PV')]
```

```
                    $\land$ UNCHANGED $\langle CurrentRound,\ MsgsToSend,\ Faulty,$
                                  $CompletedRecv,\ Silent,\ SentMsgs,$
                                  $SentDecs\rangle$

$EarlyDecide(p) \triangleq$   LET $ToSendDecs(values) \triangleq \{[from \mapsto p,$
                                                   $to \mapsto q,$
                                                   $v \mapsto DecisionValue(p,\ values)] :$
                                                  $q \in P\}$
                         IN
                          $\land\ Decision[p] \neq NULL$
                          $\land\ CurrentRound > 0$
                          $\land\ CurrentRound < Cardinality(MinCore)$
                          $\land\ E(p,\ CurrentRound) = E(p,\ CurrentRound - 1)$
                          $\land\ CompletedRecv' = [CompletedRecv$ EXCEPT $![p] = CurrentRound]$
                          $\land\ Completed' = [Completed$ EXCEPT $![p] = CurrentRound]$
                          $\land\ PV' = [PV$ EXCEPT $![p] =$
                                        $RoundValues(p,\ CurrentRound - 1)]$
                          $\land\ Decision' = [Decision$ EXCEPT $![p] =$
                                       $DecisionValue(p,\ PV')]$
                          $\land$ IF $p \in Faulty$
                                THEN $\exists\ TMsgs \in ToSendDecs(PV')$ :
                                         $SentDecs' = SentDecs \cup TMsgs$
                                ELSE $SentDecs' = SentDecs \cup ToSendDecs(PV')$
                          $\land\ Silent' = [Silent$ EXCEPT $![p] =$ TRUE$]$
                          $\land$ UNCHANGED $\langle CurrentRound,\ MsgsToSend,\ Faulty,\ SentMsgs\rangle$

$RecvDec(p) \triangleq$   LET $ToSendDecs(v) \triangleq \{[from \mapsto p,\ to \mapsto q,\ v \mapsto v] : q \in P\}$
                    IN
                    $\land\ Decision[p] \neq NULL$
                    $\land\ Completed[p] = CurrentRound - 1$
                    $\land\ CompletedRecv' = [CompletedRecv$ EXCEPT $![p] = CurrentRound]$
                    $\land\ Completed' = [Completed$ EXCEPT $![p] = CurrentRound]$
                    $\land\ \exists\ m \in SentDecs :$   $\land\ m.to = p$
                                          $\land\ Decision' = [Decision$ EXCEPT $![p] = m.v]$
                                          $\land$ IF $p \in Faulty$
                                                THEN $\exists\ TMsgs \in ToSendDecs(m.v)$ :
                                                         $SentDecs' = SentDecs \cup TMsgs$
                                                ELSE $SentDecs' =$
                                                          $SentDecs \cup ToSendDecs(m.v)$
                    $\land\ Silent' = [Silent$ EXCEPT $![p] =$ TRUE$]$
                    $\land$ UNCHANGED $\langle CurrentRound,\ MsgsToSend,\ Faulty,\ SentMsgs\rangle$

$SCCEDStep \triangleq$   $\lor\ NextRound$
                   $\lor\ \exists\ p \in P$ :
                         $\lor\ Send(p)$
                         $\lor$ IF $(\exists\ m \in SentDecs : m.to = p)$
                               THEN $RecvDec(p)$
                               ELSE IF $(E(p,\ CurrentRound) = E(p,\ CurrentRound - 1))$
                                       THEN $EarlyDecide(p)$
```

```
                                   ELSE  $Recv(p)$
                          $\lor Decide(p)$
                          $\lor Fail(p)$

$vars \triangleq \langle CurrentRound, MsgsToSend, Faulty, Silent, PV, SentMsgs,$
        $SentDecs, CompletedRecv, Completed, Decision \rangle$
$SCCEDSpec \triangleq SCCEDInit \land \Box[SCCEDStep]_{vars}$
```

---

```
$Termination \triangleq \lor CurrentRound < Cardinality(MinCore) + 1$
                        $\lor \forall p \in Correct : Decision[p] \neq NULL$
$Agreement \triangleq \exists v \in V : \forall p \in P : Decision[p] = v \lor Decision[p] = NULL$
$Validity \triangleq \forall p \in P : \lor Decision[p] = NULL$
                             $\lor \exists q \in P : Decision[p] = I[q]$
$Safety \triangleq Agreement \land Validity$
$Liveness \triangleq Termination$

THEOREM $SCCEDSpec \Rightarrow \Box SCCEDTypeOK$
```

---

# Appendix C

# SyncByz specification in TLA+

```
MODULE SyncByzConsensus
EXTENDS FiniteSets, Naturals, Sequences, TLC

CONSTANTS V,                     Values
          I,                     Initial values
          P,                     Set of processes
          SurvivorSets,          Set of cores
          NULL,                  Default value
          DEC

VARIABLES    CurrentRound,
             Faulty,
             NotSent,
             MyValues,
             SentMsgs,
             Completed,
             Decision

ASSUME  ∧ ∀ S ∈ SurvivorSets : S ∈ SUBSET P
        ∧ ∀ S1, S2, S3 ∈ SurvivorSets : S1 ∩ S2 ∩ S3 ≠ {}
        ∧ I ∈ [P → V]
        ∧ NULL ∉ V
        ∧ DEC ∉ P
```

```
 Types
VD ≜ V ∪ {NULL}

PSeqsRound(r) ≜ {S ∈ [1 .. r → P] :  ∀ i, j ∈ 1 .. r : S[i] = S[j] ⇒ i = j}

PSeqsRoundExc(p, r) ≜ {S ∈ PSeqsRound(r) :  ∀ i ∈ 1 .. r : S[i] ≠ p}

PSeqs ≜ UNION {PSeqsRound(r) : r ∈ 1 .. Cardinality(P)}

PairType ≜ [SeqOfPs : PSeqs ∪ {1 :> DEC}, Value : VD]

MinSS ≜ CHOOSE S ∈ SurvivorSets : ∀ OS ∈ SurvivorSets :
```

```
                                        $Cardinality(OS) \geq Cardinality(S)$

$Rounds \triangleq 0 \,..\, (2 + Cardinality(P) - Cardinality(MinSS))$

$LastRound \triangleq 1 + Cardinality(P) - Cardinality(MinSS)$

────────────────────────────────────────────────────────────────

$Correct \triangleq P \setminus Faulty$

$SSPred(s) \triangleq$ LET $PinS(ns) \triangleq \{p \in P : \exists\, i \in 1 \,..\, Len(ns) : ns[i] = p\}$
                      IN
                         $\exists\, S \in SurvivorSets : S \subseteq (P \setminus PinS(s))$

$Msgs \triangleq [from : P,\ to : P,\ vals : \text{SUBSET } PairType,\ r : Rounds]$

$RoundMsgs(p, r) \triangleq \{m \in SentMsgs : m.to = p \land m.r = r \land m.vals \neq \{\}\}$

$NewRoundValues(p, r) \triangleq \{pair \in PairType :$
                                  $\exists\, m \in RoundMsgs(p, r) :$
                                        $pair \in m.vals\}$

$ToSendValues(p, q, r) \triangleq$ LET $MyRoundValues \triangleq \{pair \in PairType :$
                                      $(\exists\, m \in RoundMsgs(p, r - 1) :$
                                            $\exists\, pt \in m.vals :$
                                             $\land\ pair.SeqOfPs = ((1 :> p) \circ pt.SeqOfPs)$
                                             $\land\ pair.Value = pt.Value)\}$
                                IN
                                IF $r = 0$
                                 THEN $MyValues[p]$
                                 ELSE
                                $\{\ pt \in MyRoundValues :$
                                       $\land\ (\forall\, i \in 1 \,..\, Len(pt.SeqOfPs) : pt.SeqOfPs[i] \neq q)$
                                       $\land\ (SSPred(Tail(pt.SeqOfPs)))$
                                $\}$

$IntermediateSet(p, r) \triangleq \{mseq \in PSeqsRound(r + 1) :$
                                  $\land\ SSPred(Tail(mseq))$
                                  $\land\ \exists\, tmseq \in PSeqsRoundExc(p, r) :$
                                        $mseq = (1 :> p) \circ tmseq\}$

$FaultyRoundValues(p, q, r) \triangleq \{pair \in PairType : \land\ pair.SeqOfPs \in IntermediateSet(p, r)$
                                                        $\land\ pair.Value \in V\}$

$FaultyValues(p, q, r) \triangleq \{SVals \in \text{SUBSET } FaultyRoundValues(p, q, r) :$
                                $\land\ \forall\, pt1 \in SVals : \forall\, pt2 \in (SVals \setminus pt1) :$
                                                        $pt1.SeqOfPs \neq pt2.SeqOfPs$
                             $\}$

$DecisionValue[p \in P,\ pSeqs \in \text{SUBSET } PairType,\ count \in Nat] \triangleq$
                 LET      $VSet(s) \triangleq \{v \in VD :$
                                        $\land\ \exists\, ov \in V :$
                                           $\land\ \exists\, S1, S2 \in SurvivorSets :$
                                        $\forall\, q \in S1 \cap S2$            :
                                                $[SeqOfPs \mapsto (1 :> q) \circ s,$
                                                 $Value \mapsto ov] \in MyValues[p]$
                                    $\wedge\; v = ov$
                                $\}$
                $VSetFinal \triangleq \{v \in V :$
                                $\exists\, ov \in V :$
                                    $(\wedge \exists\, S1,\, S2 \in SurvivorSets :$
                                            $\forall\, q \in S1 \cap S2 :$
                                             $[SeqOfPs \mapsto (1 :> q),$
                                              $Value \mapsto ov] \in MyValues[p]$
                                      $\wedge\; v = ov)$
                                  $\}$
                $VFinalArb \triangleq$ CHOOSE $v \in V : \exists\, s \in pSeqs : s.Value = v$
                $SChoice \triangleq$ CHOOSE $s \in pSeqs :$
                                $\forall\, os \in pSeqs :$
                                    $Len(s.SeqOfPs) \geq Len(os.SeqOfPs)$
            IN
            IF $\exists\, v \in VD : [SeqOfPs \mapsto (1 :> DEC),\; Value \mapsto v] \in pSeqs$
             THEN CHOOSE $v \in VD :$
                        $[SeqOfPs \mapsto (1 :> DEC),\; Value \mapsto v] \in pSeqs$
             ELSE  $DecisionValue[p,\, (pSeqs$
                       $\cup \{pt \in PairType :$
                          $\vee\; \wedge\, pt.SeqOfPs = Tail(SChoice.SeqOfPs)$
                             $\wedge$ IF $VSet(Tail(SChoice.SeqOfPs)) = \{\}$
                                 THEN $pt.Value = NULL$
                                 ELSE $\exists\, v \in VSet(Tail(SChoice.SeqOfPs)) :$
                                          $pt.Value = v$
                          $\vee\; \wedge\, Len(SChoice.SeqOfPs) = 1$
                             $\wedge\, pt.SeqOfPs = (1 :> DEC)$
                             $\wedge$ IF $VSetFinal = \{\}$
                                 THEN $pt.Value = VFinalArb$
                                 ELSE $\exists\, v \in VSet(Tail(SChoice.SeqOfPs)) :$
                                          $pt.Value = v\})$
                      $\setminus \{r\_pt \in pSeqs :$
                              $\vee\, (\quad \wedge\, Len(r\_pt.SeqOfPs) = 1$
                                       $\wedge\, Len(SChoice.SeqOfPs) = 1)$
                              $\vee\, (\quad \wedge\, Len(SChoice.SeqOfPs) > 1$
                                       $\wedge\, Tail(r\_pt.SeqOfPs) = Tail(SChoice.SeqOfPs))$
                      $\},\; count + 1]$
```

---

```
$SBCTypeOK \triangleq$   $\wedge\; CurrentRound \in Rounds$
                  $\wedge\; Faulty \in$ SUBSET $P$
                  $\wedge\; NotSent \in [P \rightarrow$ BOOLEAN $]$
                  $\wedge\; MyValues \in [P \rightarrow$ SUBSET $PairType]$
                  $\wedge\; SentMsgs \in$ SUBSET $Msgs$
                  $\wedge\; Completed \in [P \rightarrow Rounds]$
```

```
                    $\land$ $Decision \in [P \rightarrow VD]$

$SBCInit$ $\triangleq$     $\land$ $CurrentRound = 0$
                    $\land$ $Faulty = \{\}$
                    $\land$ $NotSent = [p \in P \mapsto \text{TRUE}]$
                    $\land$ $MyValues = [p \in P \mapsto \{[SeqOfPs \mapsto (1 :> p),\ Value \mapsto I[p]]\}]$
                    $\land$ $SentMsgs = \{\}$
                    $\land$ $Completed = [p \in P \mapsto 0]$
                    $\land$ $Decision = [p \in P \mapsto NULL]$

$Fail(p)$ $\triangleq$    $\land$ $\exists S \in SurvivorSets : (Faulty \cup \{p\}) \cap S = \{\}$
                  $\land$ $Faulty' = Faulty \cup \{p\}$
                  $\land$ UNCHANGED $\langle CurrentRound,\ NotSent,\ MyValues,$
                                  $SentMsgs,\ Completed,\ Decision\rangle$

$NextRound$ $\triangleq$  $\land$ $\forall p \in P : (\land \neg NotSent[p]$
                            $\land$ $Completed[p] = CurrentRound)$
                  $\land$ $CurrentRound' = CurrentRound + 1$
                  $\land$ UNCHANGED $\langle NotSent,\ Faulty,\ MyValues,$
                                  $SentMsgs,\ Completed,\ Decision\rangle$

$Send(p)$ $\triangleq$    LET $TMsgVals(tp,\ tq,\ r)$ $\triangleq$
                      $\{tvals \in$ SUBSET $FaultyRoundValues(tp,\ tq,\ r)$ :
                            $\land$ $\forall v1 \in tvals$ :
                                  $\forall v2 \in (tvals \setminus \{v1\})$ :
                                        $v1.SeqOfPs \neq v2.SeqOfPs$
                            $\land$ $tvals \neq \{\}\}$
                    $TMsgSet(q)$ $\triangleq$
                      $\{[from \mapsto p,\ to \mapsto q,\ r \mapsto CurrentRound,\ vals \mapsto tvals]$ :
                            $tvals \in TMsgVals(p,\ q,\ Completed[p])\}$
                    $TMsgsAll$ $\triangleq$ UNION $\{TMsgSet(q) : q \in P\}$
                    $PChoice(SubP)$ $\triangleq$ CHOOSE $q \in SubP : q \in P$
                    $TMsgsAux(q,\ Opts)$ $\triangleq$ $\{\langle m,\ M\rangle \in (TMsgSet(q) \times Opts) : \text{TRUE}\}$
                    $TMsgsOptions[SubP \in$ SUBSET $P]$ $\triangleq$
                      IF $(SubP = \{\})$
                       THEN $\{\{\}\}$
                       ELSE UNION $\{\{\{m\} \cup PM : PM \in$
                                  $TMsgsOptions[SubP \setminus \{PChoice(SubP)\}]\}$ :
                                        $m \in TMsgSet(PChoice(SubP))\}$
                    $TMsgsMerge$ $\triangleq$ UNION $\{TMsgsOptions[SubP] : SubP \in$ SUBSET $(P \setminus \{p\})\}$
                    $Pmp$ $\triangleq$ $P \setminus \{p\}$
                IN
                 $\land$ $CurrentRound < LastRound$
                 $\land$ $Completed[p] = CurrentRound$
                 $\land$ $NotSent[p] = \text{TRUE}$
                 $\land$ $NotSent' = [NotSent$ EXCEPT $![p] = \text{FALSE}]$
                 $\land$ IF $p \in Correct$
                        THEN $SentMsgs' = SentMsgs \cup$
                                                  $\{[from \mapsto p,$
                                        $to \mapsto q,$
                                        $r \mapsto CurrentRound,$
                                        $vals \mapsto ToSendValues(p,\ q,\ Completed[p])]:$
                                        $q \in P\}$
                      $\text{ELSE } \exists\, TMsgs \in TMsgsMerge : SentMsgs' = SentMsgs \cup TMsgs$
                  $\land \text{ UNCHANGED } \langle CurrentRound,\ Faulty,\ MyValues,\ Completed,\ Decision\rangle$

$Recv(p) \triangleq$    $\text{LET } TMsgSet(q) \triangleq$
                          $\{[from \mapsto p,\ to \mapsto q,\ r \mapsto CurrentRound,\ vals \mapsto tvals]:$
                              $tvals \in \text{SUBSET } FaultyRoundValues(p,\ q,\ Completed[p])\}$
                          $Pmp \triangleq P \setminus \{p\}$
                  $\text{IN}$
                  $\land\ Completed[p] = CurrentRound - 1$
                  $\land\ Completed' = [Completed \text{ EXCEPT } ![p] = CurrentRound]$
                  $\land\ MyValues' = [MyValues \text{ EXCEPT } ![p] =$
                                  $MyValues[p] \cup$
                                  $NewRoundValues(p,\ Completed[p])]$
                  $\land\ NotSent' = [NotSent \text{ EXCEPT } ![p] = \text{TRUE}]$
                  $\land \text{ UNCHANGED } \langle CurrentRound,\ Faulty,\ SentMsgs,\ Decision\rangle$

$Decide(p) \triangleq$  $\land\ CurrentRound = LastRound$
                  $\land\ Completed[p] \ \ = LastRound$
                  $\land\ Completed' = [Completed \text{ EXCEPT } ![p] = CurrentRound]$
                  $\land\ NotSent' = [NotSent \text{ EXCEPT } ![p] = \text{FALSE}]$
                  $\land\ MyValues' = [MyValues \text{ EXCEPT } ![p] =$
                                  $MyValues[p] \cup$
                                  $NewRoundValues(p,\ Completed[p])]$
                  $\land \text{ IF } p \in Faulty$
                      $\text{THEN } \exists\, v \in VD : Decision' = [Decision \text{ EXCEPT } ![p] = v]$
                      $\text{ELSE } Decision' = [Decision \text{ EXCEPT } ![p] =$
                                  $DecisionValue[p,\ MyValues[p] \cup$
                                      $NewRoundValues(p,\ Completed[p]),\ 0]]$
                  $\land \text{ UNCHANGED } \langle CurrentRound,\ Faulty,\ SentMsgs\rangle$

$SBCStep \triangleq \exists\, p \in P :$  $\lor\ Send(p)$
                      $\lor\ Recv(p)$
                      $\lor\ Decide(p)$
                      $\lor\ Fail(p)$
                      $\lor\ NextRound$

$vars \triangleq \langle CurrentRound,\ Faulty,\ NotSent,\ MyValues,\ SentMsgs,\ Completed,\ Decision\rangle$
$SBCSpec \triangleq SBCInit \land \Box[SBCStep]_{vars}$
```

```
$Termination \triangleq$  $\lor\ CurrentRound < LastRound + 1$
                  $\lor\ \forall\, p \in Correct : Decision[p] \neq NULL$

$Agreement \triangleq \exists\, v \in V : \forall\, p \in Correct : Decision[p] = v \lor Decision[p] = NULL$
$Validity \triangleq \ \forall\, p \ \ \in Correct : \exists\, q \in P \ : Decision[p] = I[q] \lor Decision[p] = NULL$
$Safety \triangleq Agreement \land Validity$
$Liveness \triangleq Termination$
```

THEOREM $SBCSpec \Rightarrow \Box SBCTypeOK$

# Appendix D

# AsyncCrash specification in TLA+

MODULE *AsyncCrashConsensus*

EXTENDS *FiniteSets*, *Naturals*, *TLC*

| | | |
|---|---|---|
| CONSTANTS | $P$, | Set of processes |
| | $I$, | Initial values |
| | $SurvivorSets$, | Survivor sets |
| | $V$, | Decision values |
| | $NULL$, | Default value |
| | $MaxRound$, | Largest round number |
| | $PMap$, | Mapping from integers to processes |
| | $EstMsg$, $CoordMsg$, $EchoMsg$, $MoveOnMsg$, $DecMsg$, | |
| | $InitRound$, $WaitEstimates$, $WaitCoordEstimate$, $WaitEchoes$, | |
| | $Decided$, $Silent$ | |

ASSUME $\land \forall S \in SurvivorSets : S \subseteq P$
$\land \forall S1, S2 \in SurvivorSets : S1 \cap S2 \neq \{\}$
$\land NULL \notin V$
$\land I \in [P \rightarrow V]$
$\land PMap \in [0 \,..\, (Cardinality(P) - 1) \rightarrow P]$

| | | |
|---|---|---|
| VARIABLES | $Estimate$, | *CurrentEstimate* |
| | $EstUpdate$, | Last round that estimate was updated |
| | $Decision$, | Decision value |
| | $MyStage$, | Current state of a process in a round |
| | $Round$, | Time counter |
| | $Faulty$, | Faulty Processes |
| | $SentMsgs$ | Sent Messages |

---

$RoundRange \triangleq 0 \,..\, MaxRound$
$MsgType \triangleq \{EstMsg, CoordMsg, EchoMsg, MoveOnMsg, DecMsg\}$
$States \triangleq \{InitRound, WaitEstimates, WaitCoordEstimate, WaitEchoes, Decided, Silent\}$
$EstimateMsgs \triangleq [type : MsgType, from : P, to : P, r : RoundRange, v : V, updr : RoundRange]$
$CoordEstMsgs \triangleq [type : MsgType, from : P, to : P, r : RoundRange, v : V]$
$EchoMsgs \triangleq [type : MsgType, from : P, to : P, r : RoundRange, v : V]$

$$
\begin{array}{l}
\mathit{MoveOnMsgs} \triangleq [\mathit{type} : \mathit{MsgType},\ \mathit{from} : P,\ \mathit{to} : P,\ r : \mathit{RoundRange}] \\
\mathit{DecideMsgs} \triangleq [\mathit{type} : \mathit{MsgType},\ \mathit{from} : P,\ \mathit{to} : P,\ d : V] \\
\mathit{Coordinator}(r) \triangleq \mathit{PMap}[r \% \mathit{Cardinality}(P)]
\end{array}
$$

$$
\begin{array}{l}
\mathit{CurEstimates}(r) \triangleq \{m \in \mathit{SentMsgs} : m.\mathit{type} = \mathit{EstMsg} \land m.\mathit{to} = \mathit{Coordinator}(r)\} \\
\mathit{CurEchoes}(p,\ r) \triangleq \{m \in \mathit{SentMsgs} : m.\mathit{type} = \mathit{EchoMsg} \land m.r = r \land m.\mathit{to} = p\} \\
\mathit{CurMoveOn}(p,\ r) \triangleq \{m \in \mathit{SentMsgs} : m.\mathit{type} = \mathit{MoveOnMsg} \land m.r = r \land m.\mathit{to} = p\} \\
\mathit{CurDecide}(p) \triangleq \{m \in \mathit{SentMsgs} : m.\mathit{type} = \mathit{DecMsg} \land m.\mathit{to} = p\} \\
\mathit{Procs}(\mathit{msgs}) \triangleq \{p \in P : \exists\, m \in \mathit{msgs} : m.\mathit{from} = p\} \\
\mathit{SurvivorSetPred}(\mathit{Ps}) \triangleq \exists\, S \in \mathit{SurvivorSets} : \forall\, p \in S : p \in \mathit{Ps} \\
\mathit{NewEstimate}(\mathit{Msgs}) \triangleq \text{CHOOSE } v \in V : \exists\, m1 \in \mathit{Msgs} : \begin{array}[t]{l} \land\ m1.v = v \\ \land\ \forall\, m2 \in \mathit{Msgs} : m1.\mathit{updr} \geq m2.\mathit{updr} \end{array}
\end{array}
$$

$$
\mathit{RoundValue}(p,\ r) \triangleq \text{CHOOSE } v \in V : \forall\, m \in \mathit{CurEchoes}(p,\ r) : m.v = v
$$

---

$$
\begin{array}{ll}
\mathit{ACCTypeOk} \triangleq & \land\ \mathit{Estimate} \in [P \rightarrow V] \\
& \land\ \mathit{EstUpdate} \in [P \rightarrow \mathit{RoundRange}] \\
& \land\ \mathit{Decision} \in [P \rightarrow V \cup \{\mathit{NULL}\}] \\
& \land\ \mathit{MyStage} \in [P \rightarrow \mathit{States}] \\
& \land\ \mathit{Round} \in [P \rightarrow \mathit{RoundRange}] \\
& \land\ \mathit{Faulty} \in \text{SUBSET } P \\
& \land\ \mathit{SentMsgs} \in \text{SUBSET } (\mathit{EstimateMsgs} \cup \\
& \qquad\qquad\qquad \mathit{CoordEstMsgs} \cup \\
& \qquad\qquad\qquad \mathit{EchoMsgs} \cup \\
& \qquad\qquad\qquad \mathit{DecideMsgs})
\end{array}
$$

$$
\begin{array}{ll}
\mathit{ACCInit} \triangleq & \land\ \mathit{Estimate} = [p \in P \mapsto I[p]] \\
& \land\ \mathit{EstUpdate} = [p \in P \mapsto 0] \\
& \land\ \mathit{Decision} = [p \in P \mapsto \mathit{NULL}] \\
& \land\ \mathit{MyStage} = [p \in P \mapsto \mathit{InitRound}] \\
& \land\ \mathit{Round} = [p \in P \mapsto 1] \\
& \land\ \mathit{Faulty} = \{\} \\
& \land\ \mathit{SentMsgs} = \{\}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{Fail}(p) \triangleq & \land\ \exists\, S \in \mathit{SurvivorSets} : (\mathit{Faulty} \cup \{p\}) \cap S = \{\} \\
& \land\ \mathit{Faulty}' = \mathit{Faulty} \cup \{p\} \\
& \land\ \text{UNCHANGED } \langle \mathit{Estimate},\ \mathit{EstUpdate},\ \mathit{Decision}, \\
& \qquad\qquad \mathit{MyStage},\ \mathit{Round},\ \mathit{SentMsgs} \rangle
\end{array}
$$

Processes send estimate to the coordinator

$$
\begin{array}{ll}
\mathit{SendEstimate}(p) \triangleq & \land\ p \notin \mathit{Faulty} \\
& \land\ \mathit{MyStage}[p] = \mathit{InitRound} \\
& \land\ \mathit{SentMsgs}' = \mathit{SentMsgs} \cup \{[\mathit{type} \mapsto \mathit{EstMsg},\ \mathit{from} \mapsto p, \\
& \qquad\qquad \mathit{to} \mapsto \mathit{Coordinator}(\mathit{Round}[p]), \\
& \qquad\qquad r \mapsto \mathit{Round}[p], \\
& \qquad\qquad v \mapsto \mathit{Estimate}[p], \\
& \qquad\qquad \mathit{updr} \mapsto \mathit{EstUpdate}[p]]\} \\
& \land\ \text{IF } p = \mathit{Coordinator}(\mathit{Round}[p]) \\
& \qquad \text{THEN } \mathit{MyStage}' = [\mathit{MyStage} \text{ EXCEPT } ![p] = \\
& \qquad\qquad\qquad \mathit{WaitEstimates}]
\end{array}
$$

```
                ELSE $MyStage' = [MyStage \text{ EXCEPT } ![p] =$
                                        $WaitCoordEstimate]$
            $\wedge$ UNCHANGED $\langle Estimate, EstUpdate, Decision,$
                              $Round, Faulty \rangle$
```

Coordinator receive estimates from a survivor set

```
$RecEstimates(p) \triangleq$   LET $ToSend \triangleq \{[type \mapsto CoordMsg,$
                                   $from \mapsto p,$
                                   $to \mapsto q,$
                                   $r \mapsto Round[p],$
                                   $v \mapsto Estimate'[p]] :$
                                   $q \in P\}$
            IN
            $\wedge MyStage[p] = WaitEstimates$
            $\wedge SurvivorSetPred(Procs(CurEstimates(Round[p])))$
            $\wedge Estimate' = [Estimate \text{ EXCEPT } ![p] =$
                             $NewEstimate(CurEstimates(Round[p]))]$
            $\wedge EstUpdate' = [EstUpdate \text{ EXCEPT } ![p] = Round[p]]$
            $\wedge$ IF $p \in Faulty$
                THEN $\exists\, TMsgs \in \text{SUBSET } ToSend :$
                     $( \wedge SentMsgs' = SentMsgs \cup TMsgs$
                       $\wedge MyStage' = [MyStage \text{ EXCEPT } ![p] = Silent])$
                ELSE $( \wedge SentMsgs' = SentMsgs \cup ToSend$
                       $\wedge MyStage' = [MyStage \text{ EXCEPT } ![p] =$
                                              $WaitCoordEstimate])$
            $\wedge$ UNCHANGED $\langle Decision, Round, Faulty \rangle$
```

Receive Certified *Estimate* from the *Coordinator*

```
$CoordEstimate(p) \triangleq$   LET $ToSend(m) \triangleq \{[type \mapsto EchoMsg,$
                                        $from \mapsto p,$
                                        $to \mapsto q,$
                                        $r \mapsto Round[p],$
                                        $v \mapsto m.v] :$
                                        $q \in P\}$
            IN
            $\wedge MyStage[p] = WaitCoordEstimate$
            $\wedge \exists\, m \in SentMsgs :$
                $\wedge m.type \in \{CoordMsg, EchoMsg\}$
                $\wedge m.to = p$
                $\wedge m.r = Round[p]$
                $\wedge Estimate' = [Estimate \text{ EXCEPT } ![p] = m.v]$
                $\wedge EstUpdate' = [EstUpdate \text{ EXCEPT } ![p] = Round[p]]$
                $\wedge$ IF $p \in Faulty$
                   THEN $\exists\, TMsgs \in \text{SUBSET } ToSend(m) : ($
                          $\wedge SentMsgs' = SentMsgs \cup TMsgs$
                          $\wedge MyStage' = [MyStage \text{ EXCEPT } ![p] = Silent])$
                   ELSE $( \wedge SentMsgs' = SentMsgs \cup ToSend(m)$
                          $\wedge MyStage' = [MyStage \text{ EXCEPT } ![p] = WaitEchoes])$
```

```
                    $\land$ UNCHANGED $\langle Decision, Round, Faulty \rangle$

Receive Echoes
$RecEchoes(p) \triangleq$ LET $ToSend(value) \triangleq \{[type \mapsto DecMsg,$
                                  $from \mapsto p,$
                                  $to \mapsto q,$
                                  $r \mapsto Round[p],$
                                  $v \mapsto value] :$
                                  $q \in P\}$
             IN
             $\land MyStage[p] = WaitEchoes$
             $\land SurvivorSetPred(Procs(CurEchoes(p, Round[p])))$
             $\land \exists\, m \in CurEchoes(p, Round[p]) : ($
                 IF $p \in Faulty$
                 THEN $\exists\, TMsgs \in$ SUBSET $ToSend(m.v)$ :
                        $\land SentMsgs' = SentMsgs \cup TMsgs$
                        $\land MyStage' = [MyStage$ EXCEPT $![p] = Silent]$
                        $\land Decision' = Decision$
                 ELSE   $\land SentMsgs' = SentMsgs \cup ToSend(m.v)$
                        $\land MyStage' = [MyStage$ EXCEPT $![p] = Decided]$
                        $\land Decision' = [Decision$ EXCEPT $![p] =$
                                      $RoundValue(p, Round[p])])$
             $\land$ UNCHANGED $\langle Estimate, EstUpdate, Round, Faulty \rangle$

Upon suspicion, send suspicion messages
$Suspicion(p) \triangleq$ LET $ToSend \triangleq \{[type \mapsto MoveOnMsg,$
                                 $from \mapsto p,$
                                 $to \mapsto q,$
                                 $r \mapsto Round[p]] :$
                                 $q \in P\}$
            IN
            $\land MyStage[p] = WaitCoordEstimate$
            $\land p \neq Coordinator(Round[p])$
            $\land \neg(\exists\, m \in SentMsgs : ($  $\land (m.type = MoveOnMsg)$
                                          $\land (m.from = p)$
                                          $\land (m.r = Round[p])))$
            $\land$ IF $p \in Faulty$
                THEN $(\exists\, TMsgs \in$ SUBSET $ToSend$ :
                       $\land SentMsgs' = SentMsgs \cup TMsgs$
                       $\land MyStage' = [MyStage$ EXCEPT $![p] = Silent])$
                ELSE (  $\land SentMsgs' = SentMsgs \cup ToSend$
                       $\land MyStage' = MyStage)$
            $\land$ UNCHANGED $\langle Estimate, EstUpdate, Decision,$
                              $Round, Faulty \rangle$

Receive $MoveOn$ msg
$MoveOn(p) \triangleq$    $\land MyStage[p] \notin \{Silent, Decided\}$
              $\land SurvivorSetPred(Procs(CurMoveOn(p, Round[p])))$
              $\land Round' = [Round$ EXCEPT $![p] = Round[p] + 1]$
```

```
                    $\land\ MyStage' = [MyStage\ \text{EXCEPT}\ ![p] = InitRound]$
                    $\land\ \text{UNCHANGED}\ \langle Estimate, EstUpdate, Decision,$
                                              $Faulty, SentMsgs\rangle$
```

Decide

```
$Decide(p)\ \triangleq$   $\land\ MyStage[p] \notin \{Decided, Silent\}$
                  $\land\ MyStage' = [MyStage\ \text{EXCEPT}\ ![p] = Decided]$
                  $\land\ \exists\, m \in SentMsgs :$
                        $\land\ m.to = p$
                        $\land\ m.type = DecMsg$
                        $\land\ Decision' = [Decision\ \text{EXCEPT}\ ![p] = m.v]$
                  $\land\ \text{UNCHANGED}\ \langle Estimate, EstUpdate, Round,$
                                            $Faulty, SentMsgs\rangle$

$ACCStep\ \triangleq\ \exists\, p \in P :$ $\lor\ Fail(p)$
                        $\lor\ Decide(p)$
                        $\lor\ Suspicion(p)$
                        $\lor\ MoveOn(p)$
                        $\lor\ SendEstimate(p)$
                        $\lor\ RecEstimates(p)$
                        $\lor\ CoordEstimate(p)$
                        $\lor\ RecEchoes(p)$

$vars\ \triangleq\ \langle Estimate, EstUpdate, Decision, MyStage, Round, Faulty, SentMsgs\rangle$

$ACCSpec\ \triangleq\ ACCInit \land \Box[ACCStep]_{vars}$
```

---

```
$Correct\ \triangleq\ P \setminus Faulty$
$Validity\ \triangleq\ \forall\, p \in Correct :$    $\lor\ Decision[p] = NULL$
                                  $\lor\ \exists\, q \in P : Decision[p] = I[q]$
$Agreement\ \triangleq\ \forall\, p, q \in Correct :$ $\lor\ Decision[p] = NULL$
                                    $\lor\ Decision[q] = NULL$
                                    $\lor\ Decision[p] = Decision[q]$

THEOREM $ACCSpec \Rightarrow \Box ACCTypeOk$
```

# Appendix E

# **AsyncByz** specification in TLA+

```
──────────────── MODULE AsyncByzConsensus ────────────────
EXTENDS FiniteSets, Naturals, TLC

CONSTANTS P,                        Set of processes
          SurvivorSets,             Set of Survivor sets
          I,                        Initial Values of processes
          V,                        Set of decision values
          NULL,                     NULL value
          MaxRound,                 Maximum round number
          PMap,                     Map from round numbers to processes
          InitRound, WaitEstimates, WaitCertEst, WaitEchoes,
              C_WaitRoundEst, WaitRoundEst, Decided,        Stages
          EstMsg, CertEstMsg, EchoMsg, RoundEstMsg,
                 SuspMsg, MoveOnMsg, DecMsg            Msg types
```

$\textsc{assume}\ \land I \in [P \rightarrow V]$
$\qquad\land \forall S \in \mathit{SurvivorSets} : \forall e \in S : e \in P$
$\qquad\land \forall S1, S2, S3 \in \mathit{SurvivorSets} : S1 \cap S2 \cap S3 \neq \{\}$
$\qquad\land \mathit{NULL} \notin V$
$\qquad\land \mathit{PMap} \in [0 \,..\, (\mathit{Cardinality}(P) - 1) \rightarrow P]$

```
VARIABLES     Faulty,        Faulty processes
              SentMsgs,      Set of messages sent
              MyStage,       Stage of processes
              Estimate,      Estimates of processes
              EstRound,      Round in which estimate was updated
              Round,         Rounds of processes
              Decision       Decision values
```

$\mathit{RoundRange} \triangleq 0 \,..\, \mathit{MaxRound}$
$\mathit{Stages} \triangleq \{\mathit{InitRound}, \mathit{WaitEstimates}, \mathit{WaitCertEst}, \mathit{WaitEchoes},$
$\qquad\qquad \mathit{C\_WaitRoundEst}, \mathit{WaitRoundEst}, \mathit{Decided}\}$
$\mathit{MsgType} \triangleq \{\mathit{EstMsg}, \mathit{CertEstMsg}, \mathit{EchoMsg}, \mathit{RoundEstMsg},$
$\qquad\qquad \mathit{SuspMsg}, \mathit{MoveOnMsg}, \mathit{DecMsg}\}$
$\mathit{EstimateMsgs} \triangleq [\mathit{type} : \mathit{MsgType}, \mathit{from} : P, \mathit{to} : P, v : V, r : \mathit{RoundRange}]$

$CertEstimateMsgs \triangleq [type : MsgType, from : P, to : P, v : V, r : RoundRange]$
$EchoMsgs \triangleq [type : MsgType, from : P, to \ : P, v : V, r : RoundRange]$
$RoundEstMsgs \triangleq [type : MsgType, from : P, to : P, v : V, r : RoundRange]$
$SuspMsgs \triangleq [type : MsgType, from : P, to : P, r : RoundRange]$
$MoveOnMsgs \triangleq [type : MsgType, from : P, to : P, v : V,$
$\quad er : RoundRange, r : RoundRange]$
$DecideMsgs \triangleq [type : MsgType, from : P, to : P, v : V]$

$CurEstimates(p, r) \triangleq \{m \in SentMsgs : m.type = EstMsg \land m.to = p\}$
$CurCertEst(p, r) \triangleq \{m \in SentMsgs : m.type = EstMsg \ \land m.to = p\}$
$CurEchoes(p, r) \triangleq \{m \in SentMsgs : m.type = EchoMsg \land m.to = p\}$
$CurRoundEst(p, r) \triangleq \{m \in SentMsgs : m.type = RoundEstMsg \land m.to = p\}$
$CurDec(p) \triangleq \{m \in SentMsgs : m.type = DecMsg \land m.to = p\}$
$CurSusp(p, r) \triangleq \{m \in SentMsgs : m.type = SuspMsg \land m.to = p\}$
$CurMoveOn(p, r) \triangleq \{m \in SentMsgs : m.type = MoveOnMsg \land m.to = p\}$
$Procs(Msgs) \triangleq \{p \in P : \exists\, m \in Msgs : m.from = p\}$

$Coordinator(r) \triangleq PMap[r \% Cardinality(P)]$
$SurvivorSetPred(PS) \triangleq \exists\, S \in SurvivorSets : \forall\, p \in S : p \in PS$

---

$ABCTypeOk \triangleq$
$\land\ Faulty \in \textsc{subset}\ P$
$\land\ SentMsgs \ \in \textsc{subset}\ (EstimateMsgs \cup$
$\quad CertEstimateMsgs \cup$
$\quad EchoMsgs \cup$
$\quad RoundEstMsgs \cup$
$\quad SuspMsgs \cup$
$\quad MoveOnMsgs \cup$
$\quad DecideMsgs)$
$\land\ Estimate \ \in [P \rightarrow V]$
$\land\ EstRound \in [P \rightarrow RoundRange]$
$\land\ MyStage \in [P \rightarrow Stages]$
$\land\ Decision \in [P \rightarrow V \cup \{NULL\}]$
$\land\ Round \in RoundRange$

$ABCInit \triangleq$
$\land\ Faulty = \{\}$
$\land\ SentMsgs \ = \{\}$
$\land\ Estimate \ = [p \in P \mapsto I[p]]$
$\land\ EstRound = [p \in P \mapsto 0]$
$\land\ MyStage = [p \in P \mapsto InitRound]$
$\land\ Decision = [p \in P \mapsto NULL]$
$\land\ Round = [p \in P \mapsto 1]$

$Fail(p) \triangleq$
$\land\ \exists\, S \in SurvivorSets : ((Faulty \cup \{p\}) \cap S) = \{\}$
$\land\ Faulty' = Faulty \cup \{p\}$
$\land\ \textsc{unchanged}\ \langle SentMsgs, Estimate, EstRound,$
$\quad MyStage, Round, Decision\rangle$

Each process sends its own current estimate

```
$SendEstimate(p) \triangleq$   LET $ToSend \triangleq \{[type \mapsto EstMsg,$
                                      $from \mapsto p,$
                                      $to \mapsto Coordinator(Round[p]),$
                                      $v \mapsto Estimate[p],$
                                      $r \mapsto Round[p]]\}$
                               IN
                               $\land$ IF $p \in (P \setminus Faulty)$
                                    THEN  $\land MyStage[p] = InitRound$
                                          $\land SentMsgs' = SentMsgs \cup ToSend$
                                          $\land$ IF $p = Coordinator(Round[p])$
                                             THEN $MyStage' =$
                                                  $[MyStage$ EXCEPT $![p] = WaitEstimates]$
                                             ELSE $MyStage' =$
                                                  $[MyStage$ EXCEPT $![p] = WaitCertEst]$
                                    ELSE  $\land \exists SM \in$ SUBSET $ToSend :$
                                                $SentMsgs' = SentMsgs \cup SM$
                                          $\land MyStage' = MyStage$
                               $\land$ UNCHANGED $\langle Faulty, Estimate, EstRound,$
                                                 $Round, Decision\rangle$
```

Coordinator sends certified estimates

```
$SendCertEst(p) \triangleq$   LET
                              $ToSendUnique(value) \triangleq \{[type \mapsto CertEstMsg,$
                                                          $from \mapsto p,$
                                                          $to \mapsto q,$
                                                          $v \mapsto value,$
                                                          $r \mapsto Round[p]] :$
                                                          $q \in P\}$
                              $ToSendMult(Vals) \triangleq$ UNION $\{ToSendUnique(value) : value \in Vals\}$
                              $CurValues \triangleq \{v \in V : \exists S1, S2 \in SurvivorSets :$
                                                  $\land \lor S1 \subseteq Procs(CurEstimates(p, Round[p]))$
                                                    $\lor S2 \subseteq Procs(CurEstimates(p, Round[p]))$
                                                  $\land \forall q \in (S1 \cap S2) :$
                                                         $\exists m \in CurEstimates(p, Round[p]) :$
                                                            $\land m.from = q$
                                                            $\land m.v = v\}$

                              IN
                              $\land SurvivorSetPred(Procs(CurEstimates(p, Round[p])))$
                               if coordinator is not faulty
                              $\land$ IF $p \notin Faulty$
                                 THEN  $\land MyStage[p] = WaitEstimates$
                                       $\land$ IF $CurValues \neq \{\}$
                                          THEN  $\land \exists v \in CurValues :$
                                                     $SentMsgs' =$
                                                      $SentMsgs \cup ToSendUnique(v)$
                                                $\land MyStage' =$
                                                      $[MyStage$ EXCEPT $![p] = WaitCertEst]$
```

```
                    ELSE   $\land$ $SentMsgs' =$
                                $SentMsgs \cup ToSendUnique(I[p])$
                           $\land$ $MyStage' =$
                                $[MyStage$ EXCEPT $![p] = WaitCertEst]$
      Coordinator is faulty
        ELSE  IF $CurValues = \{\}$
                THEN   $\land$ $MyStage' = MyStage$
                       $\land \exists SM \in$ SUBSET $ToSendMult(V)$ :
                            $\land \forall m1 \in SM$ :
                                $\forall m2 \in (SM \setminus \{m1\})$ :
                                    $m1.to \neq m2.to$
                            $\land$ $SentMsgs' = SentMsgs \cup SM$
                ELSE   ( $\land$ $MyStage' = MyStage$
                         $\land \exists SM \in$ SUBSET $ToSendMult(CurValues)$ :
                            $\land$  $\forall m1 \in SM$ :
                                $\forall m2 \in (SM \setminus \{m1\})$ :
                                    $m1.to \neq m2.to$
                            $\land$  $SentMsgs' = SentMsgs \cup SM$)
      $\land$ $Print(SentMsgs', $ TRUE)
     $\land$ UNCHANGED $\langle Faulty, Estimate, EstRound, Round, Decision \rangle$


 Non-coordinator processes reply with echo messages
$SendEcho(p) \triangleq$ LET $ToSend(value) \triangleq \{[type \mapsto EchoMsg,$
                                          $from \mapsto p,$
                                          $to \mapsto Coordinator(Round[p]),$
                                          $v \mapsto value,$
                                          $r \mapsto Round[p]]\}$
             IN
             $\land \exists m \in SentMsgs$ : (
                 $\land$ $m.to = p$
                 $\land$ $m.type = CertEstMsg$
                 $\land$ $m.r = Round[p]$
                 $\land$ IF $p \in (P \setminus Faulty)$
                   THEN $\land$ $MyStage[p] = WaitCertEst$
                        $\land$ $SentMsgs' = SentMsgs \cup \{[type \mapsto EchoMsg,$
                                                   $from \mapsto p,$
                                                   $to \mapsto Coordinator(Round[p]),$
                                                   $v \mapsto m.v,$
                                                   $round \mapsto Round[p]]\}$
                        $\land$ IF $p = Coordinator(Round[p])$
                           THEN $MyStage' =$
                                $[MyStage$ EXCEPT $![p] = WaitEchoes]$
                           ELSE $MyStage' =$
                                $[MyStage$ EXCEPT $![p] = C\_WaitRoundEst]$
                   ELSE $\land \exists m1 \in$ SUBSET $ToSend(m.v)$ :
                            $SentMsgs' = SentMsgs \cup m1$
                        $\land$ $MyStage' = MyStage$)
             $\land$ UNCHANGED $\langle Faulty, Estimate, EstRound, Round, Decision \rangle$
```

Coordinator sends $RoundEstimate$ msgs
```
$SendRoundEstimate(p) \triangleq$ LET $ToSend(value) \triangleq \{[type \mapsto RoundEstMsg,$
                                         $from \mapsto p,$
                                         $to \mapsto q,$
                                         $v \mapsto value,$
                                         $r \mapsto Round[p]] :$
                                         $q \in P\}$
                          IN
                          $\land \exists S \in SurvivorSets : ($
                              $\land S \subseteq Procs(CurEchoes(p, Round[p]))$
                              $\land \exists v \in V : ($
                                  $\land \forall q \in S :$
                                      $\exists m \in CurEchoes(p, Round[p]) :$
                                          $( \land m.from = q$
                                           $\land m.v = v)$
                                  $\land$ IF $p \in (P \setminus Faulty)$
                                     THEN $\land MyStage[p] = WaitEchoes$
                                          $\land MyStage' =$
                                             $[MyStage$ EXCEPT $![p] = C\_WaitRoundEst]$
                                          $\land SentMsgs' = SentMsgs \cup ToSend(v)$
                                     ELSE $( \land MyStage' = MyStage$
                                           $\land \exists SM \in$ SUBSET $ToSend(v) :$
                                              $SentMsgs' = SentMsgs \cup SM)))$
                          $\land$ UNCHANGED $\langle Faulty, Estimate, EstRound, Round, Decision\rangle$
```

Forward $RoundEstimate$ message to other processes
```
$EchoRoundEstimate(p) \triangleq$ LET $ToSend(value) \triangleq \{[type \mapsto RoundEstMsg,$
                                         $from \mapsto p,$
                                         $to \mapsto q,$
                                         $v \mapsto value,$
                                         $r \mapsto Round[p]] :$
                                         $q \in P\}$
                          IN
                          $\land CurRoundEst(p, Round[p]) \neq \{\}$
                          $\land \exists m \in CurRoundEst(p, Round[p]) :$
                              IF $p \in (P \setminus Faulty)$
                               THEN    $\land SentMsgs' = SentMsgs \cup ToSend(m.v)$
                                       $\land MyStage[p] = C\_WaitRoundEst$
                                       $\land MyStage' =$
                                            $[MyStage$ EXCEPT $![p] = WaitRoundEst]$
                               ELSE    $( \land \exists SM \in$ SUBSET $ToSend(m.v) :$
                                           $SentMsgs' = SentMsgs \cup SM$
                                        $\land MyStage' = MyStage)$
                          $\land$ UNCHANGED $\langle Faulty, Estimate, EstRound, Round,$
                                       $Decision\rangle$
```

Decide if process has received $RoundEstMsg$ from a survivor set
```
$Decide(p) \triangleq$      LET $ToSend(value) \triangleq \{[type \mapsto DecMsg,$
```

```
                                   $from \mapsto p,$
                                   $to \mapsto q,$
                                   $v \mapsto value] :$
                                   $q \in P\}$
          $Values \triangleq \{v \in V :$
                              $\exists S \in SurvivorSets :$
                                 $\land S \subseteq Procs(CurRoundEst(p, Round[p]))$
                                 $\land \forall q \in S :$
                                    $\exists m \in CurRoundEst(p, Round[p]) :$
                                       $\land m.from = q$
                                       $\land m.v = v\}$
         $FaultySend \triangleq$ UNION $\{ToSend(v) : v \in Values\}$
      IN
      $\land SurvivorSetPred(Procs(CurRoundEst(p, Round[p])))$
      $\land$ IF $p \in (P \setminus Faulty)$
         THEN $\land MyStage[p] = WaitRoundEst$
              $\land \exists S \in SurvivorSets :$
                 $\exists v \in V :$
                 $\land \forall q \in S :$
                    $\exists m \in CurRoundEst(p, Round[p]) :$
                             $(\land m.from = q$
                              $\land m.v = v)$
                 $\land Decision' = [Decision$ EXCEPT $![p] = v]$
                 $\land SentMsgs' = SentMsgs \cup ToSend(v)$
                 $\land MyStage' = [MyStage$ EXCEPT $![p] = Decided]$
         ELSE $\exists SM \in$ SUBSET $FaultySend : ($
                 $\land SentMsgs' = SentMsgs \cup SM$
                 $\land Decision' = Decision$
                 $\land MyStage' = MyStage)$
      $\land$ UNCHANGED $\langle Faulty, Estimate, EstRound, Round\rangle$
```

Receive a decide message

```
$RecDec(p) \triangleq$ LET $ToSend(value) \triangleq \{[type \mapsto DecMsg,$
                                   $from \mapsto p,$
                                   $to \mapsto q,$
                                   $v \mapsto value] :$
                                   $q \in P\}$
          $Values \triangleq \{v \in V : \exists m \in CurDec(p) : m.v = v\}$
          $FaultySend \triangleq$ UNION $\{ToSend(v) : v \in Values\}$
      IN
      $\land CurDec(p) \neq \{\}$
      $\land$ IF $p \in (P \setminus Faulty)$
           THEN   $\land Decision[p] \neq NULL$
                  $\land \exists m \in CurDec(p) :$
                        $\land Decision' = [Decision$ EXCEPT $![p] = m.v]$
                        $\land SentMsgs' = SentMsgs \cup ToSend(m.v)$
                        $\land MyStage' = [MyStage$ EXCEPT $![p] = Decided]$
           ELSE   $\exists SM \in$ SUBSET $FaultySend : ($
                        $\land SentMsgs' = SentMsgs \cup SM$
                        $\land\ Decision' = Decision$
                        $\land\ MyStage' = MyStage)$
          $\land$ UNCHANGED $\langle Faulty, Estimate, EstRound, Round\rangle$

$Suspect(p) \triangleq$   LET $ToSend \triangleq \{[type \mapsto SuspMsg,$
                              $from \mapsto p,$
                              $to \mapsto q,$
                              $r \mapsto Round[p]]:$
                              $q \in P\}$
          IN
          $\land$ IF $p \in (P \setminus Faulty)$
                THEN  $\land\ MyStage[p] \in \{WaitCertEst, WaitRoundEst\}$
                      $\land\ \forall m \in CurSusp(p, Round[p]) : m.from \neq p$
                      $\land\ SentMsgs' = SentMsgs \cup ToSend$
                ELSE  $(\land\ \exists SM \in$ SUBSET $ToSend :$
                          $SentMsgs' = SentMsgs \cup SM)$
          $\land$ UNCHANGED $\langle Faulty, Estimate, EstRound, MyStage, Round, Decision\rangle$

$SendMoveOn(p) \triangleq$   LET $ToSend \triangleq \{[type \mapsto MoveOnMsg,$
                                 $from \mapsto p,$
                                 $to \mapsto q,$
                                 $v \mapsto Estimate[p],$
                                 $er \mapsto EstRound[p],$
                                 $r \mapsto Round[p]]:$
                                 $q \in P\}$
             IN
             $\land\ SurvivorSetPred(Procs(CurSusp(p, Round[p])))$
             $\land$ IF $p \in (P \setminus Faulty)$
                   THEN  $\land\ \forall m \in CurMoveOn(p, Round[p]) : m.from \neq p$
                         $\land\ SentMsgs' = SentMsgs \cup ToSend$
                   ELSE $\exists SM \in$ SUBSET $ToSend :$
                          $SentMsgs' = SentMsgs \cup SM$
             $\land$ UNCHANGED $\langle Faulty, Estimate, EstRound, MyStage,$
                             $Round, Decision\rangle$

$MoveOn(p) \triangleq$   $\land\ SurvivorSetPred(Procs(CurMoveOn(p, Round[p])))$
            $\land$ IF $p \in (P \setminus Faulty)$
                  THEN $\exists m \in CurMoveOn(p, Round[p]) :$
                         $\land\ \forall tm \in CurMoveOn(p, Round[p]) :$
                              $tm.er \leq m.er$
                         $\land\ Estimate' = [Estimate$ EXCEPT $![p] = m.v]$
                         $\land\ EstRound' = [Estimate$ EXCEPT $![p] = m.er]$
                         $\land\ Round' = [Round$ EXCEPT $![p] = Round[p] + 1]$
                         $\land\ MyStage' = [MyStage$ EXCEPT $![p] = InitRound]$
                  ELSE $\exists SM \in$ SUBSET $CurMoveOn(p, Round[p]) :$
                         $\land\ \exists S \in SurvivorSets :$
                              $\forall q \in S :$
                                  $\exists m \in SM : m.from = q$
                         $\land\ \exists m \in SM :$
```

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad \land\, \forall\, tm \in SM : tm.er \leq m.er \\
\qquad\qquad\qquad\qquad \land\, Estimate' = [Estimate \text{ EXCEPT } ![p] = m.v] \\
\qquad\qquad\qquad\qquad \land\, EstRound' = [Estimate \text{ EXCEPT } ![p] = m.er] \\
\qquad\qquad\qquad\qquad \land\, \exists\, r \in RoundRange : \ Round' = \\
\qquad\qquad\qquad\qquad\qquad [Round \text{ EXCEPT } ![p] = 1 + Round[p]] \\
\qquad\qquad\qquad\qquad \land\, MyStage' = MyStage \\
\qquad\qquad \land\, \text{UNCHANGED } \langle Faulty, SentMsgs, Decision \rangle
\end{array}
$$

$$
\begin{array}{l}
ABCStep \triangleq \exists\, p \in P : \lor\, Fail(p) \\
\qquad\qquad\qquad\qquad \lor\, SendEstimate(p) \\
\qquad\qquad\qquad\qquad \lor\, SendCertEst(p) \\
\qquad\qquad\qquad\qquad \lor\, SendEcho(p) \\
\qquad\qquad\qquad\qquad \lor\, SendRoundEstimate(p) \\
\qquad\qquad\qquad\qquad \lor\, EchoRoundEstimate(p) \\
\qquad\qquad\qquad\qquad \lor\, Decide(p) \\
\qquad\qquad\qquad\qquad \lor\, RecDec(p) \\
\qquad\qquad\qquad\qquad \lor\, Suspect(p) \\
\qquad\qquad\qquad\qquad \lor\, SendMoveOn(p) \\
\qquad\qquad\qquad\qquad \lor\, MoveOn(p)
\end{array}
$$

$$
\begin{array}{l}
vars \triangleq \langle Faulty, SentMsgs, Estimate, EstRound, MyStage, Round, Decision \rangle \\
ABCSpec \triangleq ABCInit \land \Box[ABCStep]_{vars}
\end{array}
$$

---

$$
\begin{array}{l}
StrongValidity \triangleq \exists\, v \in V : \\
\qquad\qquad ((\forall\, p \in (P \setminus Faulty) : \\
\qquad\qquad\quad I[p] = v) \ \Rightarrow (\forall\, p \in (P \setminus Faulty) : \\
\qquad\qquad\qquad\qquad\qquad \lor\, Decision[p] = NULL \\
\qquad\qquad\qquad\qquad\qquad \lor\, Decision[p] = v))
\end{array}
$$

$$
\begin{array}{l}
Agreement \triangleq \exists\, v \in V : \forall\, p \in (P \setminus Faulty) : \quad \lor\, Decision[p] = NULL \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \lor\, Decision[p] = v
\end{array}
$$

THEOREM $ABCSpec \Rightarrow \Box ABCTypeOk$

# Appendix F

# ROC specification in TLA+

```
MODULE ROConsensus
EXTENDS Naturals, FiniteSets, TLC

CONSTANTS P,               Set of processes
          Vals,            Domain of d values
          NULL,            A null value
          SurvivorSet,     All survivor sets
          I                Initial values

VARIABLES A,              Accumulated values of a process
          Aprime,         Accumulated values of a process (previous round)
          d,              Decision value of a process
          messages,       Sent messages
          crashed,        Crashed processes
          faulty,         R-O faulty processes
          round,          Current round
          pRound,         Round that p is in
          MyTurn,
          recdFrom       recdFrom[p][r] is the set of processes
                           p received a message from in round r.

N ≜ Cardinality(P)

ASSUMPTION P ⊆ Nat

  SurvivorSet satisfies (3, 2)-Intersection
ASSUMPTION ∧ ∀ s ∈ SurvivorSet : s ⊆ P
           ∧ ∀ s1, s2, s3 ∈ SurvivorSet :
                ∨ s1 ∩ s2 ≠ {}
                ∨ s2 ∩ s3 ≠ {}
                ∨ s1 ∩ s3 ≠ {}
           ∧ ∀ p ∈ P : ∃ s ∈ SurvivorSet : p ∈ s

ASSUMPTION I ∈ [P → Vals]

  Used for number of rounds
```

```
$t \triangleq$ LET $x \triangleq$ CHOOSE $s \in SurvivorSet$ :
                    $\forall s2 \quad \in SurvivorSet$ :
                      $Cardinality(s) \leq Cardinality(s2)$
      IN  $N - Cardinality(x)$
```

---

```
$ROETypeOK \triangleq$ $\land A \in [P \rightarrow [P \rightarrow Vals \cup \{NULL\}]]$
                $\land Aprime \in [P \rightarrow [P \rightarrow Vals \cup \{NULL\}]]$
                $\land d \in [P \rightarrow [P \rightarrow Vals \cup \{NULL\}] \cup \{NULL\}]$
                $\land messages \subseteq [$
                     $from : P,$
                     $to : P,$
                     $round : 0 \mathrel{..} t,$
                     $val : [P \rightarrow Vals \cup \{NULL\}]]$
                $\land faulty \subseteq P$
                $\land crashed \subseteq P$
                $\land round \in 0 \mathrel{..} (t + 1)$
                $\land pRound \in [P \rightarrow 0 \mathrel{..} (t + 1)]$
                $\land Cardinality(faulty) + Cardinality(crashed) \leq t$
                $\land faulty \cap crashed = \{\}$
                $\land MyTurn \in P$

  $\land A \in [P \rightarrow [P \rightarrow Vals \cup \{NULL\}]]$
  $\land \forall p, q \in P$: IF $p = q$ THEN $A[p][q] = I[p]$
                    ELSE $A[p][q] = NULL$
$ROEInit \triangleq$ $\land A = [p \in P \mapsto [q \in P \mapsto$ IF $p = q$ THEN $I[p]$ ELSE $NULL]]$
            $\land Aprime = A$
            $\land d = [p \in P \mapsto NULL]$
            $\land messages = \{\}$
            $\land faulty = \{\}$
            $\land crashed = \{\}$
            $\land round = 0$
            $\land pRound = [p \in P \mapsto 0]$
            $\land recdFrom = [p \in P \mapsto [r \in 0 \mathrel{..} (t + 1) \mapsto$
                              IF $r = 0$ THEN $P$ ELSE $\{\}]]$
            $\land MyTurn =$ CHOOSE $p \in P : 1 = 1$
```

---

```
 Send message $v$ in round $r$
$Send(from, to, r, v) \triangleq$
  $\land from \notin crashed$
  $\land messages' = messages \cup$
      $\{[from \mapsto from, to \mapsto to, round \mapsto r, val \mapsto v]\}$

 Process $p$ has decided
$Decided(p) \triangleq d[p] \neq NULL$

 Consensus has terminated
$Terminated \triangleq \forall p \in P \setminus crashed : Decided(p)$
```

```
  Sources of messages
$From(msgs) \triangleq \{p \in P : \exists\, m \in msgs : m.from = p\}$
```

---

```
$LastStep(p) \triangleq$ $\land\ p \notin crashed$
                        $\land\ round = t + 1$
                        $\land\ pRound[p] = t + 1$
                        $\land\ \neg Decided(p)$
                        $\land\ d' = [d$ EXCEPT $![p] = A[p]]$
                        $\land$ UNCHANGED $\langle A, Aprime, messages, crashed,$
                                          $faulty, round, pRound, recdFrom,$
                                          $MyTurn\rangle$

$Fail(p) \triangleq$ $\land\ \exists\, s \in SurvivorSet : s \subseteq (P \setminus (faulty \cup crashed \cup \{p\}))$
                $\land\ \lor\ \land\ p \notin crashed$      $\cup\ faulty$
                        $\land\ faulty' = faulty \cup \{p\}$
                        $\land$ UNCHANGED $crashed$
                  $\lor\ \land\ p \notin crashed$
                        $\land\ crashed' = crashed \cup \{p\}$
                        $\land\ faulty' = faulty \setminus \{p\}$
                $\land$ UNCHANGED $\langle A, Aprime, d, messages, round, pRound,$
                                 $recdFrom, MyTurn\rangle$

  Each round (except round 0) starts by receiving messages sent in the previous round
  and terminates (except round $t + 1$) by sending $A[p]$ to all processes.

$RoundDone(r) \triangleq$ $\land\ round = r$
                         $\land\ \forall\, p \in P \setminus crashed :$
                               $\lor\ Decided(p)$
                               $\lor\ \forall\, q \in P :$
                                   $\exists\, m \in messages :$ $\land\ m.from = p$
                                                           $\land\ m.to = q$
                                                           $\land\ m.round = r$
                         $\land\ round' = round + 1$
                         $\land$ UNCHANGED $\langle A, Aprime, d, messages, crashed, faulty,$
                                          $pRound, recdFrom, MyTurn\rangle$

$RecvRound1(p) \triangleq$ LET $msgsSentToMe \triangleq \{m \in messages : (m.to = p \land m.round = 0)\}$
                              Am faulty if didn't receive messages from a survivor set
                            $IMustBeFaulty(mset) \triangleq$
                                  $\lor\ \neg(From(mset) \subseteq recdFrom[p][0])$
                                  $\lor\ \neg(\exists\, s \in SurvivorSet : s \subseteq From(mset))$
                          IN
                           $\land\ MyTurn = p$
                           $\land\ round = 1$
                           $\land\ p \notin crashed$
                           $\land\ pRound[p] = 0$
                           $\land\ \neg Decided(p)$
                           $\land\ pRound' = [pRound$ EXCEPT $![p] = 1]$
                           $\land\ \exists\, msgsRecd \in$ SUBSET $msgsSentToMe :$
                                 $\land\ (p \in faulty \lor\ msgsRecd = msgsSentToMe)$
```

```
            $\land p \in From(msgsRecd)$
            $\land recdFrom' = [recdFrom$ EXCEPT $![p][1] = From(msgsRecd)]$
            $\land$ IF $IMustBeFaulty(msgsRecd)$
              THEN $\land d' = [d$ EXCEPT $![p] = [q \in P \mapsto NULL]]$
                   $\land$ UNCHANGED $A$
              ELSE $\land A' = [A$ EXCEPT $![p] = [q \in P \mapsto$
                        CASE $A[p][q] \neq NULL \rightarrow A[p][q]$
                        $\Box$ $\land (A[p][q] = NULL)$
                           $\land (\forall m \in msgsRecd : m.val[q] = NULL)$
                           $\rightarrow A[p][q]$
                        $\Box$ $\land (A[p][q] = NULL)$
                           $\land (\exists m \in msgsRecd : m.val[q] \neq NULL)$
                           $\rightarrow$ CHOOSE $v \in Vals : (\exists m \in msgsRecd :$
                                                   $m.val[q] = v)$
                        $]]$
                   $\land$ UNCHANGED $d$
          $\land$ UNCHANGED $\langle Aprime, messages, crashed, faulty, round, MyTurn \rangle$

$RecvRoundR(p, r) \triangleq$ LET $msgsSentToMe \triangleq \{m \in messages :$
                                        $(m.to = p \land m.round = r - 1)\}$
          Am faulty if received a message from a process in this round
            not in last round OR
          Removing messages from obviously faulty processes, did not receive
            messages from a survivor set.
          $IMustBeFaulty(mset) \triangleq$
              $\lor \neg(From(mset) \subseteq recdFrom[p][r-1])$
              $\lor \neg(\exists s \in SurvivorSet : s \subseteq$
                  $From(\{m \in mset :$
                    $\forall i \in P : ( \lor (Aprime[p][i] = NULL)$
                                  $\lor (m.val[i] = Aprime[p][i]))\}))$
        IN
        $\land MyTurn = p$
        $\land round = r$
        $\land p \notin crashed$
        $\land pRound[p] = r - 1$
        $\land \neg Decided(p)$
        $\land pRound' = [pRound$ EXCEPT $![p] = r]$
        $\land Aprime' = [Aprime$ EXCEPT $![p] = A[p]]$
        $\land \exists msgsRecd \in$ SUBSET $msgsSentToMe :$
            $\land (p \in faulty \lor msgsRecd = msgsSentToMe)$
            $\land p \in From(msgsRecd)$
            $\land recdFrom' = [recdFrom$ EXCEPT $![p][r] = From(msgsRecd)]$
            $\land$ IF $IMustBeFaulty(msgsRecd)$
              THEN $\land d' = [d$ EXCEPT $![p] = [q \in P \mapsto NULL]]$
                   $\land$ UNCHANGED $A$
              ELSE $\land A' = [A$ EXCEPT $![p] = [q \in P \mapsto$
                        CASE $A[p][q] \neq NULL \rightarrow A[p][q]$
                        $\Box$ $\land (A[p][q] = NULL)$
                           $\land (\forall m \in msgsRecd : m.val[q] = NULL)$
```

```
                                          $\rightarrow A[p][q]$
                                        $\Box$  $\land (A[p][q] = NULL)$
                                             $\land (\exists\, m \in msgsRecd : m.val[q] \neq NULL)$
                                            $\rightarrow$ CHOOSE $v \in Vals : (\exists\, m \in msgsRecd :$
                                                                  $m.val[q] = v)$
                                        $]]$
                                   $\land$ UNCHANGED $d$
                          $\land$ UNCHANGED $\langle messages, crashed, faulty, round, MyTurn\rangle$
```

```
$SendRound(p, r) \triangleq$ LET $sentTo \triangleq \{q \in P : \exists\, m \in messages :$
                              $(m.from = p \land m.to = q \land m.round = r)\}$
                            IN
                            $\land MyTurn = p$
                            $\land p \notin crashed$
                            $\land \neg Decided(p)$
                            $\land round = r$
                            $\land pRound[p] = r$
                            $\land \exists\, q \in P : \land q \notin sentTo$
                                              $\land Send(p, q, r, A[p])$
                            $\land$ UNCHANGED $\langle A, Aprime, d, crashed, faulty, round, pRound,$
                                                  $recdFrom, MyTurn\rangle$
```

```
$NextP(r) \triangleq$ LET $sentTo \triangleq \{q \in P : \exists\, m \in messages :$
                          $(m.from = MyTurn \land m.to = q \land m.round = r)\}$
                    IN
                    $\land \lor \forall\, q \in P : q \in sentTo$
                       $\lor MyTurn \in crashed$
                    $\land MyTurn' =$ CHOOSE $p \in P : p \neq MyTurn \land p \notin crashed$
                    $\land$ UNCHANGED $\langle A, Aprime, d, messages, crashed, faulty, round,$
                                          $pRound, recdFrom\rangle$
```

```
$ROENext \triangleq \lor \exists\, p \in P : Fail(p)$
                 $\lor \exists\, p \in P : \lor RecvRound1(p)$
                                  $\lor LastStep(p)$
                                  $\lor \exists\, r \in 0 \,..\, t : \lor RoundDone(r)$
                                                     $\lor SendRound(p, r)$
                                  $\lor \exists\, r \in 2 \,..\, (t+1) : RecvRoundR(p, r)$
                                  $\lor \exists\, r \in 0 \,..\, (t+1) : NextP(r)$
```

```
$vars \triangleq \langle A, Aprime, d, messages, crashed, faulty, round, pRound,$
           $recdFrom, MyTurn\rangle$
```

Since behaviors are finite ignoring stuttering, ignore liveness

$ROESpec \triangleq ROEInit \land \Box[ROENext]_{vars}$

---

Some sets of processes

$NotCrashed \triangleq P \setminus crashed$

$NotFaulty \triangleq (P \setminus crashed) \setminus faulty$

$$DecidedSomething \triangleq \{p \in P : Decided(p)\}$$

$$\begin{array}{l} DecidedSomethingInteresting \triangleq \{p \in DecidedSomething : \\ \qquad\qquad\qquad\qquad\qquad\qquad \exists\, q \in P : d[p][q] \neq NULL\} \end{array}$$

---

Relations on decisions

$$dvSubsetEq(d1,\ d2) \triangleq \forall\, r \in P : (d1[r] \neq NULL) \Rightarrow (d1[r] = d2[r])$$

$$dSubsetEq(p,\ q) \triangleq dvSubsetEq(d[p],\ d[q])$$

$$dSubset(p,\ q) \triangleq dSubsetEq(p,\ q) \land (d[p] \neq d[q])$$

RO-Consensus safety properties

$$\begin{array}{l}
GoodDecision \triangleq \land \forall\, p \in NotFaulty,\ q \in P : \\
\qquad\qquad\qquad\qquad \text{IF } q \in NotCrashed \text{ THEN } d[p][q] = I[q] \\
\qquad\qquad\qquad\qquad\quad \text{ELSE } d[p][q] \in \{NULL,\ I[q]\} \\
\qquad\qquad\qquad \land \forall\, v1,\ v2 \in Vals : \\
\qquad\qquad\qquad\qquad \forall\, S1,\ S2 \in SurvivorSet : \\
\qquad\qquad\qquad\qquad\qquad (\lor \exists\, p \in S1 : I[p] \neq v1 \\
\qquad\qquad\qquad\qquad\qquad\ \ \lor \exists\, p \in S2 : (\lor I[p] \neq v2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \lor p \in faulty) \\
\qquad\qquad\qquad\qquad\qquad\ \ \lor \forall\, p1,\ p2 \in P : (\lor d[p1] = NULL \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \lor d[p1][p2] \neq v1) \\
\qquad\qquad\qquad\qquad\qquad\ \ \lor \forall\, p \in NotCrashed : \exists\, q \in P : d[p][q] = v1) \\
\qquad\qquad\qquad \land \forall\, p \in NotCrashed : \forall\, q \in P : (\lor d[p][q] = NULL \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \lor \forall\, c \in NotFaulty : \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad d[c][q] = d[p][q]) \\
\qquad\qquad\qquad \land \text{LET } vals \triangleq \{d[p] : p \in DecidedSomethingInteresting\} \\
\qquad\qquad\qquad\quad \text{IN } \land Cardinality(vals) \leq 2 \\
\qquad\qquad\qquad\qquad\ \ \land Cardinality(vals) \geq 1 \\
\qquad\qquad\qquad\qquad\ \ \land \forall\, d1,\ d2 \in vals : dvSubsetEq(d1,\ d2) \lor dvSubsetEq(d2,\ d1) \\
\qquad\qquad\qquad\qquad\ \ \land \forall\, d1,\ d2 \in vals : \\
\qquad\qquad\qquad\qquad\qquad\quad \lor d1 = d2 \\
\qquad\qquad\qquad\qquad\qquad\quad \lor \exists\, S1,\ S2 \in SurvivorSet : \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\land \forall\, p \in S1 : (\lor p \in crashed \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \lor d[p] = d1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \land \forall\, p \in S2 : (\land d[p] = d2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land p \in NotFaulty))
\end{array}$$

$$OnlyGoodDecisions \triangleq Terminated \Rightarrow GoodDecision$$

$$\text{THEOREM } ROESpec \Rightarrow \Box ROETypeOK$$
$$\text{THEOREM } ROESpec \Rightarrow \Box OnlyGoodDecisions$$