Rachid Guerraoui, Luís Rodrigues

Introduction to Distributed Algorithms

(Preliminary Draft)

November 22, 2004

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

To whom it might concern.

Preface

This manuscript aims at offering an introductory description of distributed programming abstractions and of the algorithms that are used to implement them in different distributed environments. The reader is provided with an insight on important problems in distributed computing, knowledge about the main algorithmic techniques that can be used to solve these problems, and examples of how to apply these techniques when building distributed applications.

Content

In modern computing, a program usually executes on *several* processes: in this context, a process is an abstraction that may represent a computer, a processor within a computer, or simply a specific thread of execution within a processor. The fundamental problem in devising such distributed programs usually consists in having the processes *cooperate* on some *common* task. Of course, traditional centralized algorithmic issues, on each process individually, still need to be dealt with. The added difficulty here is about achieving a robust form of cooperation, despite failures or disconnections of some of the processes, inherent to most distributed environments.

Had no notion of cooperation been required, a distributed program would simply consist of a set of detached centralized programs, each running on a specific process, and little benefit could be obtained from the availability of several machines in a distributed environment. It was the need for cooperation that revealed many of the fascinating problems addressed by this manuscript, problems that would have otherwise remained undiscovered. The manuscript, not only exposes the reader to these problems but also presents ways to solve them in different contexts.

Not surprisingly, distributed programming can be significantly simplified if the difficulty of robust cooperation is encapsulated within specific *abstractions*. By encapsulating all the tricky algorithmic issues, such distributed programming abstractions bridge the gap between network communication layers, usually frugal in terms of reliability guarantees, and distributed application layers, usually demanding in terms of reliability.

The manuscript presents various distributed programming abstractions and describes algorithms that implement these abstractions. In a sense, we give the distributed application programmer a library of abstraction interface specifications, and the distributed system builder a library of algorithms that implement the specifications.

A significant amount of the preparation time of this manuscript was devoted to preparing the exercises and working out their solutions. We strongly encourage the reader to work out the exercises. We believe that no reasonable understanding can be achieved in a passive way. This is especially true in the field of distributed computing where the underlying anthropomorphism may provide fast but wrong intuitions. Many exercises are rather easy and can be discussed within an undergraduate teaching classroom. Some exercises are more difficult and need more time. These can be given as homeworks.

The manuscript comes with a companion set of running examples implemented in the Java programming language, using the *Appia* protocol composition framework. These examples can be used by students to get a better understanding of the implementation details not covered in the high-level description of the algorithms. Instructors can use these protocol layers as a basis for practical exercises, by suggesting students to perform optimizations on the code provided, to implement variations of the algorithms for different system models, or to design applications that make use of these abstractions.

Presentation

The manuscript is written in a self-contained manner. This has been made possible because the field of distributed algorithms has reached a certain level of maturity where details, for instance about the network and various kinds of failures, can be abstracted away when reasoning about the distributed algorithms. Elementary notions of algorithms, first order logics, programming languages, networking, and operating systems might be helpful, but we believe that most of our abstraction specifications and algorithms can be understood with minimal knowledge about these notions.

The manuscript follows an incremental approach and was primarily built as a textbook for teaching at the undergraduate or basic graduate level. It introduces basic elements of distributed computing in an intuitive manner and builds sophisticated distributed programming abstractions on top of more primitive ones. Whenever we devise algorithms to implement a given abstraction, we consider a simple distributed system model first, and then we revisit the algorithms in more challenging models. In other words, we first devise algorithms by making strong simplifying assumptions on the distributed environment and then we discuss how to weaken those assumptions.

We have tried to balance intuition and presentation simplicity, on one hand, with rigour, on the other hand. Sometimes rigour was impacted, and this might not have been always on purpose. The focus here is rather on abstraction specifications and algorithms, not on calculability and complexity. Indeed, there is no theorem in this manuscript. Correctness arguments are given with the aim of better understanding the algorithms: they are not formal correctness proofs per se. In fact, we tried to avoid Greek letters and mathematical notations: references are given to papers with more formal treatment of some of the material presented here.

Organization

- In Chapter 1 we *motivate* the need for distributed programming abstractions by discussing various applications that typically make use of such abstractions. The chapter also presents the programming notations used in the manuscript to describe specifications and algorithms.
- In Chapter 2 we present different kinds of *assumptions* that we will be making about the underlying distributed environment, i.e., we present different distributed system models. Basically, we describe the basic abstractions on which more sophisticated ones are built. These include process and communication link abstractions. This chapter might be considered as a reference throughout other chapters.

The rest of the chapters are each devoted to one family of related abstractions, and to various algorithms implementing them.

- In Chapter 3 we introduce specific distributed programming abstractions: those related to the *reliable delivery* of messages that are *broadcast* to a group of processes. We cover here issues such as how to make sure that a message delivered by one process is delivered by all, despite the crash of the original sender process.
- In Chapter 4 we discuss *shared memory* abstractions which encapsulate simple forms of distributed storage objects with read-write semantics, e.g., files and register abstractions. We cover here issues like how to ensure that a value written (stored) within a set of processes is eventually read (retrieved) despite the crash of some of the processes.
- In Chapter 5 we address the *consensus* abstraction through which a set of processes can decide on a common value, based on values, each process initially proposed, despite the crash of some of the processes. This abstraction is key in building abstractions described in subsequent chapters.
- In Chapter 6 we consider *ordering* abstractions. In particular, we discuss the causal ordering of messages that are broadcast in the system, as well as a how consensus can be used to ensure totally ordered delivery of messages. The resulting total order abstraction makes it easy to implement sophisticated forms of shared distributed objects, beyond read-write shared memory objects.

- In Chapter 7 we gather what we call *coordination* abstractions. These include leader election, terminating reliable broadcast, non-blocking atomic commit and group membership.

The distributed algorithms we will study differ naturally according to the actual abstraction they aim at implementing, but also according to the assumptions on the underlying distributed environment (we will also say distributed system model), i.e., on the initial abstractions they take for granted. Aspects such as the reliability of the links, the degree of synchrony of the system, whether a deterministic or a randomized (probabilistic) solution is sought, have a fundamental impact on how the algorithm is designed. To give the reader an insight of how these parameters affect the algorithm design, the manuscript includes several classes of algorithmic solutions to implement the same distributed programming abstractions for various distributed system models.

Covering all chapters, with their associated exercises, constitutes a full course in the field. Focusing for each chapter on the specifications of the abstractions and their underlying algorithms in their simplest form (i.e., for the simplest model of computation considered in the manuscript), would constitute a shorter, more elementary course. This can provide a nice companion to a more practically oriented course possibly based on our protocol framework.

References

We have been exploring the world of distributed programming abstractions for more than a decade now. During this period, we were influenced by many researchers in the field of distributed computing. A special mention to Leslie Lamport and Nancy Lynch for having posed fascinating problems in distributed computing, and to the Cornell-Toronto *school*, including Ken Birman, Tushar Chandra, Vassos Hadzilacos, Robert van Renessee, Fred Schneider, and Sam Toueg, for their seminal work on various forms of distributed programming abstractions.

Many other researchers have directly or indirectly inspired the material of this manuscript. We did our best to reference their work throughout the text. Most chapters end with a historical note. This intends to provide hints for further readings, to trace the history of the concepts presented in the chapters, as well as to give credits to those who invented and worked out the concepts. At the end of the manuscript, we reference other manuscripts for further readings on other aspects of distributed computing.

Acknowledgements

We would like to express our gratitude to our undergraduate and graduate students from the Swiss Federal Institute of Technology in Lausanne (EPFL) and the University of Lisboa (UL), for serving as reviewers of preliminary drafts of this manuscript. Indeed they had no choice and needed to prepare their exams anyway. But they were indulgent toward the bugs and typos that could be found in earlier versions of the manuscript as well as associated slides, and they did provide us with useful feedback.

Partha Dutta, Corine Hari, Ron Levy, Petr Kouznetsov and Bastian Pochon, graduate students at the Distributed Programming Laboratory of the Swiss Federal Institute of Technology in Lausanne (EPFL) at the time of writing this manuscript, as well as Filipe Araújo, and Hugo Miranda, graduate students at the Distributed Algorithms and Network Protocol (DIALNP) group at the Departamento de Informtica da Faculdade de Cincias da Universidade de Lisboa (UL), at the same period, suggested many improvements to the algorithms presented in the manuscript. Several of the implementation for the "hands-on" part ofthe book were developed by or with the help of several DIALNP team members and students, including Nuno Carvalho, Maria João Monteiro, Alexandre Pinto, and Luís Sardinha.

Finally, we would like to thank all several of our colleagues who were kind enough to read and comment earlier drafts of this book. These include Lorenzo Alvisi, Roberto Baldoni, Carole Delporte, Hugues Fauconnier, Pascal Felber, Felix Gaertner, Anne-Marie Kermarrec, Fernando Pedone, Michel Raynal, and Marten Van Steen.

Rachid Guerraoui and Luís Rodrigues

Contents

1. Introduction

God does not often clap his hands. When he does, every body should dance
(African Proverb)

This chapter first motivates the need for distributed programming abstractions. Special attention is given to abstractions that capture the problems that underly robust forms of cooperations between multiple processes in a distributed system, such as agreement abstractions. The chapter then advocates a modular strategy for the development of distributed programs by making use of those abstractions through specific Application Programming Interfaces (APIs).

A concrete simple example API is also given to illustrate the notation and event-based invocation scheme used throughout the manuscript to describe the algorithms that implement our abstractions. The notation and invocation schemes are very close to those we have used to implement our algorithms in our Appia protocol framework.

1.1 Motivation

Distributed computing has to do with devising algorithms for a set of processes that seek to achieve some form of cooperation. Besides executing concurrently, some of the processes of a distributed system might stop operating, for instance by crashing or being disconnected, while others might stay alive and keep operating. This very notion of *partial failures* is a characteristic of a distributed system. In fact, this can be useful if one really feels the need to differentiate a distributed system from a concurrent system. It is usual to quote Leslie Lamport here:

> "A distributed system is one in which the failure of a computer you did not even know existed can render your own computer unusable".

When a subset of the processes have failed, or got disconnected, the challenge is for the processes that are still operating to synchronize their activities in a consistent way. In other words, the cooperation must be made robust to tolerate partial failures. This makes distributed computing quite hard, yet extremely stimulating, problem. As we will discuss in detail later in the manuscript, due to several factors such as the asynchrony of the underlying components and the possibility of failures in the communication infrastructure, it may be impossible to accurately detect process failures, and in particular distinguish a process failure from a network failure. This makes the problem of ensuring a consistent cooperation even more difficult. The challenge of researchers in distributed computing is precisely to devise algorithms that provide the processes that remain operating with enough consistent information so that they can cooperate correctly and solve common tasks.

In fact, many programs that we use today are distributed programs. Simple daily routines, such as reading e-mail or browsing the web, involve some form of distributed computing. However, when using these applications, we are typically faced with the simplest form of distributed computing: *client-server* computing. In client-server computing, a centralized process, the *server*, provides a service to many remote *clients*. The clients and the server communicate by exchanging messages, usually following a request-reply form of interaction. For instance, in order to display a web page to the user, a browser sends a request to the WWW server and expects to obtain a response with the information to be displayed. The core difficulty of distributed computing, namely achieving a consistent form of cooperation in the presence of partial failures, may be revealed even by using this simple form of interaction. Going back to our browsing example, it is reasonable to expect that the user continues surfing the web if the site it is consulting fails (by automatically switching to other sites), and even more reasonable that the server process keeps on providing information to the other client processes, even when some of them fail or got disconnected.

The problems above are already difficult to deal with when distributed computing is limited to the interaction between two parties, such as in the client-server case. However, there is more to distributed computing than client-server computing. Quite often, not only two, but several processes need to cooperate and synchronize their actions to achieve a common goal. The existence of not only two, but multiple processes does not make the task of distributed computation any simpler. Sometimes we talk about *multi-party* interactions in this general case. In fact, both patterns might coexist in a quite natural manner. Actually, a real distributed application would have parts following a client-server interaction pattern and other parts following a multi-party interaction one. This might even be a matter of perspective. For instance, when a client contacts a server to obtain a service, it may not be aware that, in order to provide that service, the server itself may need to request the assistance of several other servers, with whom it needs to coordinate to satisfy the client's request.

1.2 Distributed Programming Abstractions

Just like the act of smiling, the act of abstraction is restricted to very few natural species. By capturing properties which are common to a large and significant range of systems, abstractions help distinguish the fundamental from the accessory and prevent system designers and engineers from reinventing, over and over, the same solutions for the same problems.

From The Basics. Reasoning about distributed systems should start by abstracting the underlying physical system: describing the relevant components in an abstract way, identifying their intrinsic properties, and characterizing their interactions, leads to what is called a *system model*. In this book we will use mainly two abstractions to represent the underlying physical system: *processes* and *links*.

The processes of a distributed program abstract the active entities that perform computations. A process may represent a computer, a processor within a computer, or simply a specific thread of execution within a processor. To cooperate on some common task, the processes might typically need to exchange messages using some communication network. Links abstract the physical and logical network that supports communication among processes. It is possible to represent different realities of a distributed system by capturing different properties of processes and links, for instance, by describing the different ways these components may fail. Chapter 2 will provide a deeper discussion on the various distributed systems models that are used in this book.

To The Advanced. Given a system model, the next step is to understand how to build abstractions that capture recurring interaction patterns in distributed applications. In this book we are interested in abstractions that capture robust cooperation problems among groups of processes, as these are important and rather challenging. The cooperation among processes can sometimes be modelled as a distributed *agreement* problem. For instance, the processes may need to agree if a certain event did (or did not) take place, to agree on a common sequence of actions to be performed (from a number of initial alternatives), to agree on the order by which a set of inputs need to be processed, etc. It is desirable to establish more sophisticated forms of agreement from solutions to simpler agreement problems, in an incremental manner. Consider for instance the following problems:

- In order for processes to be able to exchange information, they must initially agree on who they are (say using IP addresses) and some common format for representing messages. They might also need to agree on some reliable way of exchanging messages (say to provide TCP-like semantics).

- After exchanging some messages, the processes may be faced with several alternative plans of action. They may then need to reach a *consensus* on a common plan, from all alternatives, and each participating process may have initially its own plan, different from the plans of the remaining processes.
- In some cases, it may be only acceptable for the cooperating processes to take a given step if all other processes also agree that such a step should take place. If this condition is not met, all processes must agree that the step should *not* take place. This form of agreement is utmost importance in the processing of distributed transactions, where this problem is known as the *atomic commitment* problem.
- Processes may need not only to agree on which actions they should execute but to agree also on the order by which these actions need to be executed. This form of agreement is the basis of one of the most fundamental techniques to replicate computation in order to achieve fault-tolerance, and it is called the *total order* problem.

This book is about mastering the difficulty underlying these problems, and devising *abstractions* that encapsulate such problems. In the following, we try to motivate the relevance of some of the abstractions covered in this manuscript. We distinguish the case where the abstractions pop up from the natural distribution of the abstraction, from the case where these abstractions come out as artifacts of an engineering choice for distribution.

1.2.1 Inherent Distribution

Applications which require sharing or dissemination of information among several participant processes are a fertile ground for the emergence of distributed programming abstractions. Examples of such applications are information dissemination engines, multi-user cooperative systems, distributed shared spaces, cooperative editors, process control systems, and distributed databases.

Information Dissemination. In distributed applications with information dissemination requirements, processes may play one of the following roles: information producers, also called *publishers*, or information consumers, also called *subscribers*. The resulting interaction paradigm is often called *publish-subscribe*.

Publishers produce information in the form of notifications. Subscribers register their interest in receiving certain notifications. Different variants of the paradigm exist to match the information being produced with the subscribers interests, including channel-based, subject-based, content-based or type-based subscriptions. Independently of the subscription method, it is very likely that several subscribers are interested in the same notifications, which will then have to be multicast. In this case, we are typically interested in having subscribers of the same information receiving the same set of messages.

Otherwise the system will provide an unfair service, as some subscribers could have access to a lot more information than other subscribers.

Unless this reliability property is given for free by the underlying infrastructure (and this is usually not the case), the sender and the subscribers may need to coordinate to agree on which messages should be delivered. For instance, with the dissemination of an audio stream, processes are typically interested in receiving most of the information but are able to tolerate a bounded amount of message loss, especially if this allows the system to achieve a better throughput. The corresponding abstraction is typically called a *best-effort broadcast*.

The dissemination of some stock exchange information might require a more reliable form of broadcast, called *reliable broadcast*, as we would like all active processes to receive the same information. One might even require from a strock exchange infrastructure that information be disseminated in an ordered manner. The adequate communication abstraction that offers ordering in addition to reliability is called *total order broadcast*. This abstraction captures the need to disseminate information, such that all participants can get a consistent view of the global state of the disseminated information.

In several publish-subscribe applications, producers and consumers interact indirectly, with the support of a group of intermediate cooperative brokers. In such cases, agreement abstractions might be useful for the cooperation of the brokers.

Process Control. Process control applications are those where several software processes have to control the execution of a physical activity. Basically, the (software) processes might be controlling the dynamic location of an aircraft or a train. They might also be controlling the temperature of a nuclear installation, or the automation of a car production system.

Typically, every process is connected to some sensor. The processes might for instance need to exchange the values output by their assigned sensors and output some common value, say print a single location of the aircraft on the pilot control screen, despite the fact that, due to the inaccuracy or failure of their local sensors, they may have observed slightly different input values. This cooperation should be achieved despite some sensors (or associated control processes) having crashed or not observed anything. This type of cooperation can be simplified if all processes agree on the same set of inputs for the control algorithm, a requirement captured by the *consensus* abstraction.

Cooperative Work. Users located on different nodes of a network might cooperate in building a common software or document, or simply in setting-up a distributed dialogue, say for a virtual conference. A shared working space abstraction is very useful here to enable effective cooperation. Such distributed shared memory abstraction is typically accessed through *read* and *write* operations that the users exploit to store and exchange information. In its simplest form, a shared working space can be viewed as a virtual register or a distributed file system. To maintain a consistent view of the shared space, the processes need to agree on the relative order among *write* and *read* operations on that shared board.

Distributed Databases. These constitute another class of applications where agreement abstractions can be helpful to ensure that all transaction managers obtain a consistent view of the running transactions and can make consistent decisions on the way these transactions are serialized.

Additionally, such abstractions can be used to coordinate the transaction managers when deciding about the outcome of the transactions. That is, the database servers on which a given distributed transaction has executed would need to coordinate their activities and decide on whether to commit or abort the transaction. They might decide to abort the transaction if any database server detected a violation of the database integrity, a concurrency control inconsistency, a disk error, or simply the crash of some other database server. An distributed programming abstraction that is useful here is the *atomic commit* (or commitment) form of distributed cooperation.

1.2.2 Distribution as an Artifact

In general, even if the application is not inherently distributed and might not, at first glance, need sophisticated distributed programming abstractions, this need sometimes appears as an artifact of the engineering solution to satisfy some specific requirements such as *fault-tolerance*, *load-balancing*, or *fast-sharing*.

We illustrate this idea through replication, which is a powerful way to achieve fault-tolerance in distributed systems. Briefly, replication consists in making a centralized service highly-available by executing several copies of it on several machines that are presumably supposed to fail independently. The service continuity is in a sense ensured despite the crash of a subset of the machines. No specific hardware is needed: fault-tolerance through replication is software-based. In fact, replication might also be used within an information system to improve the read-access performance to data by placing it close to the processes where it is supposed to be queried.

For replication to be effective, the different copies must be maintained in a consistent state. If the state of the replicas diverge arbitrarily, it does not make sense to talk about replication anyway. The illusion of *one* highly-available service would brake and replaced by that of several distributed services, each possibly failing independently. If replicas are deterministic, one of the simplest manners to guarantee full consistency is to ensure that all replicas receive the same set of requests in the same order. Typically, such guarantees are enforced by an abstraction called *total order broadcast* and discussed earlier: the processes need to agree here on the sequence of messages they deliver. Algorithms that implement such a primitive are non-trivial, and providing the programmer with an abstraction that encapsulates these algorithms makes the design of replicated components easier. If replicas are non-deterministic, then ensuring their consistency requires different *ordering* abstractions, as we will see later in the manuscript.

After a failure, it is desirable to replace the failed replica by a new component. Again, this calls for systems with *dynamic group membership* abstraction and for additional auxiliary abstractions, such as a *state-transfer* mechanism that simplifies the task of bringing the new replica up-to-date.

1.3 The End-to-end Argument

Distributed Programming abstractions are useful but may sometimes be difficult or expensive to implement. In some cases, no simple algorithm is able to provide the desired abstraction and the algorithm that solves the problem can have a high complexity, e.g., in terms of the number of inter-process communication steps and messages. Therefore, depending on the system model, the network characteristics, and the required quality of service, the overhead of the abstraction can range from the negligible to the almost impairing.

Faced with performance constraints, the application designer may be driven to mix the relevant logic of the abstraction with the application logic, in an attempt to obtain an optimized integrated solution. The intuition is that such a solution would perform better than a modular approach, where the abstraction is implemented as independent services that can be accessed through well defined interfaces. The approach can be further supported by a superficial interpretation of the end-to-end argument: most complexity should be implemented at the higher levels of the communication stack. This argument could be applied to any distributed programming.

However, even if, in some cases, performance gains can be obtained by collapsing the application and the underlying layers, such an approach has many disadvantages. First, it is very error prone. Some of the algorithms that will be presented in this manuscript have a considerable amount of difficulty and exhibit subtle dependencies among their internal components. An apparently obvious "optimization" may break the algorithm correctness. It is usual to quote Knuth here:

"Premature optimization is the source of all evil"

Even if the designer reaches the amount of expertise required to master the difficult task of embedding these algorithms in the application, there are several other reasons to keep both implementations independent. The most important of these reasons is that there is usually no single solution to solve a given distributed computing problem. This is particularly true because the variety of distributed system models. Instead, different solutions can usually be proposed and none of these solutions might strictly be superior to the others: each might have its own advantages and disadvantages, performing better under different network or load conditions, making different trade-offs between network traffic and message latency, etc. To rely on a modular approach allows the most suitable implementation to be selected when the application is deployed, or even commute in run-time among different implementations in response to changes in the operational envelope of the application.

Encapsulating tricky issues of distributed interactions within abstractions with well defined interfaces significantly helps reason about the correctness of the application and port it from one system to the other. We strongly believe that, in many distributed applications, especially those that require many-to-many interaction, building preliminary prototypes of the distributed application using several abstraction layers can be very helpful.

Ultimately, one might indeed consider optimizing the performance of the final release of a distributed application and using some integrated prototype that implements several abstractions in one monolithic peace of code. However, full understanding of each of the inclosed abstractions in isolation is fundamental to ensure the correctness of the combined code.

1.4 Software Components

1.4.1 Composition Model

Notation. One of the biggest difficulties we had to face when thinking about describing distributed algorithms was to find out an adequate way to represent these algorithms. When representing a centralized algorithm, one could decide to use a programming language, either by choosing an existing popular one, or by inventing a new one with pedagogical purposes in mind.

Although there have indeed been several attempts to come up with distributed programming languages, these attempts have resulted in rather complicated notations that would not have been viable to describe general purpose distributed algorithms in a pedagogical way. Trying to invent a distributed programming language was not an option. Had we had the time to invent one and had we even been successful, at least one book would have been required to present the language.

Therefore, we have opted to use pseudo-code to describe our algorithms. The pseudo-code assumes a reactive computing model where components of the same process communicate by exchanging events: an algorithm is described as a set of event handlers, that react to incoming events and may trigger new events. In fact, the pseudo-code is very close to the actual way we programmed the algorithms in our experimental framework. Basically, the algorithm description can be seen as actual code, from which we removed all implementation-related details that were more confusing than useful for understanding the algorithms. This approach will hopefully simplify the task of those that will be interested in building running prototypes from the descriptions found in the book.

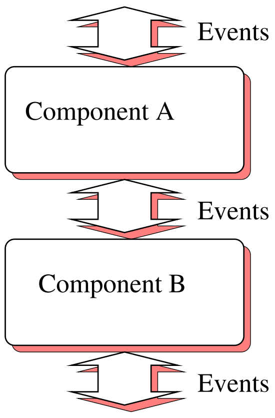

Figure 1.1. Composition model

A Simple Example. Abstractions are typically represented through application programming interfaces (API). We will informally discuss here a simple example API for a distributed programming abstraction.

To describe this API and our APIs in general, as well as the algorithms implementing these APIs, we shall consider, throughout the manuscript, an *asynchronous event-based composition* model. Every process hosts a set of software modules, called *components*. Each component is identified by a name, characterized by a set of properties, and provides an interface in the form of the events that the component accepts and produces in return. Distributed Programming abstractions are typically made of a collection of components, at least one on every process, that are supposed to satisfy some common properties.

Software Stacks. Components can be composed to build software stacks, at each process: each component represents a specific layer in the stack. The application layer is on the top of the stack whereas the networking layer is at the bottom. The layers of the distributed programming abstractions we will consider are in the middle. Components within the same stack communicate through the exchange of *events*, as illustrated in Figure 1.1. A given abstraction is typically materialized by a set of components, each running at a process.

According to this model, each component is constructed as a state-machine whose transitions are triggered by the reception of events. Events may carry information such as a data message, a group view, etc, in one or more *attributes*. Events are denoted by $\langle$ *EventType*, att1, att2, ... $\rangle$.

Each event is processed through a dedicated handler by the process (i.e., the corresponding component). The processing of an event may result in new events being created and triggered on the same or on other components. Every event triggered on a component of the same process is eventually processed, unless the process crashes. Events from the same component are processed in the same order they were triggered. Note that this FIFO (*first-in-first-out*) order is only enforced on events exchanged among local components in a given stack. The messages among different processes may also need to be ordered according to some criteria, using mechanisms orthogonal to this one. We shall address this inter-process communication issue later in the book.

We assume that every process executes the code triggered by events in a mutually exclusive way. Basically, the same process does not handle two events concurrently. Once the handling of an event is terminated, the process keeps on checking if any other event is triggered.

The code of each component looks like this:

```
upon event ⟨ Event1, $att_1^1$, $att_1^2$, ... ⟩ do
    something
    // send some event
    trigger ⟨ Event2, $att_2^1$,$att_2^2$, ... ⟩;

upon event ⟨ Event3, $att_3^1$, $att_3^2$, ... ⟩ do
    something else
    // send some other event
    trigger ⟨ Event4, $att_4^1$, $att_4^2$, ... ⟩;
```

This decoupled and asynchronous way of interacting among components matches very well the requirements of distributed applications: for instance, new processes may join or leave the system at any moment and a process must be ready to handle both membership changes and reception of messages at any time. Hence, a process should be able to concurrently handle several events, and this is precisely what we capture through our component model.

1.4.2 Programming Interface

A typical interface includes the following types of events:

- *Request* events are used by a component to request a service from another component: for instance, the application layer might trigger a *request* event at a component in charge of broadcasting a message to a set of processes in a group with some reliability guarantee, or proposing a value to be decided on by the group.
- *Confirmation* events are used by a component to confirm the completion of a request. Typically, the component in charge of implementing a broadcast will confirm to the application layer that the message was indeed broadcast or that the value suggested has indeed been proposed to the group: the component uses here a *confirmation* event.

- *Indication* events are used by a given component to *deliver* information to another component. Considering the broadcast example above, at every process that is a destination of the message, the component in charge of implementing the actual broadcast primitive will typically perform some processing to ensure the corresponding reliability guarantee, and then use an *indication* event to deliver the message to the application layer. Similarly, the decision on a value will be indicated with such an event.

A typical execution at a given layer consists of the following sequence of actions. We consider here the case of a broadcast kind of abstraction, e.g., the processes need to agree on whether or not to deliver a message broadcast by some process.

1. The execution is initiated by the reception of a *request* event from the layer above.
2. To ensure the properties of the broadcast abstraction, the layer will send one or more messages to its remote peers using the services of the layer below (using request events).
3. Messages sent by the peer layers are also *received* using the services of the underlying layer (through indication events).
4. When a message is received, it may have to be stored temporarily until the adequate reliability property is satisfied, before being *delivered* to the layer above (using a indication event).

This data-flow is illustrated in Figure 1.2. Events used to deliver information to the layer above are *indications*. In some cases, the layer may confirm that a service has been concluded using a *confirmation* event.

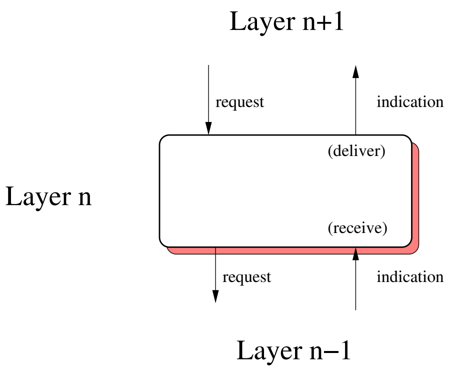

Figure 1.2. Layering

Module:

Name: Print (lpr).

Events:

Request: ⟨ *lprPrint*, rqid, string ⟩: Requests a string to be printed. The token rqid is an identifier of the request.

Confirmation:⟨ *lprOk*, rqid ⟩: Used to confirm that the printing request with identifier rqid succeeded.

Module 1.1 Interface of a printing module.

Algorithm 1.1 Printing service.

```
Implements:
    Print (lpr).

upon event ⟨ lprPrint, rqid, string ⟩ do
    print string;
    trigger ⟨ lprOk, rqid ⟩;
```

1.4.3 Modules

Not surprisingly, the modules described in this manuscript perform some interaction with the correspondent modules on peer processes: after all, this is a manuscript about distributed computing. It is however also possible to have modules that perform only local actions.

To illustrate the notion of modules, we use the example of a simple printing module. This module receives a print request, issues a print command and provides a confirmation of the print operation having been achieved. Module 1.1 describes its interface and Algorithm 1.1 its implementation. The algorithm is supposed to be executed by every process p_i.

To illustrate the way modules are composed, we use the printing module above to build a *bounded* printing service. The bounded printer only accepts a limited, pre-defined, number of printing requests. The bounded printer also generates an indication when the threshold of allowed print requests is reached. The bounded printer uses the service of the (unbounded) printer introduced above and maintains a counter to keep track of the number of printing requests executed in the past. Module 1.2 provides the interface of the bounded printer and Algorithm 1.2 its implementation.

To simplify the presentation of the components, we assume that a special ⟨ *Init* ⟩ event is generated automatically by the run-time system when a component is created. This event is used to perform the initialization of the component. For instance, in the bounded printer example, this event is used to initialize the counter of executed printing requests.

Module:

Name: BoundedPrint (blpr).

Events:

Request: ⟨ *blprPrint*, rqid, string ⟩: Request a string to be printed. The token rqid is an identifier of the request.

Confirmation:⟨ *blprStatus*, rqid, status ⟩: Used to return the outcome of the printing request: Ok or Nok.

Indication:⟨ *blprAlarm* ⟩: Used to indicate that the threshold was reached.

Module 1.2 Interface of a bounded printing module.

Algorithm 1.2 Bounded printer based on (unbounded) print service.

```
Implements:
    BoundedPrint (blpr).

Uses:
    Print (lpr).

upon event ⟨ Init ⟩ do
    bound := PredefinedThreshold;

upon event ⟨ blprPrint, rqid, string ⟩ do
    if bound > 0 then
        bound := bound-1;
        trigger ⟨ lprPrint, rqid, string ⟩;
        if bound = 0 then trigger ⟨ blprAlarm ⟩;
    else
        trigger ⟨ blprStatus, rqid, Nok ⟩;

upon event ⟨ lprOk, rqid ⟩ do
    trigger ⟨ blprStatus, rqid, Ok ⟩;
```

1.4.4 Classes of Algorithms

As noted above, in order to provide a given service, a layer at a given process may need to execute one or more rounds of message exchange with the peer layers at remote processes. The behavior of each peer, characterized by the set of messages that it is capable of producing and accepting, the format of each of these messages, and the legal sequences of messages, is sometimes called a *protocol*. The purpose of the protocol is to ensure the execution of some *distributed algorithm*, the concurrent execution of different sequences of steps that ensure the provision of the desired service. This manuscript covers several of these distributed algorithms.

To give the reader an insight of how design solutions and system-related parameters affect the algorithm design, the book includes five different classes of algorithmic solutions to implement our distributed programming abstractions, namely: *fail-stop* algorithms, designed under the assumption that processes can fail by crashing but the crashes can be reliably detected by all the other processes; *fail-silent* algorithms where process crashes can never be reliably detected; *fail-noisy* algorithms, where processes can fail by crashing and the crashes can be detected, but not always in a reliable manner; *fail-recovery* algorithms, where processes can crash and later recover and still participate in the algorithm; *randomized* algorithms, where processes use randomization to ensure the properties of the abstraction with some known probability.

These classes are not disjoint and it is important to notice that we do not give a solution from each class to every abstraction. First, there are cases where it is known that some abstraction cannot be implemented from an algorithm of a given class. For example, the coordination abstractions we consider in Chapter 7 do not have fail-noisy (and hence fail-silent) solutions and it is not clear either how to devise meaningful randomized solutions to such abstractions. In other cases, such solutions might exist but devising them is still an active area of research.

Reasoning about distributed algorithms in general, and in particular about algorithms that implement distributed programming abstractions, first goes through defining a clear model of the distributed system where these algorithms are supposed to operate. Put differently, we need to figure out what basic abstractions the processes assume in order to build more sophisticated ones. The basic abstractions we consider capture the allowable behavior of the processes and their communication links in the distributed system. Before delving into concrete algorithms to build sophisticated distributed programming abstractions, we thus need to understand such basic abstractions. This will be the topic of the next chapter.

Hands-On

We have implemented several of the algorithms that we will be presenting in the book. By using these implementations, the reader has the opportunity to run and experiment the algorithms in a real setting, look at the code, make changes and improvements to the given code and, eventually, take it as a basis to implement her own algorithms.

The algorithms have been implemented in the java programming language with the support of the *Appia* protocol composition and execution framework (Miranda, Pinto, and Rodrigues 2001). *Appia* is a tool that simplifies the development of communication protocols. To start with, *Appia* already implements a number of basic services that are required in several protocols, such as methods to add and extract headers from messages or launch timers. Additionally, *Appia* simplifies the task of composing different protocol modules.

Central to the use of *Appia* is the notion of *protocol composition*. In its simpler form, a protocol composition is a stack of instances of the Layer class. For each different protocol module, a different specialization of the Layer class should be defined. In *Appia*, modules communicate through the exchange of *events*. *Appia* defines the class Event, from which all events exchanged in the *Appia* framework must be derived. In order for a module to consume and produce events, a layer must explicitly declare the set of events *accepted*, *provided*, and *required*. When a layer requires an event, *Appia* checks if there is another layer in the composition that provides that event, otherwise it generates an exception. This offers a simple form of detecting inconsistencies in the protocol composition.

Print Module

Consider for instance the implementation of the Print module (Module 1.1). In first place, we define the events accepted and provided by this module. This is illustrated in Listing 1.1.

Listing 1.1. Events for the Print module.

```
class PrintRequestEvent extends Event {
    int      r_id;
    String   data;

    void setId (int rid);
    void setStrint (String s);
    int getId ();
    String getString ();
}

class PrintConfirmEvent extends Event {
    int      r_id;

    void setId (int rid);
    int getId ();
}
```

Then, we implement the layer for this module. This is illustrated in Listing 1.2. As expected, the layer accepts the PrintRequestEvent and provides the PrintConfirmEvent. The PrintLayer is also responsible for creating objects of class PrintSession, whose purpose is described in the next paragraphs.

Listing 1.2. PrintLayer.

```
public class PrintLayer extends Layer {

  public PrintLayer(){
    /* events that the protocol will create */
    evProvide = new Class[1];
    evProvide[0] = PrintConfirmEvent.class;

    /* events that the protocol requires to work.  This is
     * a subset of the accepted events */
    evRequire = new Class[0];

    /* events that the protocol will accept */
    evAccept = new Class[2];
    evAccept[0] = PrintRequestEvent.class;
    evAccept[1] = ChannelInit.class;
  }

  public Session createSession() {
    return new PrintSession(this);
  }

}
```

Layers are used to describe the behavior of each module. The actual methods and the state required by the algorithm is maintained by *Session* objects. Thus, for every layer, the programmer needs to define the corresponding session. The main method of a session is the handle method, that is invoked by the *Appia* kernel whenever there is an event to be processed by the session. For the case of our Print module, the implementation of the PrintSession is given in Listing 1.3.

Listing 1.3. PrintSession.

```
public class PrintSession extends Session {

    public PrintSession(Layer layer) {
        super(layer);
    }

    public void handle(Event event){
        if(event instanceof ChannelInit)
            handleChannelInit((ChannelInit)event);
        else if(event instanceof PrintRequestEvent){
            handlePrintRequest ((PrintRequestEvent)event);
        }
    }

    private void handleChannelInit(ChannelInit init) {
        try {
            init.go();
        } catch (AppiaEventException e) {
            e.printStackTrace();
        }
    }

    private void handlePrintRequest(PrintRequestEvent request) {
        try {
            PrintConfirmEvent ack = new PrintConfirmEvent ();

            doPrint (request.getString ());
            request.go ();

            ack.setChannel(request.getChannel());
            ack.setDir(Direction.UP);
            ack.setSource(this);
            ack.setId(request.getId ());
            ack. init ();
            ack.go();
        } catch (AppiaEventException e) {
            e.printStackTrace();
        }
    }
}
```

There are a couple of issues the reader should note in the previous code. First, as in most of our algorithms, every session should be ready to accept the ChannelInit event. This event is automatically generated and should be used to initialize the session state. Second, in *Appia*, the default behaviour for a session is to always forward downwards (or upwards) in the stack the events it consumes . As it will become clear later in the text, it is often very convenient to have the same event be processed by different modules in sequence.

BoundedPrint Module

Having defined the events, the layer, and the session for the Print module, we can now perform a similar job for the BoundedPrint module (Module 1.2). As before we start by providing the required events, as depicted in Listing 1.4. Note that we define the PrintAlarmEvent and the PrintStatusEvent11. On the other hand, we do not need to define a new event for the PrintRequestEvent, as we can reuse the event used in the basic Print module.

Listing 1.4. Events for BoundedPrint module.

```
class PrintAlarmEvent extends Event {
}

class PrintStatusEvent extends Event {
    int      r_id;
    Status   stat;

    void setId (int rid);
    void setStatus (Status s);
    int getId ();
    int getStatus ();
}
```

We proceed to define the BoundedPrintLayer, as depicted in Listing 1.5. Since the BoundedPrint module uses the services of the basic Print module, it requires the PrintConfirmEvent produced by that module.

Listing 1.5. Bounded PrintLayer.

```
public class BoundedPrintLayer extends Layer {

  public BoundedPrintLayer(){
    /* events that the protocol will create */
    evProvide = new Class[2];
    evProvide[0] = PrintStatusEvent.class;
    evProvide[1] = PrintAlarmEvent.class;

    /* events that the protocol require to work.
     * This is a subset of the accepted events */
    evRequire = new Class[1];
    evRequire[0] = PrintConfirmEvent.class;

    /* events that the protocol will accept */
    evAccept = new Class[3];
    evAccept[0] = PrintRequestEvent.class;
    evAccept[1] = PrintConfirmEvent.class;
    evAccept[2] = ChannelInit.class;
  }

  public Session createSession() {
    return new BoundedPrintSession(this);
  }

}
```

Subsequently, we can implement the session for the BoundedPrint module, depicted in Listing 1.6.

Listing 1.6. BoundedPrintSession.

```
public class BoundedPrintSession extends Session {
    int bound;

    public BoundedPrintSession(Layer layer) {
        super(layer);
    }

    public void handle(Event event){
        if(event instanceof ChannelInit) {
            handleChannelInit((ChannelInit)event);
        }
        else if(event instanceof PrintRequestEvent) {
            handlePrintRequest ((PrintRequestEvent)event);
        }
        else if(event instanceof PrintConfirmEvent) {
            handlePrintConfirm ((PrintConfirmEvent)event);
        }
    }

    private void handleChannelInit(ChannelInit init) {
        try {
            bound = PredefinedThreshold;

            init.go();
        } catch (AppiaEventException e) {
            e.printStackTrace();
        }
    }

    private void handlePrintRequest(PrintRequestEvent request) {
        if (bound > 0){
            bound = bound -1;
            try {
                request.go ();
            } catch (AppiaEventException e) {
                e.printStackTrace();
            }
            if (bound == 0) {
                PrintAlarmEvent alarm = new PrintAlarmEvent ();
                alarm.setChannel (request.getChannel());
                alarm.setSource (this);
                alarm.setDir(Direction.UP);
                try {
        alarm.init ();
                    alarm.go ();
                } catch (AppiaEventException e) {
                    e.printStackTrace();
                }
            }
        }
        else {
            PrintStatusEvent status = new PrintStatusEvent ();
            status.setChannel (request.getChannel());
            status.setSource (this);
            status.setDir(Direction.UP);
            status.setId (request.getId ());
            status.setStatus (Status.NOK);
            try {
    status.init ();
                status.go ();
            } catch (AppiaEventException e) {
                e.printStackTrace();
            }
        }
    }

    private void handlePrintConfirm(PrintConfirmEvent conf) {
        PrintStatusEvent status = new PrintStatusEvent ();
        status.setChannel (request.getChannel());
        status.setSource (this);
        status.setDir(Direction.UP);
        status.setId (conf.getId ());
        status.setStatus (Status.OK);
        try {
      status.init ();
            status.go ();
        } catch (AppiaEventException e) {
            e.printStackTrace();
        }
    }
}
```

Composing Modules

The two modules that we have described can then be easily composed using the *Appia* framework. The first step consists in creating a protocol composition by stacking the BoundedPrintLayer on top of a the PrintLayer. Actually, in order to be able to experiment with these two layers, we further add on top of the stack a simple application layer. A protocol composition in *Appia* is called a QoS (Quality of Service) and can simply be created by providing the desired array of layers, as shown in Listing 1.7. After defining a protocol composition, it is possible to create one or more communication *channels* that

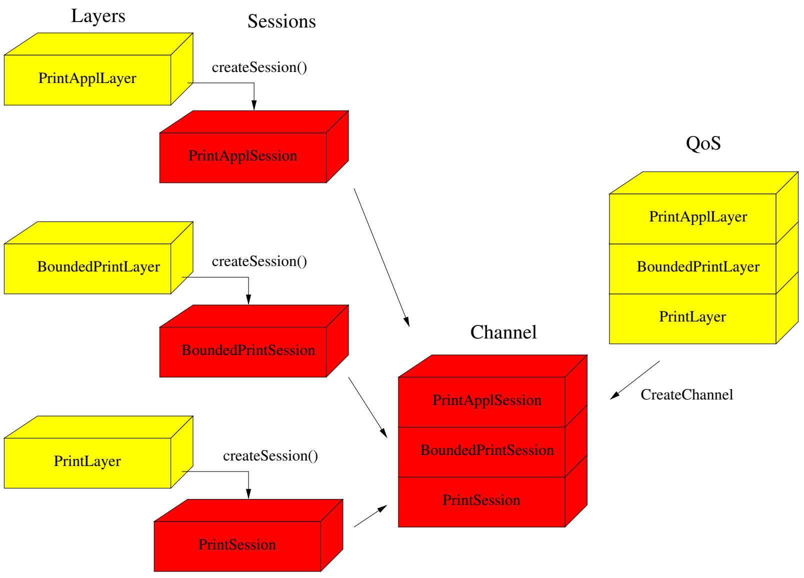

Figure 1.3. Layers, Sessions, QoS and Channels

use that composition. Therefore, channels can be seen as instances of protocol compositions. Channels are made of *sessions*. When a channel is created from a composition, it is possible to automatically create a new session for every layer in the composition. The relation between Layers, Sessions, QoS and Channles is illustrated in Figure 1.3. The code required to perform create a channel is also depicted in Listing 1.7.

Listing 1.7. Creating a PrintChannel.

```
public class Example {

        public static void main(String[] args) {
            /* Create layers and put them on a array */
            Layer[] qos =
                {new PrintLayer(),
                 new BoundedPrintLayer(),
                 new PrintApplicationLayer()};

            /* Create a QoS */
            QoS myQoS = null;
            try {
                myQoS = new QoS("Print_stack", qos);
            } catch (AppiaInvalidQoSException ex) {
                System.err.println("Invalid_QoS");
                System.err.println(ex.getMessage());
                System.exit(1);
            }

            /* Create a channel. Uses default event scheduler. */
            Channel channel = myQoS.createUnboundChannel("Print_Channel");

            try {
```

```
            channel.start ();
        } catch(AppiaDuplicatedSessionsException ex) {
            System.err.println("Error in start");
            System.exit(1);
        }

        /* All set. Appia main class will handle the rest */
        System.out.println("Starting Appia...");
        Appia.run();
    }
}
```

The reader is now invited to install the *Appia* distribution provided as a companion of this book and try the implementations described above.

2. Basic Abstractions

These are my principles. If you don't like them, I have others.
(Groucho Marx)

Applications that are deployed in practical distributed systems are usually composed of a myriad of different machines and communication infrastructures. Physical machines differ on the number of processors, type of processors, amount and speed of both volatile and persistent memory, etc. Communication infrastructures differ on parameters such as latency, throughput, reliability, etc. On top of these machines and infrastructures, a huge variety of software components are sometimes encompassed by the same application: operating systems, file-systems, middleware, communication protocols, each component with its own specific features.

One might consider implementing distributed services that are tailored to specific combinations of the elements listed above. Such implementation would depend on one type of machine, one form of communication, one distributed operating system, etc. However, in this book we are interested in studying abstractions and algorithms that are relevant for a wide range of distributed environments. In order to achieve this goal we need to capture the fundamental characteristics of various distributed systems in some basic abstractions, and on top of which we can later define other more elaborate, and generic, distributed programming abstractions.

This chapter presents the basic abstractions we use to model a distributed system composed of computational entities (*processes*) communicating by exchanging messages. Two kinds of abstractions will be of primary importance: those representing *processes* and those representing communication *links.* Not surprisingly, it does not seem to be possible to model the huge diversity of physical networks and operational conditions with a single process abstraction and a single link abstraction. Therefore, we will define different instances for each kind of basic abstraction: for instance, we will distinguish process abstractions according to the types of faults that they may exhibit. Besides our process and link abstractions, we will introduce the *failure detector* abstraction as a convenient way to capture assumptions that might be reasonable to make on the timing behavior of processes and links. Later in the chapter we will identify relevant combinations of our three categories of abstractions. Such a combination is what we call a *distributed system model.*

This chapter also contains our first module descriptions, used to specify our basic abstractions, as well as our first algorithms, used to implement these abstractions. The specifications and the algorithms are rather simple and should help illustrate our notation, before proceeding in subsequent chapters to more sophisticated specifications and algorithms.

2.1 Distributed Computation

2.1.1 Processes and Messages

We abstract the units that are able to perform computations in a distributed system through the notion of *process.* We consider that the system is composed of N uniquely identified processes, denoted by $p_1, p_2, \ldots, p_N$. Sometimes we also denote the processes by p, q, r. The set of system processes is denoted by Π. Unless explicitly stated otherwise, it is assumed that this set is static (does not change) and processes do know of each other.

We do not assume any particular mapping of our abstract notion of process to the actual processors, processes, or threads of a specific machine or operating system. The processes communicate by exchanging messages and the messages are uniquely identified, say by their original sender process using a sequence number or a local clock, together with the process identifier. Messages are exchanged by the processes through communication *links.* We will capture the properties of the links that connect the processes through specific link abstractions, and which we will discuss later.

2.1.2 Automata and Steps

A *distributed algorithm* is viewed as a collection of distributed automata, one per process. The automaton at a process regulates the way the process executes its computation steps, i.e., how it reacts to a message. The *execution* of a distributed algorithm is represented by a sequence of steps executed by the processes. The elements of the sequences are the steps executed by the processes involved in the algorithm. A partial execution of the algorithm is represented by a finite sequence of steps and an infinite execution by an infinite sequence.

It is convenient for presentation simplicity to assume the existence of a global clock, outside the control of the processes. This clock provides a global and linear notion of time that regulates the execution of the algorithms. The steps of the processes are executed according to ticks of the global clock:

one step per clock tick. Even if two steps are executed at the same physical instant, we view them as if they were executed at two different times of our global clock. A *correct* process executes an infinite number of steps, i.e., every process has an infinite share of time units (we come back to this notion in the next section). In a sense, there is some entity (a global scheduler) that schedules time units among processes, though the very notion of time is outside the control of the processes.

A process step consists in *receiving* (sometimes we will be saying *delivering*) a message from another process (global event), *executing* a local computation (local event), and *sending* a message to some process (global event) (Figure 2.1). The execution of the local computation and the sending of a message is determined by the process automaton, i.e., the algorithm. Local events that are generated are typically those exchanged between modules of the same process at different layers.

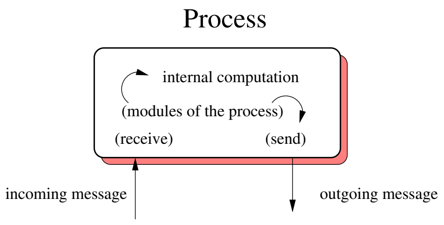

Figure 2.1. Step of a process

The fact that a process has no message to receive or send, but has some local computation to perform, is simply captured by assuming that messages might be *nil*, i.e., the process receives/sends the *nil* message. Of course, a process might not have any local computation to perform either, in which case it does simply not touch any of its local variables. In this case, the local computation is also nil.

It is important to notice that the interaction between local components of the very same process is viewed as a local computation and not as a communication. We will not be talking about messages exchanged between modules of the same process. The process is the unit of communication, just like it is the unit of failures as we will discuss shortly below. In short, a *communication step* of the algorithm occurs when a process sends a message to another process, and the latter receives this message. The number of communication steps reflects the latency an implementation exhibits, since the network latency is typically a limiting factor of the performance of distributed algorithms. An important parameter of the process abstraction is the restriction imposed on the speed at which local steps are performed and messages are exchanged.

Unless specified otherwise, we will consider *deterministic* algorithms. That is, for every step performed by any given process, the local computation executed by the process and the message sent by this process are uniquely determined by the message received by the process and its local state prior to executing the step. We will also, in specific situations, describe *randomized* (or *probabilistic*) algorithms where processes make use of underlying *random* oracles to choose the local computation to be performed or the next message to be sent, among a set of possibilities.

2.1.3 Liveness and Safety

When we devise a distributed algorithm to implement a given distributed programming abstraction, we seek to satisfy the properties of the abstraction in all possible executions of the algorithm, i.e., in all possible sequences of steps executed by the processes according to the algorithm. These properties usually fall into two classes: *safety* and *liveness*. Having in mind the distinction between these classes usually helps understand the abstraction and hence devise an adequate algorithm to implement it.

- Basically, a safety property is a property of a distributed algorithm that can be violated at some time t and never be satisfied again after that time. Roughly speaking, safety properties state that the algorithm should not do anything wrong. To illustrate this, consider a property of perfect links (which we will discuss in more details later in this chapter) that roughly stipulates that no process should receive a message unless this message was indeed sent. In other words, processes should not invent messages out of thin air. To state that this property is violated in some execution of an algorithm, we need to determine a time t at which some process receives a message that was never sent.
  More precisely, a safety property is a property that whenever it is violated in some execution E of an algorithm, there is a partial execution E' of E such that the property will be violated in any extension of E'. In more sophisticated terms, we would say that safety properties are closed under execution prefixes.
  Of course, safety properties are not enough. Sometimes, a good way of preventing bad things from happening consists in simply doing nothing. In our countries of origin, some public administrations seem to understand this rule quite well and hence have an easy time ensuring safety.
- Therefore, to define a useful abstraction, it is necessary to add some liveness properties to ensure that eventually something good happens. For instance, to define a meaningful notion of perfect links, we would require that if a correct process sends a message to a correct destination process, then the destination process should eventually deliver the message. To state that such a property is violated in a given execution, we need to show that there is no chance for a message to be delivered.

More precisely, a liveness property is a property of a distributed system execution such that, for any time t, there is some hope that the property can be satisfied at some time $t' \geq t$. It is a property for which, quoting Cicero:

"While there is life there is hope".

In general, the challenge is to guarantee both liveness and safety. (The difficulty is not in *talking*, or *not lying*, but in *telling the truth*). Indeed, useful distributed services are supposed to provide both liveness and safety properties. Consider for instance a traditional inter-process communication service such as TCP: it ensures that messages exchanged between two processes are not lost or duplicated, and are received in the order they were sent. As we pointed out, the very fact that the messages are not lost is a liveness property. The very fact that the messages are not duplicated and received in the order they were sent are rather safety properties. Sometimes, we will consider properties that are neither pure liveness nor pure safety properties, but rather a union of both.

2.2 Abstracting Processes

2.2.1 Process Failures

Unless it *fails*, a process is supposed to execute the algorithm assigned to it, through the set of components implementing the algorithm within that process. Our unit of failure is the process. When the process fails, all its components are supposed to fail as well, and at the same time.

Process abstractions differ according to the nature of the failures that are considered. We discuss various forms of failures in the next section (Figure 2.2).

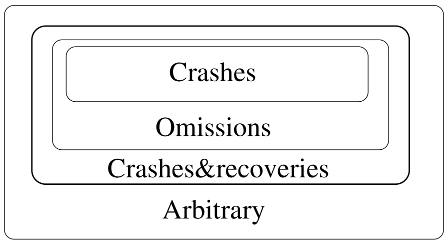

Figure 2.2. Failure modes of a process

2.2.2 Lies and Omissions

A process is said to fail in an *arbitrary* manner if it deviates arbitrarily from the algorithm assigned to it. The *arbitrary fault* behavior is the most general one. In fact, it makes no assumptions on the behavior of faulty processes, which are allowed any kind of output and in particular can send any kind of messages. These kinds of failures are sometimes called *Byzantine* (see the historical note at the end of this chapter) or *malicious* failures. Not surprisingly, arbitrary faults are the most expensive to tolerate, but this is the only acceptable option when an extremely high coverage is required or when there is the risk of some processes being indeed controlled by malicious users that deliberately try to prevent correct system operation.

An arbitrary fault need not be intentional and malicious: it can simply be caused by a bug in the implementation, the programming language or the compiler, that causes the process to deviate from the algorithm it was supposed to execute. A more restricted kind of faults to consider is the *omission* (Figure 2.2). An omission fault occurs when a process does not send (resp. receive) a message it is supposed to send (resp. receive), according to its algorithm.

In general, omission faults are due to buffer overflows or network congestion. Omission faults result in lost messages. With an omission, the process deviates from the algorithm it is supposed to execute by dropping some messages that should have been exchanged with other processes.

2.2.3 Crashes

An interesting particular case of omissions is when a process executes its algorithm correctly, including the exchange of messages with other processes, possibly until some time t, after which the process does not send any message to any other process. This is what happens if the process for instance crashes at time t and never recovers after that time. If, besides not sending any message after some time t, the process also stops executing any local computation after t, we talk here about a *crash failure* (Figure 2.2), and a *crash stop* process abstraction. The process is said to *crash* at time t. With this abstraction, a process is said to be *faulty* if it crashes. It is said to be *correct* if it does never crash and executes an infinite number of steps. We discuss in the following two ramifications underlying the crash-stop abstraction.

- It is usual to devise algorithms that implement a given distributed programming abstraction, say some form of agreement, provided that a minimal number F of processes are correct, e.g., at least one, or a majority. It is important to understand here that such assumption does not mean that the hardware underlying these processes is supposed to operate correctly forever. In fact, the assumption means that, in every execution of the algorithm making use of that abstraction, it is very unlikely that more than a certain number F of processes crash, during the lifetime of that very execution. An engineer picking such algorithm for a given application should be confident enough that the chosen elements underlying the software and hardware architecture make that assumption plausible. In general, it is also a good practice, when devising algorithms that implement a given distributed abstraction under certain assumptions to determine precisely which properties of the abstraction are preserved and which can be violated when a specific subset of the assumptions are not satisfied, e.g., when more than F processes crash.

- Considering a crash-stop process abstraction boils down to assuming that a process executes its algorithm correctly, unless it crashes, in which case it does not recover. That is, once it crashes, the process does never perform any computation. Obviously, in practice, processes that crash can in general be rebooted and hence do usually recover. It is important to notice that, in practice, the crash-stop process abstraction does not preclude the possibility of recovery, nor does it mean that recovery should be prevented for a given algorithm (assuming a crash-stop process abstraction) to behave correctly. It simply means that the algorithm should not rely on some of the processes to recover in order to pursue its execution. These processes might not recover, or might recover only after a long period encompassing the crash detection and then the rebooting delay. In some sense, an algorithm that is not relying on crashed processes to recover would typically be faster than an algorithm relying on some of the processes to recover (we will discuss this issue in the next section). Nothing prevents however recovered processes from getting informed about the outcome of the computation and participate in subsequent instances of the distributed algorithm.

Unless explicitly stated otherwise, we will assume the crash-stop process abstraction throughout this manuscript.

2.2.4 Recoveries

Sometimes, the assumption that certain processes never crash is simply not plausible for certain distributed environments. For instance, assuming that a majority of the processes do not crash, even only long enough for an algorithm execution to terminate, might simply be too strong.

An interesting alternative as a process abstraction to consider in this case is the *crash-recovery* one; we also talk about a *crash-recovery* kind of failure (Figure 2.2). We say that a process is faulty in this case if either the process crashes and never recovers, or the process keeps infinitely crashing and recovering. Otherwise, the process is said to be correct. Basically, such a process is eventually always (i.e., during the lifetime of the algorithm execution of interest) up and operating. A process that crashes and recovers a finite number of times is correct in this model (i.e., according to this abstraction of a process).

According to the crash-recovery abstraction, a process can indeed crash, in this case the process stops sending messages, but might later recover. This can be viewed as an omission fault, with one exception however: a process might suffer *amnesia* when it crashes and loses its internal state. This significantly complicates the design of algorithms because, upon recovery, the process might send new messages that contradict messages that the process might have sent prior to the crash. To cope with this issue, we sometimes assume that every process has, in addition to its regular volatile memory, a *stable storage* (also called a *log*), which can be accessed through *store* and *retrieve* primitives.

Upon recovery, we assume that a process is aware that it has crashed and recovered. In particular, a specific ⟨ *Recovery* ⟩ event is supposed to be automatically generated by the run-time environment in a similar manner to the ⟨ *Init* ⟩ event, executed each time a process starts executing some algorithm. The processing of the ⟨ *Recovery* ⟩ event should for instance retrieve the relevant state of the process from stable storage before the processing of other events is resumed. The process might however have lost all the remaining data that was preserved in volatile memory. This data should thus be properly re-initialized. The ⟨ *Init* ⟩ event is considered atomic with respect to recovery. More precisely, if a process crashes in the middle of its initialization procedure and recovers, say without having processed the ⟨ *Init* ⟩ event properly, the process should redo again the ⟨ *Init* ⟩ procedure before proceeding to the ⟨ *Recovery* ⟩ one.

In some sense, a crash-recovery kind of failure matches an omission one if we consider that every process stores every update to any of its variables in stable storage. This is not very practical because access to stable storage is usually expensive (as there is a significant delay in accessing it). Therefore, a crucial issue in devising algorithms with the crash-recovery abstraction is to minimize the access to stable storage.

We discuss in the following three important ramifications underlying the crash-recovery abstraction.

- One way to alleviate the need for accessing any form of stable storage is to assume that some of the processes never crash (during the lifetime of an algorithm execution). This might look contradictory with the actual motivation for introducing the crash-recovery process abstraction at the first place. In fact, there is no contradiction, as we explain below. As discussed earlier, with crash-stop failures, some distributed programming abstractions can only be implemented under the assumption that a certain number of processes do never crash, say a majority the processes participating in the computation, e.g., 4 out of 7 processes. This assumption might be considered unrealistic in certain environments. Instead, one might consider it more reasonable to assume that at least 2 processes do not crash during the execution of an algorithm. (The rest of the processes would indeed crash and recover.) As we will discuss later in the manuscript, such assumption makes it sometimes possible to devise algorithms assuming the crash-recovery process abstraction without any access to a stable storage. In fact, the processes that do not crash implement a virtual stable storage abstraction, and this is made possible without knowing in advance which of the processes will not crash in a given execution of the algorithm.

- At first glance, one might believe that the crash-stop abstraction can also capture situations where processes crash and recover, by simply having the processes change their identities upon recovery. That is, a process that recovers after a crash, would behave, with respect to the other processes, as if it was a different process that was simply not performing any action. This could easily be done assuming a re-initialization procedure where, besides initializing its state as if it just started its execution, a process would also change its identity. Of course, this process should be updated with any information it might have missed from others, as if indeed it did not receive that information yet. Unfortunately, this view is misleading as we explain below. Again, consider an algorithm devised using the crash-stop process abstraction, and assuming that a majority of the processes do never crash, say at least 4 out of a total of 7 processes composing the system. Consider furthermore a scenario where 4 processes do indeed crash, and process one recovers. Pretending that the latter process is a different one (upon recovery) would mean that the system is actually composed of 8 processes, 5 of which should not crash, and the same reasoning can be made for this larger number of processes. This is because a fundamental assumption that we build upon is that the set of processes involved in any given computation is static and the processes know of each other in advance. In Chapter 7, we will revisit that fundamental assumption and discuss how to build the abstraction of a dynamic set of processes.
- A tricky issue with the crash-recovery process abstraction is the interface between software modules. Assume that some module at a process, involved in the implementation of some specific distributed abstraction, delivers some message or decision to the upper layer (say the application) and subsequently the process hosting the module crashes. Upon recovery, the module cannot determine if the upper layer (i.e., the application) has processed the message or decision before crashing or not. There are at least two ways to deal with this issue.
  1. One way is to change the interface between modules. Instead of delivering a message (or a decision) to the upper layer, the module may instead store the message (decision) in a stable storage that is exposed to the upper layer. It is then up to the upper layer to access the stable storage and exploit delivered information.
  2. A different approach consists in having the module periodically delivering the message or decision to the application until the latter explicitely asks for stopping the delivery. That is, the distributed programming

abstraction implemented by the module is in this case responsible for making sure the application will make use of the delivered information.

2.3 Abstracting Communication

The *link* abstraction is used to represent the network components of the distributed system. We assume that every pair of processes is connected by a bidirectional link, a topology that provides full connectivity among the processes. In practice, different topologies may be used to implement this abstraction, possibly using routing algorithms. Concrete examples, such as the ones illustrated in Figure 2.3, include the use of a broadcast medium (such as an Ethernet), a ring, or a mesh of links interconnected by bridges and routers (such as the Internet). Many implementations refine the abstract network view to make use of the properties of the underlying topology.

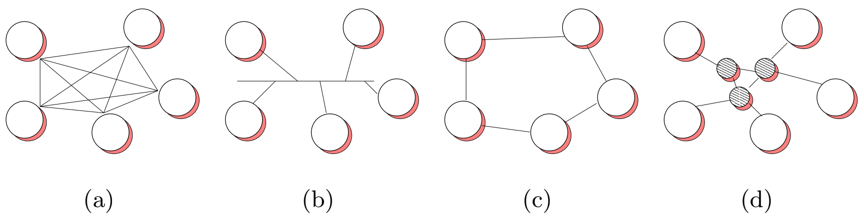

Figure 2.3. The link abstraction and different instances.

We assume that messages exchanged between processes are uniquely identified and every message includes enough information for the recipient of a message to uniquely identify its sender. Furthermore, when exchanging messages in a request-reply manner among different processes, the processes have means to identify which reply message is a response to which request message. This can typically be achieved by having the processes generating (random) timestamps, based on sequence numbers or on local clocks. This assumption alleviates the need for explicitly introducing these timestamps in the algorithm.

2.3.1 Link Failures

In a distributed system, it is common for messages to be lost when transiting through the network. However, it is reasonable to assume that the probability for a message to reach its destination is non-zero. Hence, a natural way to overcome the inherent unreliability of the network is to keep on retransmitting messages until they reach their destinations.

In the following, we will describe different kinds of link abstractions: some are stronger than others in the sense that they provide more reliability guarantees. All three are *point-to-point* link abstractions, i.e., they support the communication between pairs of processes. (In the next chapter, we will be defining broadcast communication abstractions.)

We will first describe the abstraction of *fair-loss* links, which captures the basic idea that messages might be lost but the probability for a message not to be lost is non-zero. Then we describe higher level abstractions that could be implemented over *fair-loss* links using retransmission mechanisms to hide from the programmer part of the unreliability of the network. We will more precisely consider *stubborn* and *perfect* link abstractions. As we pointed out earlier, unless explicitly stated otherwise, we will be assuming the crash-stop process abstraction.

We define the properties of each of our link abstractions using two kinds of primitives: *send* and *deliver*. The term *deliver* is privileged upon the more general term *receive* to make it clear that we are talking about a specific link abstraction to be implemented over the network: a message might typically be *received* at a given port of the network and stored within some buffer, then some algorithm will be executed to make sure the properties of the required link abstraction are satisfied, before the message is actually *delivered*. When there is no ambiguity, we might however use the term *receive* to mean *deliver*.

A process invokes the *send* primitive of a link abstraction to request the sending of a message using that abstraction. When the process invokes that primitive, we say that the process sends the message. It might then be up to the link abstraction to make some effort in transmitting the message to the destinator process, according to the actual specification of the abstraction. The *deliver* primitive is invoked by the algorithm implementing the abstraction on a destinator process. When this primitive is invoked on a process p for a message m, we say that p delivers m.

2.3.2 Fair-loss Links

The interface of the fair-loss link abstraction is described by Module 2.1, "Fair Loss Point To Point Links (flp2p)". This consists of two events: a request event, used to send messages, and an indication event, used to deliver the messages. Fair-loss links are characterized by the properties FLL1-FLL3.

Basically, the *fair loss* property guarantees that a link does not systematically drop any given message. Therefore, if neither the sender nor the recipient crashes, and if a message keeps being re-transmitted, the message is eventually delivered. The *finite duplication* property intuitively ensures that the network does not perform more retransmission than those performed by the processes themselves. Finally, the *no creation* property ensures that no message is created or corrupted by the network.

Module:

Name: FairLossPointToPointLinks (flp2p).

Events:

Request: ⟨ *flp2pSend*, dest, m ⟩: Used to request the transmission of message *m* to process *dest*.

Indication: ⟨ *flp2pDeliver*, src, m ⟩: Used to deliver message *m* sent by process *src*.

Properties:

FLL1: *Fair loss:* If a message m is sent infinitely often by process p_i to process p_j, and neither p_i nor p_j crash, then m is delivered an infinite number of times by p_j.

FLL2: *Finite duplication:* If a message m is sent a finite number of times by process p_i to process p_j, then m cannot be delivered an infinite number of times by p_j.

FLL3: *No creation:* If a message m is delivered by some process p_j, then m has been previously sent to p_j by some process p_i.

Module 2.1 Interface and properties of fair-lossy point-to-point links.

Module:

Name: StubbornPointToPointLink (sp2p).

Events:

Request: ⟨ *sp2pSend*, *dest*, *m* ⟩: Used to request the transmission of message *m* to process *dest*.

Indication:⟨ *sp2pDeliver*, *src*, *m* ⟩: Used to deliver message *m* sent by process *src*.

Properties:

SL1: *Stubborn delivery:* Let p_i be any process that sends a message m to a correct process p_j. If p_i does not crash, then p_j delivers m an infinite number of times.

SL2: *No creation:* If a message m is delivered by some process p_j, then m was previously sent to p_j by some process p_i.

Module 2.2 Interface and properties of stubborn point-to-point links.

2.3.3 Stubborn Links

We define the abstraction of *stubborn* channels in Module 2.2, “Stubborn Point To Point Link (sp2p)”. This abstraction hides lower layer retransmission mechanisms used by the sender process, when using actual fair loss links, to make sure its messages are eventually delivered by the destination processes.

Algorithm 2.1, that we have called “Retransmit Forever” describes a very simple implementation of stubborn links over fair-loss ones. As the name im-

Algorithm 2.1 Retransmit Forever.

```
Implements:
    StubbornPointToPointLink (sp2p).

Uses:
    FairLossPointToPointLinks (flp2p).

upon event ⟨ Init ⟩ do
    sent := ∅;
    startTimer (TimeDelay);

upon event ⟨ Timeout ⟩ do
    forall $(dest, m)$ ∈ sent do
        trigger ⟨ flp2pSend, dest, m ⟩;
    startTimer (TimeDelay);

upon event ⟨ sp2pSend, dest, m ⟩ do
    trigger ⟨ flp2pSend, dest, m ⟩;
    sent := sent ∪ { (dest,m) };

upon event ⟨ flp2pDeliver, src, m ⟩ do
    trigger ⟨ sp2pDeliver, src, m ⟩;
```

plies, it simply keeps on retransmiting all messages sent, to overcome eventual omissions in the links. We discuss in the following the correctness of the algorithm as well as some performance considerations.

Correctness. The *fair loss* property of the underlying links guarantees that, if the destinator process is correct, it will indeed deliver, infinitely often, every message that was sent by every process that does not subsequently crashes. This is because the algorithm makes sure the sender process will keep on sp2pSending those messages infinitely often, unless that sender process itself crashes. The *no creation* property is simply preserved from the underlying links.

Performance. The algorithm is clearly not performant and its purpose is primarily pedagogical. It is pretty clear that, within a practical application, it does not make much sense for a process to keep on, and at every step, sending messages infinitely often. There are at least three complementary ways to prevent that and hence make the algorithm more practical. First, the sender process might very well introduce a time delay between two sending events (using the fair loss links). Second, it is very important to remember that the very notion of infinity and infinitely often are context dependent: they basically depend on the algorithm making use of stubborn links. After the algorithm making use of those links has ended its execution, there is no need to keep on sending messages. Third, an acknowledgement mechanism, possibly used for groups of processes, can very well be added to mean to a

Module:

Name: PerfectPointToPointLink (pp2p).

Events:

Request: ⟨ *pp2pSend*, *dest*, *m* ⟩: Used to request the transmission of message *m* to process *dest*.

Indication:⟨ *pp2pDeliver*, *src*, *m* ⟩: Used to deliver message *m* sent by process *src*.

Properties:

PL1: *Reliable delivery:* Let p_i be any process that sends a message m to a process p_j. If neither p_i nor p_j crashes, then p_j eventually delivers m.

PL2: *No duplication:* No message is delivered by a process more than once.

PL3: *No creation:* If a message m is delivered by some process p_j, then m was previously sent to p_j by some process p_i.

Module 2.3 Interface and properties of perfect point-to-point links.

sender that it does not need to keep on sending a given set of messages anymore. This mechanism can be performed whenever a destinator process has properly consumed those messages, or has delivered messages that semantically subsume the previous ones, e.g., in stock exchange applications when new values might subsume old ones. Such a mechanism should however be viewed as an external algorithm, and cannot be integrated within our algorithm implementing stubborn links. Otherwise, the algorithm might not be implementing the stubborn link abstraction anymore.

2.3.4 Perfect Links

With the stubborn link abstraction, it is up to the destinator process to check whether a given message has already been delivered or not. Adding, besides mechanisms for message retransmission, mechanisms for duplicate verification helps build an even higher level abstraction: the *perfect* link one, sometimes also called the *reliable channel* abstraction. The perfect link abstraction specification is captured by the "Perfect Point To Point Link (pp2p)" module, i.e., Module 2.3. The interface of this module also consists of two events: a request event (to send messages) and an indication event (used to deliver messages). Perfect links are characterized by the properties PL1-PL3.

Algorithm 2.2 ("Eliminate Duplicates") describes a very simple implementation of perfect links over stubborn ones. It simply keeps a record of all messages that have been delivered in the past; when a message is received, it is only delivered if it is not a duplicate. We discuss in the following the correctness of the algorithm as well as some performance considerations.

Correctness. Consider the *reliable delivery* property of perfect links. Let m be any message pp2pSent by some process p to some process q and assume

Algorithm 2.2 Eliminate Duplicates.

```
Implements:
    PerfectPointToPointLinks (pp2p).

Uses:
    StubbornPointToPointLinks (sp2p).

upon event ⟨ Init ⟩ do
    delivered := ∅;

upon event ⟨ pp2pSend, dest, m ⟩ do
    trigger ⟨ sp2pSend, dest, m ⟩;

upon event ⟨ sp2pDeliver, src, m ⟩ do
    if $m \notin$ delivered then
        delivered := delivered ∪ { m };
        trigger ⟨ pp2pDeliver, src, m ⟩;
```

that none of these processes crash. By the algorithm, process p sp2pSends m to q using the underlying stubborn links. By the *stubborn delivery* property of the underlying links, q eventually sp2pDelivers m () m at least once and hence pp2pDelivers it. The *no duplication* property follows from the test performed by the algorithm before delivering any message: whenever a message is sp2pDelivered and before pp2pDelivering that message. The *no creation* property simply follows from the *no creation* property of the underlying stubborn links.

Performance. Besides the performance considerations we discussed for our stubborn link implementation, i.e., Algorithm 2.1 ("Retransmit Forever"), and which clearly apply to the perfect link implementation of Algorithm 2.2 ("Eliminate Duplicates"), there is an additional concern related to maintaining the ever growing set of messages *delivered* at every process, provided actual physical memory limitations.

At first glance, one might think of a simple way to circumvent this issue by having the destinator acknowledging messages periodically and the sender acknowledging having received such acknowledgements and promising not to send those messages anymore. There is no guarantee however that such messages would not be still in transit and will later reach the destinator process. Additional mechanisms, e.g., timestamp-based, to recognize such old messages could however be used.

2.3.5 Processes and Links

Throughout this manuscript, we will mainly assume perfect links. It may seem awkward to assume that links are perfect when it is known that real links may crash, lose and duplicate messages. This assumption only captures the fact that these problems can be addressed by some lower level protocol. As long as the network remains connected, and processes do not commit an unbounded number of omission failures, link crashes may be masked by routing. The loss of messages can be masked through re-transmission as we have just explained through our algorithms. This functionality is often found in standard transport level protocols such as TCP. These are typically supported by the operating system and do not need to be re-implemented.

The details of how the perfect link abstraction is implemented is not relevant for the understanding of the fundamental principles of many distributed algorithms. On the other hand, when developing actual distributed applications, these details become relevant. For instance, it may happen that some distributed algorithm requires the use of sequence numbers and message retransmissions, even assuming perfect links. In this case, in order to avoid the redundant use of similar mechanisms at different layers, it may be more effective to rely just on weaker links, such as fair-loss or stubborn links. This is somehow what will happen when assuming the crash-recovery abstraction of a process, as we will explain below.

Indeed, consider the *reliable delivery* property of perfect links: if a process p_i sends a message m to a process p_j, then, unless p_i or p_j crashes, p_j eventually delivers m. With a crash-recovery process abstraction, p_j might indeed deliver m but crash and then recover. If the act of delivering is simply that of transmitting a message, then p_j might not have had the time to do anything useful with the message before crashing. One alternative is to define the act of delivering a message as its logging in stable storage. It is then up to the receiver process to check in its log which messages it has delivered and make use of them. Having to log every message in stable storage might not however be very realistic for the logging being a very expensive operation.

The second alternative in this case is to go back to the fair-loss assumption and build on top of it a retransmission module which ensures that the receiver has indeed the time to perform something useful with the message, even if it crashes and recovers, and without having to log the message. The *stubborn delivery* property ensures exactly that: *if a process p_i sends a message m to a correct process p_j, and p_i does not crash, then p_j delivers m from p_i an infinite number of times.* Hence, the receiver will have the opportunity to do something useful with the message, provided that it is correct. Remember that, with a crash-recovery abstraction, a process is said to be correct if, eventually, it is up and does not crash anymore.

Interestingly, Algorithm 2.1 ("Retransmit Forever") implements stubborn links over fair loss ones also with the crash-recovery abstraction of a process; though with a different meaning of the very notion of a correct process. This is clearly not the case for Algorithm 2.2 ("Eliminate Duplicates"), i.e., this algorithm is not correct with the crash-recovery abstraction of a process.

2.4 Timing Assumptions

An important aspect of the characterization of a distributed system is related with the behaviour of its processes and links with respect to the passage of time. In short, determining whether we can make any assumption on the existence of time bounds on communication bounds and process (relative) speeds is if primary importance when defining a model of a distributed system. We address some time-related issues in this section and then suggest the *failure detector* abstraction as a meaningful way to abstract useful timing assumptions.

2.4.1 Asynchronous System

Assuming an *asynchronous* distributed system comes down to not making any timing assumption about processes and channels. This is precisely what we have been doing so far, i.e., when defining our process and link abstractions. That is, we did not assume that processes have access to any sort of physical clock, nor did we assume any bounds on processing delays and also no bounds on communication delay.

Even without access to physical clocks, it is still possible to measure the passage of time based on the transmission and delivery of messages, i.e., time is defined with respect to communication. Time measured this way is called *logical time*.

The following rules can be used to measure the passage of time in an asynchronous distributed system:

- Each process p keeps an integer called *logical clock* l_p, initially 0.
- Any time an event occurs at process p, the logical clock l_p is incremented by one unit.
- When a process sends a message, it timestamps the message with the value of its logical clock at the moment the message is sent and tags the message with that timestamp. The timestamp of event e is denoted by $t(e)$.
- When a process p receives a message m with timestamp l_m, p increments its timestamp in the following way: $l_p = max(l_p, l_m) + 1$.

An interesting aspect of logical clocks is the fact that they capture cause-effect relations in systems where the processes can only interact through message exchanges. We say that an event e_1 may potentially have caused another event e_2, denoted as $e_1 \rightarrow e_2$ if the following relation, called the *happened-before* relation, applies:

- e_1 and e_2 occurred at the same process p and e_1 occurred before e_2 (Figure 2.4 (a)).
- e_1 corresponds to the transmission of a message m at a process p and e_2 to the reception of the same message at some other process q (Figure 2.4 (b)).

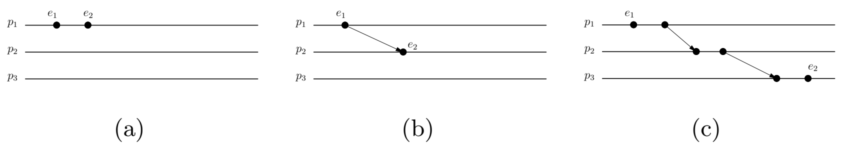

Figure 2.4. The *happened-before* relation.

- there exists some event e' such that $e_1 \rightarrow e'$ and $e' \rightarrow e_2$ (Figure 2.4 (c)).

It can be shown that if the events are timestamped with logical clocks, then $e_1 \rightarrow e_2 \Rightarrow t(e_1) < t(e_2)$. Note that the opposite implication is not true.

As we discuss in the next chapters, even in the absence of any physical timing assumption, and using only a logical notion of time, we can implement some useful distributed programming abstractions. Many abstractions do however need some physical timing assumptions. In fact, even a very simple form of agreement, namely *consensus*, is impossible to solve in an asynchronous system even if only one process fails, and it can only do so by crashing (see the historical note at the end of this chapter). In this problem, which we will address later in this manuscript, the processes start, each with an initial value, and have to agree on a common final value, out the initial values. The consequence of this result is immediate for the impossibility of deriving algorithms for many agreement abstractions, including group membership or totally ordered group communication.

2.4.2 Synchronous System

Whilst assuming an *asynchronous* system comes down not to make any physical timing assumption on processes and links, assuming a *synchronous* system comes down to assuming the following three properties:

1. *Synchronous computation.* There is a known upper bound on processing delays. That is, the time taken by any process to execute a step is always less than this bound. Remember that a step gathers the delivery of a message (possibly *nil*) sent by some other process, a local computation (possibly involving interaction among several layers of the same process), and the sending of a message to some other process.
2. *Synchronous communication.* There is a known upper bound on message transmission delays. That is, the time period between the instant at which a message is sent and the time at which the message is delivered by the destination process is less than this bound.
3. *Synchronous physical clocks.* Processes are equipped with a local physical clock. There is a known upper bound on the rate at which the local physical clock deviates from a global real time clock (remember that we

make here the assumption that such a global real time clock exists in our universe, i.e., at least as a fictional device to simplify the reasoning about the processes, but this is not accessible to the processes).

In a synchronous distributed system, several useful services can be provided, such as, among others:

- *Timed failure detection.* Every crash of a process may be detected within bounded time: whenever a process p crashes, all processes that did not crash, detect the crash of p within a known bounded time. This can be achieved for instance using a heartbeat mechanism, where processes periodically exchange (heartbeat) messages and detect, within a limited time period, the crash of processes that have crashed.
- *Measure of transit delays.* It is possible to get a good approximation the delays spent by messages in the communication links and, from there, infer which nodes are more distant or connected by slower or overloaded links.
- *Coordination based on time.* One can implement a *lease* abstraction that provides the right to execute some action that is granted for a fixed amount of time, e.g., manipulating a specific file.
- *Worst case performance.* By assuming a bound on the number of faults and on the load of the system, it is possible to derive *worst case response times* for a given algorithm. This allows a process to know when a message it has sent has been received by the destination process (provided that the latter is correct). This can be achieved even if we assume that processes commit omission failures without crashing, as long as we bound the number of these omission failures.
- *Synchronized clocks.* A synchronous system makes it possible to synchronize the clocks of the different processes in such a way that they are never apart by more than some known constant δ, known as the clock synchronization precision. Synchronized clocks allow processes to coordinate their actions and ultimately execute synchronized global steps. Using synchronized clocks makes it possible to timestamp events using the value of the local clock at the instant they occur. These timestamps can be used to order events in the system.
  If there was a system where all delays were constant, it would be possible to achieve perfectly synchronized clocks (i.e., where δ would be 0). Unfortunately, such a system cannot be built. In practice, δ is always greater than zero and events within δ cannot be ordered.

Not surprisingly, the major limitation of assuming a synchronous system is the *coverage* of the system, i.e., the difficulty of building a system where the timing assumptions hold with high probability. This typically requires careful analysis of the network and processing load and the use of appropriate processor and network scheduling algorithms. Whilst this might be feasible for some local area networks, it might not be so, or even desirable, in larger scale systems such as the Internet. In this case, i.e., on the Internet, there are periods where messages can take a very long time to arrive to their destination. One should consider very large values to capture the processing and communication bounds. This however would mean considering worst cases values which are typically much higher than average values. These worst case values are usually so high that any application based on them would be very inefficient.

2.4.3 Partial Synchrony

Generally, distributed systems are completely synchronous *most of the time*. More precisely, for most systems we know of, it is relatively easy to define physical time bounds that are respected *most of the time*. There are however periods where the timing assumptions do not hold, i.e., periods during which the system is asynchronous. These are periods where the network is for instance overloaded, or some process has a shortage of memory that slows it down. Typically, the buffer that a process might be using to store incoming and outgoing messages might get overflowed and messages might thus get lost, violating the time bound on the delivery. The retransmission of the messages might help ensure the reliability of the channels but introduce unpredictable delays. In this sense, practical systems are *partially synchronous*.

One way to capture the partial synchrony observation is to assume that the timing assumptions only hold eventually (without stating when exactly). This boils down to assuming that there is a time after which these assumptions hold forever, but this time is not known. In a way, instead of assuming a synchronous system, we assume a system that is eventually synchronous. It is important to notice that making such assumption does not in practice mean that (1) there is a time after which the underlying system (including application, hardware and networking components) is synchronous forever, nor does it mean that (2) the system needs to be initially asynchronous and then only after some (long time) period becomes synchronous. The assumption simply captures the very fact that the system might not always be synchronous, and there is no bound on the period during which it is asynchronous. However, we expect that there are periods during which the system is synchronous, and some of these periods are long enough for an algorithm to terminate its execution.

2.5 Abstracting Time

2.5.1 Failure Detection

So far, we contrasted the simplicity with the inherent limitation of the asynchronous system assumption, as well the power with the limited coverage of the synchronous assumption, and we discussed the intermediate partially synchronous system assumption. Each of these make some sense for specific environments, and need to be considered as plausible assumptions when reasoning about general purpose implementations of high level distributed programming abstractions.

As far as the asynchronous system assumption is concerned, there is no timing assumptions to be made and our process and link abstractions directly capture that case. These are however not sufficient for the synchronous and partially synchronous system assumptions. Instead of augmenting our process and link abstractions with timing capabilities to encompass the synchronous and partially synchronous system assumptions, we consider a separate kind of abstractions to encapsulates those capabilities. Namely, we consider *failure detectors*. As we will discuss in the next section, failure detectors provide information (not necessarily fully accurate) about which processes are crashed. We will in particular introduce a failure detector that encapsulates timing assumptions of a synchronous system, as well as failure detectors that encapsulate timing assumptions of a partially synchronous system. Not surprisingly, the information provided by the first failure detector about crashed processes will be more accurate than those provided by the others. More generally, the stronger are the timing assumptions we make on the distributed system, i.e., to implement the failure detector, the more accurate that information can be.

There are at least two advantages of the failure detector abstraction, over an approach where we would directly make timing assumptions on processes and links. First, the failure detector abstraction alleviates the need for extending the process and link abstractions introduced earlier in this chapter with timing assumptions: the simplicity of those abstractions is preserved. Second, and as will see in the following, we can reason about failure detector properties using axiomatic properties with no explicit references about physical time. Such references are usually very error prone. In practice, except for specific applications like process control, timing assumptions are indeed mainly used to detect process failures, i.e., to implement failure detectors: this is exactly what we do.

2.5.2 Perfect Failure Detection

In synchronous systems, and assuming a process crash-stop abstraction, crashes can be accurately detected using *timeouts*. For instance, assume that a process sends a message to another process and awaits a response. If the recipient process does not crash, then the response is guaranteed to arrive within a time period equal to the worst case processing delay plus two times the worst case message transmission delay (ignoring the clock drifts). Using its own clock, a sender process can measure the worst case delay required to obtain a response and detect a crash in the absence of such a reply within the timeout period: the crash detection will usually trigger a corrective pro-

Module:

Name: PerfectFailureDetector ($\mathcal{P}$).

Events:

Indication: ⟨ *crash*, p_i ⟩: Used to notify that process p_i has crashed.

Properties:

PFD1: *Strong completeness:* Eventually every process that crashes is permanently detected by every correct process.

PFD2: *Strong accuracy:* No process is detected by any process before it crashes.

Module 2.4 Interface and properties of the perfect failure detector.

Algorithm 2.3 Exclude on Timeout.

```
Implements:
    PerfectFailureDetector (𝒫).

Uses:
    PerfectPointToPointLinks (pp2p).

upon event ⟨ Init ⟩ do
    alive := $\Pi$;
    detected := $\emptyset$;
    startTimer (TimeDelay);

upon event ⟨ Timeout ⟩ do
    forall $p_i \in \Pi$ do
        if $p_i \notin$ alive and $p_i \notin$detected then
            detected := detected $\cup \{p_i\}$;
            trigger ⟨ crash, $p_i$ ⟩;
        trigger ⟨ pp2pSend, $p_i$, [HEARTBEAT] ⟩;
    alive := $\emptyset$;
    startTimer (TimeDelay);

upon event ⟨ pp2pDeliver, src, [HEARTBEAT] ⟩ do
    alive := alive $\cup \{src\}$;
```

cedure. We encapsulate such a way of detecting failures in a synchronous system through the use of a *perfect failure detector* abstraction.

Specification. The perfect failure detector outputs, at every process, the set of processes that are detected to have crashed. A perfect failure detector can be described by the *accuracy* and *completeness* properties of Module 2.4, "Perfect Failure Detector ($\mathcal{P}$)". The act of detecting a crash coincides with the triggering of the event *crash*: once the crash of a process p is detected by some process q, the detection is permanent, i.e., q will not change its mind.

Algorithm. Algorithm 2.3, that we call "Exclude on Timeout", implements a perfect failure detector assuming a synchronous system. Communication links do not lose messages sent by a correct process to a correct process (perfect links) and the transmission period of every message is bounded by some known constant, in comparison to which the local processing time of a process, as well as the clock drifts, are negligible. The algorithm makes use of a specific timeout mechanism initialized with a timeout delay chosen to be large enough such that, within that period, every process has enough time to send a message to all, and each of these messages has enough time to be delivered at its destination. Whenever the timeout period expires, the specific *Timeout* event is triggered.

In order for the algorithm no to trigger an infinite number of failure detection events *(crash* p_i) for every faulty process p_i, once an event has been triggered for a given process p_i, we simply put that process in a specific variable *detected* and save the triggering of future failure detection events for p_i.

Correctness. Consider the *strong completeness* property of a perfect failure detector. If a process p crashes, it stops sending heartbeat messages and no process will deliver its messages: remember that perfect links ensure that no message is delivered unless it was sent. Every correct process will thus detect the crash of p.

Consider now the *strong accuracy* property of a perfect failure detector. The crash of a process p is detected by some other process q, only if q does not deliver a message from p before a timeout period. This can only happen if p has indeed crashed because the algorithm makes sure p must have otherwise sent a message and the synchrony assumption implies that the message should have been delivered before the timeout period.

2.5.3 Eventually Perfect Failure Detection

Just like we can encapsulate timing assumptions of a synchronous system in a *perfect failure detector* abstraction, we can similarly encapsulate timing assumptions of a partially synchronous system within an *eventually perfect failure detector* abstraction.

Specification. Basically, the eventually perfect failure detector abstraction guarantees that there is a time after which crashes can be accurately detected. This captures the intuition that, most of the time, timeout delays can be adjusted so they can accurately detect crashes. However, there are periods where the asynchrony of the underlying system prevents failure detection to be accurate and leads to false suspicions. In this case, we talk about failure *suspicion* instead of *detection.*

More precisely, to implement an eventually perfect failure detector abstraction, the idea is to also use a timeout, and to suspect processes that did not send heartbeat messages within a timeout delay. Obviously, a suspicion might be wrong in a partially synchronous system. A process p might suspect a process q, even if q has not crashed, simply because the timeout

Module:

Name: EventuallyPerfectFailureDetector ($\diamond\mathcal{P}$).

Events:

Indication: ⟨ *suspect*, p_i ⟩: Used to notify that process p_i is suspected to have crashed.

Indication: ⟨ *restore*, p_i ⟩: Used to notify that process p_i is not suspected anymore.

Properties:

EPFD1: *Strong completeness:* Eventually, every process that crashes is permanently suspected by every correct process.

EPFD2: *Eventual strong accuracy:* Eventually, no correct process is suspected by any correct process.

Module 2.5 Interface and properties of the eventually perfect failure detector.

delay chosen by p to suspect the crash of q was too short. In this case, p's suspicion about q is false. When p receives a message from q, and p will if p and q are correct, p revises its judgement and stops suspecting q. Process p also increases its timeout delay: this is because p does not know what the bound on communication delay will eventually be; it only knows there will be one. Clearly, if q now crashes, p will eventually suspect q and will never revise its judgement. If q does not crash, then there is a time after which p will stop suspecting q, i.e., the timeout delay used by p to suspect q will eventually be large enough because p keeps increasing it whenever it commits a false suspicion. This is because we assume that there is a time after which the system is synchronous.

An eventually perfect failure detector can be described by the *accuracy* and *completeness* properties (EPFD1-2) of Module 2.5, "Eventually Perfect Failure Detector ($\diamond\mathcal{P}$)". A process p is said to be *suspected* by process q whenever q triggers the event *suspect*(p_i) and does not subsequently trigger the event *restore*(p_i).

Algorithm. Algorithm 2.4, that we have called "Increasing Timeout", implements an eventually perfect failure detector assuming a partially synchronous system. As for Algorithm 2.3 ("Exclude on Timeout"), we make use of a specific timeout mechanism initialized with a timeout delay. The main difference here is that the timeout delay increases whenever a process realizes that it has falsely suspected a process that is actually correct.

Correctness. The *strong completeness* property is satisfied as for Algorithm 2.3 ("Exclude on Timeout"). If a process crashes, it will stop sending messages, will be suspected by every correct process and no process will ever revise its judgement about that suspicion.

Consider now the *eventual strong accuracy* property. Consider the time after which the system becomes synchronous, and the timeout delay becomes

Algorithm 2.4 Increasing Timeout.

```
Implements:
    EventuallyPerfectFailureDetector ($\Diamond\mathcal{P}$).

Uses:
    PerfectPointToPointLinks (pp2p).

upon event ⟨ Init ⟩ do
    alive := $\Pi$;
    suspected := $\emptyset$;
    period := TimeDelay;
    startTimer (period);

upon event ⟨ Timeout ⟩ do
    if alive $\cap$ suspected $\neq \emptyset$ then
        period := period + $\Delta$;
    forall $p_i \in \Pi$ do
        if $p_i \notin$ alive then
            suspected := suspected $\cup \{p_i\}$;
            trigger ⟨ crash, $p_i$ ⟩;
        else
            if $p_i \in$ suspected then
                suspected := suspected $\setminus \{p_i\}$;
                trigger ⟨ restore, $p_i$ ⟩;
        trigger ⟨ pp2pSend, $p_i$, [HEARTBEAT] ⟩;
    alive := $\emptyset$;
    startTimer (period);

upon event ⟨ pp2pDeliver, src, [HEARTBEAT] ⟩ do
    alive := alive $\cup$ { src };
```

larger than message transmission delays (plus clock drifts and local processing periods). After this time, any message sent by a correct process to a correct process is delivered within the timeout delay. Hence, any correct process that was wrongly suspecting some correct process will revise its suspicion and no correct process will ever be suspected by a correct process.

2.5.4 Eventual Leader Election

Often, one may not need to detect which processes have failed, but rather need to agree on a process that has *not* failed and that may act as the coordinator in some steps of a distributed algorithm. This process is in a sense *trusted* by the other processes and elected as their *leader*. The *leader detector* abstraction we discuss here provides such support.

Specification. The *eventual leader detector* abstraction, with the properties (CD1-2) stated in Module 2.6, and denoted by Ω, encapsulates a leader election algorithm which ensures that eventually the correct processes will

Module:

Name: EventualLeaderDetector (Ω).

Events:

Indication: $\langle$ *trust*, p_i $\rangle$: Used to notify that process p_i is trusted to be leader.

Properties:

CD1: *Eventual accuracy:* There is a time after which every correct process trusts some correct process.

CD2: *Eventual agreement:* There is a time after which no two correct processes trust different processes.

Module 2.6 Interface and properties of the eventual leader detector.

elect the same correct process as their leader. Nothing precludes the possibility for leaders to change in an arbitrary manner and for an arbitrary period of time. Once a unique leader is determined, and does not change again, we say that the leader has *stabilized*. Such a stabilization is guaranteed by the specification of Module 2.6.

Algorithms. With a crash-stop process abstraction, Ω can be obtained directly from $\diamond\mathcal{P}$. Indeed, it is is enough to trust the process with the highest identifier among all processes that are not suspected by $\diamond\mathcal{P}$. Eventually, and provided at least one process is correct, exactly one correct process will be trusted by all correct processes.

Interestingly, the leader abstraction Ω can also be implemented with the process crash-recovery abstraction, also using timeouts and assuming the system to be partially synchronous. Algorithm 2.5 describes such implementation assuming that at least one process is correct. Remember that this implies, with a process crash-recovery abstraction, that at least one process does never crash, or eventually recovers and never crashes again (in every execution of the algorithm).

In Algorithm 2.5, called "elect Lower Epoch", every process p_i keeps track of how many times it crashed and recovered, within an *epoch* integer variable. This variable, representing the *epoch number* of p_i, is retrieved, incremented, and then stored in stable storage whenever p_i recovers from a crash. The goal of the algorith is to elect has a leader the active process with lower epoch, *i.e.*, that has crashed and recovered less times.

Process p_i periodically sends to all a *heartbeat* message together with its current epoch number. Besides, every process p_i keeps a list of potential leader processes, within the variable *candidate*. Initially, at every process p_i, *candidate* contains all processes. Then any process that does not communicate with p_i is excluded from *candidate*. A process p_j that communicates with p_i, after having recovered or being slow in communicating with p_i, is simply added again to *candidate*, i.e., considered a potential leader for p_i.

Initially, the leader for all processes is the same and is process p_1. After every timeout delay, p_i checks whether p_1 can still be the leader. This test is performed through a function *select* that returns one process among a set of processes, or nothing if the set is empty. The function is the same at all processes and returns the same process (identifier) for the same given set (*candidate*), in a deterministic manner and following the following rule: among processes with the lowest epoch number, the process with the lowest index is returned. This guarantees that, if a process p_j is elected leader, and p_j keeps on crashing and recovering forever, p_j will eventually be replaced by a correct process. By definition, the epoch number of a correct process will eventually stop increasing.

A process increases its timeout delay whenever it changes a leader. This guarantees that, eventually, if leaders keep changing because of the timeout delay being too short with respect to communication delays, the delay will increase and become large enough for the leader to stabilize when the system becomes synchronous.

Correctness. Consider the *eventual accuracy* property and assume by contradiction that there is a time after which a correct process p_i permanently trusts the same faulty process, say p_j. There are two cases to consider (remember that we consider a crash-recovery process abstraction): (1) process p_j eventually crashes and never recovers again, or (2) process p_j keeps crashing and recovering forever.

Consider case (1). Since p_j crashes and does never recover again, p_j will send its *heartbeat* messages to p_i only a finite number of times. By the *no creation* and *finite duplication* properties of the underlying links (fair loss), there is a time after which p_i stops delivering such messages from p_i. Eventually, p_j will be excluded from the set (*candidate*) of potential leaders for p_i and p_i will elect a new leader.

Consider now case (2). Since p_j keeps on crashing and recovering forever, its epoch number will keep on increasing forever. If p_k is a correct process, then there is a time after which its epoch number will be lower than that of p_j. After this time, either (2.1) p_i will stop delivering messages from p_j, and this can happen if p_j crashes and recovers so quickly that it does not have the time to send enough messages to p_i (remember that with fail loss links, a message is guaranteed to be delivered by its destinator only it is sent infinitely often), or (2.2) p_i delivers messages from p_j but with higher epoch numbers than those of p_k. In both cases, p_i will stop trusting p_j.

Process p_i will eventually trust only correct processes.

Consider now the *eventual agreement* property. We need to explain why there is a time after which no two different processes are trusted by two correct processes. Consider the subset of correct processes in a given execution S. Consider furthermore the time after which (a) the system becomes synchronous, (b) the processes in S do never crash again, (c) their epoch numbers stop increasing at every process, and (d) for every correct process

Algorithm 2.5 Elect Lower Epoch.

```
Implements:
    EventualLeaderDetector ($\Omega$).

Uses:
    FairLossPointToPointLinks (flp2p)

upon event ⟨ Init ⟩ do
    leader := $p_1$;
    candidate := $\Pi$;
    period := TimeDelay;
    startTimer (TimeDelay);
    epoch := 0;

upon event ⟨ Recovery ⟩ do
    leader := $p_1$;
    candidate := $\Pi$;
    period := TimeDelay;
    startTimer (TimeDelay);
    retrieve(epoch); epoch := epoch + 1; store(epoch);

upon event ⟨ Timeout ⟩ do
    if leader $\neq$ select(candidate) then
        period := period + $\Delta$;
        leader := select(candidate);
        trigger ⟨ trust, leader ⟩;
    candidate := $\emptyset$;
    forall $p_i \in \Pi$ do
        trigger ⟨ sp2pSend, $p_i$, [HEARTBEAT, epoch] ⟩;
    startTimer (period);

upon event ⟨ flp2pDeliver, src, [HEARTBEAT, epc] ⟩ do
    if exists $(s, e) \in$ candidate such that (s= src) $\wedge$ (e<epc) then
        candidate := candidate $\setminus$ $\{(s, e)\}$;
    candidate := candidate $\cup$ { (src, epc) };
```

p_i and every faulty process p_j, p_i stops delivering messages from p_j, or p_j's epoch number at p_i gets strictly larger than the largest epoch number of S's processes at p_i. By the assumptions of a partially synchronous system, the properties of the underlying fair loss channels and the algorithm, this time will eventually be reached. After it is reached, every process that is trusted by a correct process will be one of the processes in S. By the function *select* all correct processes will trust the same process within this set.

2.6 Distributed System Models

A combination of (1) a process abstraction, (2) a link abstraction and (3) (possibly) a failure detector abstraction defines a *distributed system model*.

In the following, we discuss five models that will be considered throughout this manuscript to reason about distributed programming abstractions and the algorithms used to implement them. We will also discuss some important properties of abstraction specifications and algorithms that will be useful reasoning tools for the following chapters.

2.6.1 Combining Abstractions

Clearly, we will not consider all possible combinations of basic abstractions. On the other hand, it is interesting to discuss more than one possible combination to get an insight on how certain assumptions affect the algorithm design. We have selected five specific combinations to define five different models studied in this manuscript. Namely, we consider the following models:

- **Fail-stop**. We consider the crash-stop process abstraction, where the processes execute the deterministic algorithms assigned to them, unless they possibly crash, in which case they do not recover. Links are considered to be perfect. Finally, we assume the existence of a perfect failure detector ($\mathcal{P}$) (Module 2.4). As the reader will have the opportunity to observe, when comparing algorithms in this model with algorithms in the four other models discussed below, making these assumptions substantially simplify the design of distributed algorithms.
- **Fail-silent**. We also consider here the crash-stop process abstraction together with perfect links. Nevertheless, we do not assume here any failure detection abstraction: that is, the processes have no means to get any information about other processes having crashed.
- **Fail-noisy**. This case is somehow intermediate between the two above. We also consider here the crash-stop process abstraction together with perfect links. In addition, we assume here the existence of the eventually perfect failure detector ($\diamond\mathcal{P}$) of Module 2.5 or the eventual leader detector (Ω) of Module 2.6.
- **Fail-recovery**. We consider here the crash-recovery process abstraction, according to which processes may crash and later recover and still participate in the algorithm. Algorithms devised with this basic abstraction in mind have to deal with the management of stable storage and with the difficulties of dealing with amnesia, i.e., the fact that a process might forget what it might have done prior to crashing. Links are assumed to be stubborn and we might rely on the eventual leader detector (Ω) of Module 2.6.
- **Randomized**. We will consider here a specific particularity in the process abstraction: algorithms might not be deterministic. That is, the processes might use a random oracle to choose among several steps to execute. Typically, the corresponding algorithms implement a given abstraction with some (hopefully high) probability.

It is important to notice that some of the abstractions we study cannot be implemented in all models. For example, the coordination abstractions we consider in Chapter 7 do not have fail-silent solutions and it is not clear either how to devise meaningful randomized solutions to such abstractions. For other abstractions, such solutions might exist but devising them is still an active area of research. This is for instance the case for randomized solutions to the shared memory abstractions we consider in Chapter 4.

2.6.2 Measuring Performance

When we present an algorithm that implements a given abstraction, we analyze its cost mainly using two metrics: (1) the number of messages required to terminate an operation of the abstraction, and (2) the number of communication steps required to terminate such an operation. When evaluating the performance of distributed algorithms in a crash-recovery model, besides the number of communication steps and the number of messages, we also consider (3) the number of accesses to stable storage (also called logs).

In general, we count the messages, communication steps, and disk accesses in specific executions of the algorithm, specially executions when no failures occur. Such executions are more likely to happen in practice and are those for which the algorithms are optimized. It does make sense indeed to plan for the worst, by providing means in the algorithms to tolerate failures, and hope for the best, by optimizing the algorithm for the case where failures do not occur. Algorithms that have their performance go proportionally down when the number of failures increass are sometimes called *gracefully degrading* algorithms.

Precise performance studies help select the most suitable algorithm for a given abstraction in a specific environment and conduct *real-time* analysis. Consider for instance an algorithm that implements the abstraction of perfect communication links and hence ensures that every message sent by a correct process to a correct process is eventually delivered by the latter process. It is important to notice here what such a property states in terms of timing guarantees: for every execution of the algorithm, and every message sent in that execution, there is a time delay within which the message is eventually delivered. The time delay is however defined *a posteriori*. In practice one would require that messages be delivered within some time delay defined *a priori*, for every execution and possibly every message. To determine whether a given algorithm provides this guarantee in a given environment, a careful performance study needs to be conducted on the algorithm, taking into account various parameters of the environment, such as the operating system, the scheduler, and the network. Such studies are out of the scope of this manuscript. We indeed present algorithms that are applicable to a wide range of distributed systems, where bounded delays cannot be enforced, and where infrastructures such as real-time are not strictly required.

Hands-On

We now describe the implementation of some of the abstractions presented in this chapter. However, before proceeding we need to introduce some additional components of the *Appia* framework.

Sendable Event

For the implementation of the protocols that we will be describing, we have defined a specialization of the basic *Appia* Event, called SendableEvent. The interface of this event is presented in Listing 2.1.

Listing 2.1. SendableEvent interface.

```
public class SendableEvent extends Event implements Cloneable {
    public Object dest;
    public Object source;
    protected Message message;

    public SendableEvent();
    public SendableEvent(Channel channel, int dir, Session source);
    public SendableEvent(Message msg);
    public SendableEvent(Channel channel, int dir, Session source, Message msg);

    public Message getMessage();
    public void setMessage(Message message);
    public Event cloneEvent();
}
```

A SendableEvent owns three relevant attributes, namely: a Message, that contains the data to be sent on the network, the source, that identifies the sending process; and the destination attribute, that identifies the recipient processes.

Since our implementation are based on low-level protocols from the IP family, processes will be identified by a tuple (ip address, port). Therefore, both the source and the dest attributes should contain an object of type InetWithPort (used by java TCP and UDP interface).

Message and Extended Message

The Message component is provided by the *Appia* framework to simplify the task of adding and extracting protocol headers to/from the message payload. Chunks of data can be added or extracted from the message using the auxiliary MsgBuffer data structure, depicted in Listing 2.2.

Listing 2.2. MsgBuffer interface.

```
public class MsgBuffer {
  public byte[] data;
  public int off;
  public int len;
  public MsgBuffer();
  public MsgBuffer(byte[] data, int off, int len);
}
```

The interface of the Message object is partially listed in Listing 2.3. Note the methods to push and pop MsgBuffers to/from a message, as well as methods to fragment and concatenate messages.

Listing 2.3. Message interface (partial).

```
public class Message implements Cloneable {

    public Message();

    public int length();
    public void peek(MsgBuffer mbuf);
    public void pop(MsgBuffer mbuf);
    public void push(MsgBuffer mbuf);
    public void frag(Message m, int length);
    public void join(Message m);
    public Object clone() throws CloneNotSupportedException;
}
```

To ease the programming of distributed protocols in java, the basic Message class was extended to allow arbitrary objects to be pushed and pop. The class that provides this extended functionality is the ExtendedMessage class, whose interface is depicted in Listing 2.4. This is the class we will be using throughout the remaining of the book.

Listing 2.4. ExtendedMessage interface (partial).

```
public class ExtendedMessage extends Message {

        public ExtendedMessage(); {

        public void pushObject(Object obj);
        public void pushLong(long l);
        public void pushInt(int i);
        /* ... */
        public Object popObject();
        public long popLong();
        public int popInt();
        /* ... */
        public Object peekObject();
        public long peekLong();
        public int peekInt();
        /* ... */
        public Object clone() throws CloneNotSupportedException;
}
```

Fair Loss Point to Point Links

The Fair Loss Point to Point Links abstraction is implemented in *Appia* by the UdpSimple protocol. The UdpSimple protocol uses UDP sockets as unreliable communication channels. When a UdpSimple session receives a SendableEvent with the down direction (i.e., a transmission request) it extracts the message from the event and pushes it to the TCP socket. When a message is received from a TCP socket, a SendableEvent is created with the up direction.

Perfect Point to Point Links

The Perfect Point to Point Links abstraction is implemented in *Appia* by the TcpComplete protocol. As its name implies, this implementation is based on the TCP protocol, more precisely, it uses TCP sockets as communication channels. When a TcpComplete session receives a SendableEvent with the down direction (i.e., a transmission request) it extracts the message from the event and pushes it to the TCP socket. When a message is received from a TCP socket, a SendableEvent is created with the up direction.

A TcpComplete session automatically establishes a TCP connection when requested to send a message to a given destination for the first time. Therefore, a single session implements multiple point-to-point links.

It should be noted that, in pure asynchronous systems, this implementation is just an approximation of the Perfect Point-to-Point Link abstraction. In fact, TCP includes acknowledgements and retransmission mechanisms (to recover from omissions in the network). However, if the other endpoint is unresponsive, TCP breaks the connection, assuming that the corresponding node has crashed. Therefore, TCP makes synchronous assumptions about the system and fails to deliver the messages when it erroneously "suspects" correct processes.

Perfect Failure Detector

In this case, the Perfect Failure Detector (PFD) is only used with Perfect Point to Point Links (PP2PL), which are builded using TCP channels. When a TCP socket is closed, the protocol that implements PP2PL sends an event to the *Appia* channel. This event is accepted by the PFD protocol, which sends a Crash event to notify other layers. The implementation of this notification is shown in Listing 2.5. The protocols that need a PFD declare that will accept the Crash event and will process it, as shown in the implementation of the reliable broadcast protocols, which are described in the next Section.

To notify other layers of a closed socket, the PP2PL protocol must first create the corresponding TCP sockets. The way the PP2PL is implemented, these sockets are open on-demand, i.e., when there is the need to send/receive something from a remote peer. To ensure that these sockets are created, the PFD session send a message to all other processes when it is started.

Note that the PFD abstraction assumes that all processes are started before it starts operating. Therefore, the user must start all processes before activating the perfect failure detector. Otherwise, the detector may detect as failed processes that have not yet been launched. Therefore, in subsequent Chapters, when using the perfect failure detector in conjunction with other protocols, you will be requested to explicitly start the perfect failure detector. In most test applications, this is achieved by issuing the `startpfd` request on the command line. The implementation is ilustrated in Listing 2.5.

Listing 2.5. Perfect failure detector implementation.

```
public class PerfectFailureDetectorSession extends Session {
  private Channel channel;
  private ProcessSet processes;
  private boolean started;

  public PerfectFailureDetectorSession(Layer layer) {
    super(layer);
    started = false;
  }

  public void handle(Event event) {
    if (event instanceof TcpUndeliveredEvent)
      notifyCrash((TcpUndeliveredEvent) event);
    else if (event instanceof ChannelInit)
      handleChannelInit((ChannelInit) event);
    else if (event instanceof ProcessInitEvent)
      handleProcessInit((ProcessInitEvent) event);
    else if(event instanceof PFDStartEvent)
      handlePFDStart((PFDStartEvent) event);
  }

  private void handleChannelInit(ChannelInit init) {
    channel = init.getChannel();
    init.go();
  }

  private void handleProcessInit(ProcessInitEvent event) {
    processes = event.getProcessSet();
    event.go();
  }

  private void handlePFDStart(PFDStartEvent event) {
    started = true;
    event.go();
    CreateChannelsEvent createChannels =
      new CreateChannelsEvent(channel,Direction.DOWN,this);
    createChannels.go();
  }

  private void notifyCrash(TcpUndeliveredEvent event) {
    if(started){
      SampleProcess p = processes.getProcess((InetWithPort) event.who);
      if (p.isCorrect()) {
        p.setCorrect(false);
        Crash crash =
          new Crash(channel,Direction.UP,this,p.getProcessNumber());
        crash.go();
      }
    }
  }
}
```

Exercises

Exercise 2.1 *Explain under which assumptions (a) the fail-recovery and (b) the fail-silent models where any process can commit omission failures are similar?*

Exercise 2.2 *Does the following statement satisfy the synchronous processing assumption:* on my server, no request ever takes more than one week to be processed*?*

Exercise 2.3 *Can we implement the perfect failure detector in a model where the processes could commit omissions failures but where we could not bound the number of such failures? What if this number is bounded but unknown? What if processes that can commit omission failures commit a limited and known number of such failures and then crash?*

Exercise 2.4 *In a fail-stop model, can we determine* a priori *a time period, such that, whenever a process crashes, all correct processes suspect this process to have crashed after this period?*

Exercise 2.5 *In a fail-stop model, which of the following properties are safety properties:*

1. *every process that crashes is eventually detected;*
2. *no process is detected before it crashes;*
3. *no two processes decide differently;*
4. *no two correct processes decide differently;*
5. *every correct process decides before X time units;*
6. *if some correct process decides, then every correct process decides.*

Exercise 2.6 *Consider any algorithm A that implements a distributed programming abstraction M using a failure detector D that is assumed to be eventually perfect. Can A violate the safety property of M if failure detector D is not eventually perfect, e.g., D permanently outputs the empty set?*

Exercise 2.7 *Specify a distributed programming abstraction M and an algorithm A implementing M using a failure detector D that is supposed to satisfy a set of properties, such that the liveness of M is violated if D does not satisfy its properties.*

Solutions

Solution 2.1 When processes crash, they lose the content of their volatile memory and they commit omissions. If we assume (1) that processes do have stable storage and store every update on their state within the stable storage, and (2) that they are not aware they have crashed and recovered, then the two models are similar. □

Solution 2.2 Yes. This is because the time it takes for the process (i.e. the server) to process a request is bounded and known: it is one week. □

Solution 2.3 It is impossible to implement a perfect failure detector if the number of omissions failures is unknown. Indeed, to guarantee the *strong completeness* property of the failure detector, a process p must detect the crash of another one q after some timeout delay. No matter how this delay is chosen, it can however exceed the tranmission delay times the number of omissions that q commits. This would lead to violate the *strong accuracy* property of the failure detector. If the number of possible omissions is known in a synchronous system, we can use it to calibrate the timeout delay of the processes to accurately detect failures. If the delay exceeds the maximum time during which a process can commit omission failures without having actually crashed, it can safely detect the process to have crashed. □

Solution 2.4 No. The perfect failure detector only ensures that processes that crash are eventually detected: there is no bound on the time it takes for these crashes to be detected. This points out a fundamental difference between algorithms assuming a synchronous system and algorithms assuming a perfect failure detector (fail-stop model). In a precise sense, a synchronous model is strictly stronger. □

Solution 2.5

1. Eventually, every process that crashes is eventually detected. This is a liveness property; we can never exhibit a time t in some execution and state that the property is violated. There is always the hope that eventually the failure detector detects the crashes.
2. No process is detected before it crashes. This is a safety property. If a process is detected at time t before it has crashed, then the property is violated at time t.
3. No two processes decide differently. This is also a safety property, because it can be violated at some time t and never be satisfied again.
4. No two correct processes decide differently. If we do not bound the number of processes that can crash, then the property turns out to be a liveness property. Indeed, even if we consider some time t at which two processes have decided differently, then there is always some hope that,

eventually, some of the processes might crash and validate the property. This remains actually true even if we assume that at least one process is correct.
Assume now that we bound the number of failures, say by $F < N - 1$. The property is not anymore a liveness property. Indeed, if we consider a partial execution and a time t at which $N - 2$ processes have crashed and the two remaining processes, decide differently, then there is not way we can extend this execution and validate the property. But is the property a safety property? This would mean that in any execution where the property does not hold, there is a partial execution of it, such that no matter how we extend it, the property would still not hold. Again, this is not true. To see why, Consider the execution where less than $F - 2$ processes have crashed and two correct processes decide differently. No matter what partial execution we consider, we can extend it by crashing one of the two processes that have decided differently and validate the property. To conclude, in the case where $F < N - 1$, the property is the union of both a liveness and a safety property.

5. Every correct process decides before X time units. This is a safety property: it can be violated at some t, where all correct processes have executed X of their own steps. If violated, at that time, there is no hope that it will be satisfied again.
6. If some correct process decides, then every correct process decides. This is a liveness property: there is always the hope that the property is satisfied. It is interesting to note that the property can actually be satisfied by having the processes not doing anything. Hence, the intuition that a safety property is one that is satisfied by doing nothing might be misleading.

□

Solution 2.6 No. Assume by contradiction that A violates the safety property of M if D does not satisfy its properties. Because of the very nature of a safety property, there is a time t and an execution R of the system such that the property is violated at t in R. Assume now that the properties of the eventually perfect failure detector hold after t in a run R' that is similar to R up to time t. A would violate the safety property of M in R', even if the failure detector is eventually perfect. □

Solution 2.7 An example of such abstraction is simply the eventually perfect failure detector. Note that such abstraction has no safety property. □

Historical Notes

- In 1978, the notions of causality and logical time were introduced in probably the most influential paper in the area of distributed computing: (Lamport 1978).
- In 1982, In (Lamport, Shostak, and Pease 1982), agreement problems were considered in an arbitrary fault-model, also called the malicious or the Byzantine model.
- In 1984, algorithms which assume that processes can only fail by crashing and every process has accurate information about which process has crashed have been called fail-stop algorithms (Schneider, Gries, and Schlichting 1984).
- In 1985, it was proven that, even a very simple form of agreement, namely consensus, is impossible to solve with a deterministic algorithm in an asynchronous system even if only one process fails, and it can only do so by crashing (Fischer, Lynch, and Paterson 1985).
- In 1988, intermediate models between the synchronous and the asynchronous model were introduced to circumvent the consensus impossibility (Dwork, Lynch, and Stockmeyer 1988).
- In 1989, the use of synchrony assumptions to build leasing mechanisms was explored (Gray and Cheriton 1989).
- In 1991 (Chandra and Toueg 1996; Chandra, Hadzilacos, and Toueg 1996), it was observed that, when solving consensus, timing assumptions where mainly used to detect process crashes. This observation led to the definition of an abstract notion of failure detector that encapsulates timing assumptions. The very fact that consensus can be solved in eventually synchronous systems (Dwork, Lynch, and Stockmeyer 1988) is translated, in the parlance of (Chandra, Hadzilacos, and Toueg 1996), by saying that consensus can be solved even with unreliable failure detectors.
- In 2000, the notion of unreliable failure detector was precisely defined (Guerraoui 2000). Algorithms that rely on such failure detectors have been called *indulgent* algorithms in (Guerraoui 2000; Dutta and Guerraoui 2002).
- In 1985, the notions of safety and liveness were considered and it was shown that any property of a distributed system execution can be viewed as a composition of a liveness and a safety property (Alpern and Schneider 1985; Schneider 1987).

3. Reliable Broadcast

He said: "I could have been someone";
She replied: "So could any one".
(The Pogues)

This chapter covers the specifications of *broadcast communication* abstractions. These are used to disseminate information among a set of processes and they differ according to the reliability of the dissemination. For instance, *best-effort broadcast* guarantees that all correct processes deliver the same set of messages if the senders are correct. Stronger forms of reliable broadcast guarantee this property even if the senders crash while broadcasting their messages.

We will consider several related abstractions: *best-effort broadcast, (regular) reliable broadcast, uniform reliable broadcast, logged broadcast, stubborn broadcast and probabilistic broadcast.* For each of these abstractions, we will provide several algorithms implementing it, and these will cover the different models addressed in this book.

3.1 Motivation

3.1.1 Client-Server Computing

In traditional distributed applications, interactions are often established between two processes. Probably the most representative of this sort of interaction is the now classic *client-server* scheme. According to this model, a *server* process exports an interface to several *clients.* Clients use the interface by sending a request to the server and by later collecting a reply. Such interaction is supported by *point-to-point* communication protocols. It is extremely useful for the application if such a protocol is *reliable.* Reliability in this context usually means that, under some assumptions (which are by the way often not completely understood by most system designers), messages exchanged between the two processes are not lost or duplicated, and are delivered in the order in which they were sent. Typical implementations of this abstraction are reliable transport protocols such as TCP. By using a reliable point-to-point communication protocol, the application is free from dealing explicitly with issues such as acknowledgments, timeouts, message re-transmissions, flow-control and a number of other issues that become encapsulated by the protocol interface. The programmer can focus on the actual functionality of the application.

3.1.2 Multi-Participant Systems

As distributed applications become bigger and more complex, interactions are no longer limited to bilateral relationships. There are many cases where more than two processes need to operate in a coordinated manner. Consider, for instance, a multi-user virtual environment where several users interact in a virtual space. These users may be located at different physical places, and they can either directly interact by exchanging multimedia information, or indirectly by modifying the environment.

It is convenient to rely here on *broadcast* abstractions. These allow a process to send a message within a *group* of processes, and make sure that the processes agree on the messages they deliver. A naive transposition of the reliability requirement from point-to-point protocols would require that no message sent to the group be lost or duplicated, i.e., the processes agree to deliver every message broadcast to them. However, the definition of agreement for a broadcast primitive is not a simple task. The existence of multiple senders and multiple recipients in a group introduces degrees of freedom that do not exist in point-to-point communication. Consider for instance the case where the sender of a message fails by crashing. It may happen that some recipients deliver the last message while others do not. This may lead to an inconsistent view of the system state by different group members. Roughly speaking, broadcast abstractions provide reliability guarantees ranging from *best-effort*, that only ensures delivery among all correct processes if the sender does not fail, through *reliable* that, in addition, ensures *all-or-nothing* delivery semantics even if the sender fails, to *totally ordered* that furthermore ensures that the delivery of messages follow the same global order, and *terminating* which ensures that the processes either deliver a message or are eventually aware that they will never deliver the message. In this chapter, we will focus on best-effort and reliable broadcast abstractions. Totally ordered and terminating forms of broadcast will be considered later in this manuscript.

Module:

Name: BestEffortBroadcast (beb).

Events:

Request: ⟨ *bebBroadcast*, m ⟩: Used to broadcast message m to all processes.

Indication: ⟨ *bebDeliver*, src, m ⟩: Used to deliver message m broadcast by process *src*.

Properties:

BEB1: *Best-effort validity:* For any two processes p_i and p_j. If p_i and p_j are correct, then every message broadcast by p_i is eventually delivered by p_j.

BEB2: *No duplication:* No message is delivered more than once.

BEB3: *No creation:* If a message m is delivered by some process p_j, then m was previously broadcast by some process p_i.

Module 3.1 Interface and properties of best-effort broadcast.

3.2 Best-Effort Broadcast

A broadcast abstraction enables a process to send a message, in a one-shot operation, to all the processes in a system, including itself. We give here the specification and algorithm for a broadcast communication primitive with a weak form of reliability, called *best-effort broadcast*.

3.2.1 Specification

With best-effort broadcast, the burden of ensuring reliability is put only on the sender. Therefore, the remaining processes do not have to be concerned with enforcing the reliability of received messages. On the other hand, no delivery guarantees are offered in case the sender fails. More precisely, best-effort broadcast is characterized by the properties BEB1-3 depicted in Module 3.1. BEB1 is a liveness property whereas BEB2 and BEB3 are safety properties. Note that broadcast messages are implicitly addressed to all processes. Remember also that messages are uniquely identified.

3.2.2 Fail-Silent Algorithm: Basic Broadcast

We first provide an algorithm that implements best effort broadcast using perfect links. This algorithm does not make any assumption on failure detection: it is a fail-silent algorithm. To provide best effort broadcast on top of perfect links is quite simple. It suffices to send a copy of the message to every process in the system, as depicted in Algorithm 3.1, called “Basic Broadcast”, and illustrated by Figure 3.1. As long as the sender of the message does not

Algorithm 3.1 Basic Broadcast.

```
Implements:
    BestEffortBroadcast (beb).

Uses:
    PerfectPointToPointLinks (pp2p).

upon event ⟨ bebBroadcast, m ⟩ do
    forall $p_i \in \Pi$ do
        trigger ⟨ pp2pSend, $p_i, m$ ⟩;

upon event ⟨ pp2pDeliver, $p_i$, $m$ ⟩ do
    trigger ⟨ bebDeliver, $p_i$, $m$ ⟩;
```

crash, the properties of perfect links ensure that all correct processes will deliver the message.

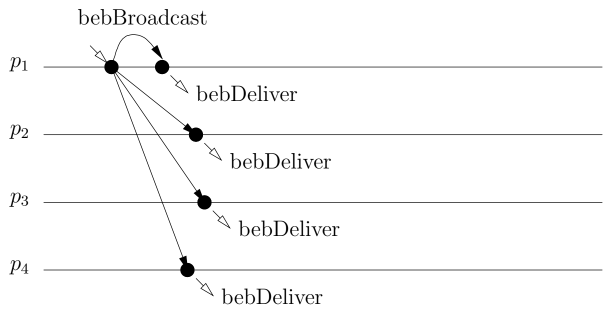

Figure 3.1. Sample execution of Basic Broadcast algorithm.

Correctness. The properties are trivially derived from the properties of perfect point-to-point links. *No duplication* and *no creation* are safety properties that are derived from PL2 and PL3. *Validity* is a liveness property that is derived from PL1 and from the fact that the sender sends the message to every other process in the system.

Performance. The algorithm requires a single communication step and exchanges N messages.

3.3 Regular Reliable Broadcast

Best-effort broadcast ensures the delivery of messages as long as the sender does not fail. If the sender fails, the processes might disagree on whether or not to deliver the message. Actually, even if the process sends a message

Module:

Name: (regular)ReliableBroadcast (rb).

Events:

Request: ⟨ *rbBroadcast*, m ⟩: Used to broadcast message m.

Indication: ⟨ *rbDeliver*, src, m ⟩: Used to deliver message m broadcast by process src.

Properties:

RB1: *Validity:* If a correct process p_i broadcasts a message m, then p_i eventually delivers m.

RB2: *No duplication:* No message is delivered more than once.

RB3: *No creation:* If a message m is delivered by some process p_j, then m was previously broadcast by some process p_i.

RB4: *Agreement:* If a message m is delivered by some correct process p_i, then m is eventually delivered by every correct process p_j.

Module 3.2 Interface and properties of regular reliable broadcast.

to all processes before crashing, the delivery is not ensured because perfect links do not enforce delivery when the sender fails. We now consider the case where agreement is ensured even if the sender fails. We do so by introducing a broadcast abstraction with a stronger form of reliability, called *(regular) reliable broadcast.*

3.3.1 Specification

Intuitively, the semantics of a reliable broadcast algorithm ensure that correct processes agree on the set of messages they deliver, even when the senders of these messages crash during the transmission. It should be noted that a sender may crash before being able to transmit the message, case in which no process will deliver it. The specification is given in Module 3.2. This extends the specification of Module 3.1 with a new liveness property: *agreement.*

3.3.2 Fail-Stop Algorithm: Lazy Reliable Broadcast

To implement regular reliable broadcast, we make use of the best-effort abstraction described in the previous section as well as the perfect failure detector module introduced earlier in the manuscript (i.e., we consider a fail-stop algorithm). This is depicted in Algorithm 3.2, that we have called "Lazy Reliable Broadcast".

To rbBroadcast a message, a process uses the best-effort broadcast primitive to disseminate the message to all, i.e., it bebBroadcasts the message. Note that this implementation adds some protocol headers to the messages exchanged. In particular, the protocol adds a message descriptor ("DATA")

Algorithm 3.2 Lazy Reliable Broadcast.

```
Implements:
    ReliableBroadcast (rb).

Uses:
    BestEffortBroadcast (beb).
    PerfectFailureDetector ($\mathcal{P}$).

upon event ⟨ Init ⟩ do
    delivered := $\emptyset$;
    correct := $\Pi$;
    forall $p_i \in \Pi$ do
        from[$p_i$] := $\emptyset$;

upon event ⟨ rbBroadcast, $m$ ⟩ do
    trigger ⟨ bebBroadcast, [Data, self, $m$] ⟩;

upon event ⟨ bebDeliver, $p_i$, [Data, $s_m$, $m$] ⟩ do
    if $m \notin$ delivered then
        delivered := delivered $\cup$ {$m$}
        trigger ⟨ rbDeliver, $s_m$, $m$ ⟩;
        from[$p_i$] := from[$p_i$] $\cup$ {($s_m$, $m$)}
        if $p_i \notin$ correct then
            trigger ⟨ bebBroadcast, [Data, $s_m$, $m$] ⟩;

upon event ⟨ crash, $p_i$ ⟩ do
    correct := correct \ {$p_i$}
    forall [$s_m$, $m$] $\in$ from[$p_i$] do
        trigger ⟨ bebBroadcast, [Data, $s_m$, $m$] ⟩;
```

and the original source of the message to the protocol header. This is denoted by [DATA, s_m, m] in the algorithm. A process that gets the message (i.e., bebDelivers the message) delivers it immediately (i.e., rbDelivers it). If the sender does not crash, then the message will be delivered by all correct processes. The problem is that the sender might crash. In this case, the process that delivers the message from some other process can detect that crash and relays the message to all. It is important to notice here that this is a language abuse: in fact, the process relays a copy of the message (and not the message itself).

Our algorithm is said to be *lazy* in the sense that it only retransmits a message if the original sender has been detected to have crashed.

Correctness. The *no creation* (resp. *validity*) property of our reliable broadcast algorithm follows from the *no creation* (resp. *validity*) property of the underlying best effort broadcast primitive. The *no duplication* property of reliable broadcast follows from our use of a variable *delivered* that keeps track of the messages that have been rbDelivered at every process. *Agreement* follows here from the *validity* property of the underlying best effort broadcast

Algorithm 3.3 Eager Reliable Broadcast.

```
Implements:
    ReliableBroadcast (rb).

Uses:
    BestEffortBroadcast (beb).

upon event ⟨ Init ⟩ do
    delivered := ∅;

upon event ⟨ rbBroadcast, $m$ ⟩ do
    delivered := delivered ∪ {$m$}
    trigger ⟨ rbDeliver, self, $m$ ⟩;
    trigger ⟨ bebBroadcast, [DATA, self, $m$] ⟩;

upon event ⟨ bebDeliver, $p_i$, [DATA, $s_m$, $m$] ⟩ do
    if $m \notin$ delivered do
        delivered := delivered ∪ { m }
        trigger ⟨ rbDeliver, $s_m$, $m$ ⟩;
        trigger ⟨ bebBroadcast, [DATA, $s_m$, $m$] ⟩;
```

primitive, from the fact that every process relays every message it rbDelivers when it suspects the sender, and from the use of a perfect failure detector.

Performance. If the initial sender does not crash, to rbDeliver a message to all processes, the algorithm requires a single communication step and N messages. Otherwise, in the worst case, if the processes crash in sequence, N steps and N^2 messages are required to terminate the algorithm.

3.3.3 Fail-Silent Algorithm: Eager Reliable Broadcast

In our lazy reliable broadcast algorithm (Algorithm 3.2), we make use of the *completeness* property of the failure detector to ensure the broadcast *agreement.* If the failure detector does not ensure *completeness*, then the processes might not be relaying messages that they should be relaying (e.g., messages broadcast by processes that crashed), and hence might violate *agreement.* If the *accuracy* property of the failure detector is not satisfied, then the processes might be relaying messages when it is not really necessary. This wastes resources but does not impact correctness.

In fact, we can circumvent the need for a failure detector (*completeness*) property as well by adopting an *eager* scheme: every process that gets a message relays it immediately. That is, we consider the worst case where the sender process might have crashed and we relay every message. This relaying phase is exactly what guarantees the *agreement* property of reliable broadcast. The resulting algorith, called Eager Reliable Broadcast, is depicted in Algorithm 3.3.

Module:

Name: UniformReliableBroadcast (urb).

Events:

$\langle$ *urbBroadcast*, m $\rangle$, $\langle$ *urbDeliver*, src, m $\rangle$, with the same meaning and interface as in regular reliable broadcast.

Properties:

RB1-RB3: Same as in regular reliable broadcast.

URB4: *Uniform Agreement:* If a message m is delivered by some process p_i (whether correct or faulty), then m is also eventually delivered by every other correct process p_j.

Module 3.3 Interface and properties of uniform reliable broadcast.

Algorithm 3.3 is in this sense eager but fail-silent: it makes use only of the best-effort primitive described in Section 3.2 (and no failure detector abstraction). In Figure 3.2a we illustrate how the algorithm ensures *agreement* even if the sender crashes: process p_1 crashes and its message is not bebDelivered by p_3 and p_4. However, since p_2 retransmits the message (bebBroadcasts it), the remaining processes also bebDeliver it and then rbDeliver it. In our first algorithm (the lazy one), p_2 will be relaying the message only after it has detected the crash of p_1.

Correctness. All properties, except *agreement*, are ensured as in the lazy reliable broadcast algorithm. The *agreement* property follows from the *validity* property of the underlying best effort broadcast primitive and from the fact that every process relays every message it rbDelivers.

Performance. In the best case, to rbDeliver a message to all processes, the algorithm requires a single communication step and N^2 messages. In the worst case, if processes crash in sequence, N steps and N^2 messages are required to terminate the algorithm.

3.4 Uniform Reliable Broadcast

With regular reliable broadcast, the semantics just require correct processes to deliver the same information, regardless of what messages have been delivered by faulty processes. The uniform definition is stronger in the sense that it guarantees that the set of messages delivered by faulty processes is always a sub-set of the messages delivered by correct processes.

3.4.1 Specification

Uniform reliable broadcast differs from reliable broadcast by the formulation of its *agreement* property. The specification is given in Module 3.3.

Uniformity is typically important if processes might interact with the external world, e.g., print something on a screen or trigger the delivery of money through an ATM. In this case, the fact that a process has delivered a message is important, even if the process has crashed afterwards. This is because the process, before crashing, could have communicated with the external world after having delivered the message. The processes that remain alive in the system (i.e., that did not crash) should also be aware of that message having been delivered.

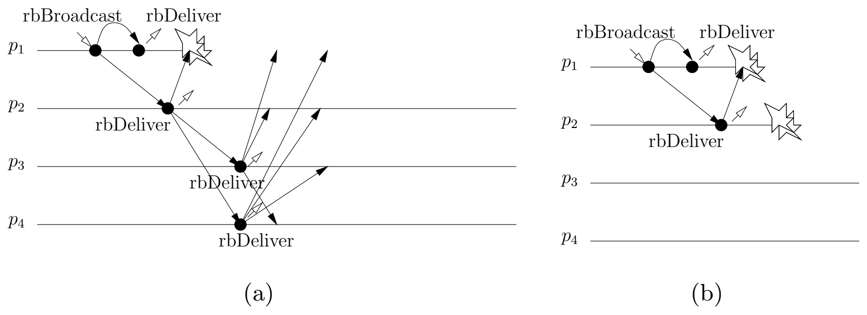

Figure 3.2. Sample executions of Eager Reliable Broadcast.

Figure 3.2b shows why our reliable broadcast algorithm does not ensure uniformity. Both processes p_1 and p_2 rbDeliver the message as soon as they bebDeliver it, but crash before relaying the message to the remaining processes. Still, processes p_3 and p_4 are consistent among themselves (none of them have rbDelivered the message).

3.4.2 Fail-Stop Algorithm: All-Ack URB

Basically, our lazy reliable broadcast algorithm does not ensure *uniform agreement* because a process may rbDeliver a message and then crash: even if this process has relayed its message to all (through a bebBroadcast primitive), the message might not reach any of the remaining processes. Note that even if we considered the same algorithm and replaced the best-effort broadcast with a reliable broadcast, we would still not implement a uniform broadcast abstraction. This is because a process delivers a message before relaying it to all.

Algorithm 3.4, named All-Ack Uniform Reliable Broadcast, implements the uniform version of reliable broadcast. Basically, in this algorithm, a process only delivers a message when it knows that the message has been *seen* by all correct processes. All processes relay the message once they have *seen*

Algorithm 3.4 All-Ack Uniform Reliable Broadcast.

```
Implements:
    UniformReliableBroadcast (urb).

Uses:
    BestEffortBroadcast (beb).
    PerfectFailureDetector ($\mathcal{P}$).

function canDeliver(m) returns boolean is
    return (correct $\subseteq$ ack$_m$) $\wedge$ (m $\notin$ delivered));

upon event ⟨ Init ⟩ do
    delivered := $\emptyset$;
    pending := $\emptyset$;
    correct := $\Pi$;
    forall $m$ do ack$_m$ := $\emptyset$;

upon event ⟨ urbBroadcast, m ⟩ do
    pending := pending $\cup$ { (self, m) }
    trigger ⟨ bebBroadcast, [DATA, self, m] ⟩;

upon event ⟨ bebDeliver, $p_i$, [DATA, $s_m$, m] ⟩ do
    ack$_m$ := ack$_m$ $\cup$ $\{p_i\}$
    if ($s_m$, m) $\notin$ forward do
        forward := forward $\cup$ { ($s_m$, m) };
        trigger ⟨ bebBroadcast, [DATA, $s_m$, m] ⟩;

upon event ⟨ crash, $p_i$ ⟩ do
    correct := correct $\setminus\{p_i\}$;

upon exists $(s_m, m) \in$ pending such that canDeliver($m$) do
    delivered := delivered $\cup\{m\}$;
    trigger ⟨ urbDeliver, $s_m$, $m$ ⟩;
```

it. Each process keeps a record of which processes have already retransmitted a given message. When all correct processes retransmitted the message, all correct processes are guaranteed to deliver the message, as illustrated in Figure 3.3.

Correctness. As before, except for uniform agreement, all properties are trivially derived from the properties of the best-effort broadcast. (We also rely for *validity* on the *completeness* property of the failure detector). *Uniform agreement* is ensured by having each process wait to urbDeliver a message until all correct processes have bebDelivered the message. We rely here on the *accuracy* property of the perfect failure detector.

Performance. In the best case the algorithm requires two communication steps to deliver the message to all processes. In the worst case, if processes crash in sequence, $N + 1$ steps are required to terminate the algorithm. The algorithm exchanges N^2 messages in each step. Therefore, uniform reliable

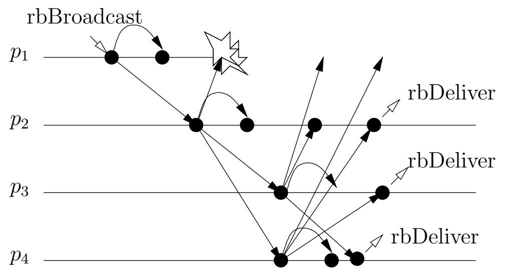

Figure 3.3. Sample execution of All-Ack Uniform Reliable Broadcast.

Algorithm 3.5 Majority-ack uniform reliable broadcast.

```
Implements:
    UniformReliableBroadcast (urb).

Uses:
    BestEffortBroadcast (beb).

function canDeliver(m) returns boolean is
    return ($|\text{ack}_m| > N/2$) $\wedge$ ($m \notin$ delivered);
// Except for the function above, and the non-use of the
// perfect failure detector, same as Algorithm 3.4.
```

broadcast requires one step more to deliver a message than its regular counterpart.

3.4.3 Fail-Silent Algorithm: Majority-Ack URB

The uniform algorithm of Section 3.4.2 (i.e., Algorithm 3.4) is not correct if the failure detector is not perfect. *Uniform agreement* would be violated if *accuracy* is not satisfied and *validity* would be violated if *completeness* is not satisfied.

We give in the following a uniform reliable broadcast algorithm that does not rely on a perfect failure detector but assumes a majority of correct processes. We leave it as an exercise to show why the majority assumption is needed in a fail-silent model.

In the example above of Figure 3.2, the correct majority assumption means that at most one process can crash in any given execution. Algorithm 3.5 is similar to the previous uniform reliable broadcast algorithm except that processes do not wait until all correct processes have seen a message (bebDelivered a copy of the message), but until a majority has seen the message.

Correctness. The *no-creation* property follows from the *no-creation* property of best-effort broadcast. The *no-duplication* property follows from the use of the variable *delivered* which prevents processes from delivering twice the same message. To discuss the *uniform agreement* and *validity* properties, we first argue that if a correct process p_i bebDelivers any message m, then p_i urbDelivers m. Indeed, if p_i is correct, and given that p_i bebBroadcasts m, every correct process bebDelivers and hence bebBroadcasts m. As we assume a correct majority, then p_i bebDelivers m from a majority of processes and urbDelivers m. Consider now the *validity* property: if a correct process p_i urbBroadcasts a message m, then p_i bebBroadcasts and hence p_i bebDelivers m: by the argument above, p_i eventually urbDelivers m. Consider now *uniform agreement* and let p_j be some process that urbDelivers m. To do so, p_j must have bebDelivered m from a majority of processes. By the assumption of a correct majority, at least one correct must have bebBroadcast m. Therefore, all correct processes have bebDelivered m, which implies that all correct processes eventually urbDeliver m.

Performance. Similar to the algorithm of Section 3.2.

3.5 Stubborn Broadcast

We now consider broadcast abstractions in a setting where processes can crash and recover, i.e., in the fail-recovery model. We first discuss the issue underlying fail-recovery when broadcasting messages and then we give examples of specifications and underlying algorithms in this model.

3.5.1 Overview

It is first important to notice why the specifications we have considered for the fail-stop and fail-silent models are not really adequate for the fail-recovery model. Indeed, we argue that even the strongest of our specifications, uniform reliable broadcast, does not provide useful semantics in a setting where processes that crash can recover (are not excluded from the computation).

Consider a message m that is broadcast by some process p_i. Consider furthermore some other process p_j that crashes at some instant, recovers, and never crashes again. In the fail-recovery sense, process p_j is correct. With the semantics of uniform reliable broadcast however, it might happen that p_j delivers m, crashes without having processed m, and then recovers with no memory about m. Ideally, there should be some way for process p_j to find out about m upon recovery, and hence to be able to execute any associated action accordingly.

We start by presenting a generalization of the stubborn point-to-point communication idea to the broadcast situation. Correct processes are supposed to deliver all messages (broadcast by processes that did not crash) an infinite number of times, and hence eventually deliver such messages upon recovery. The corresponding specification is called *stubborn broadcast*.

Module:

Name: StubbornBestEffortBroadcast (sbeb).

Events:

Request: ⟨ *sbebBroadcast*, m ⟩: Used to broadcast message m to all processes.

Indication: ⟨ *sbebDeliver*, m ⟩: Used to deliver message m.

Properties:

SBEB1: *Best-effort validity:* If p_j is correct and p_i does not crash, then every message broadcast by p_i is delivered by p_j an infinite number of times.

SBEB2: *No creation:* If a message m is delivered by some process p_j, then m was previously broadcast by some process p_i.

Module 3.4 Interface and properties of stubborn best-effort broadcast.

3.5.2 Specification

The specification of stubborn broadcast we consider is given in Module 3.4 and we focus here on the best-effort case. Stronger abstractions (regular and uniform) can be easily obtained accordingly. The key difference with the best-effort abstraction defined for the fail no-recovery settings is in the stubborn delivery of every message broadcast, as long as the process which has broadcast that message did not crash. Note also that the *no duplication* property is not ensured. In fact, the very fact that processes have now to deal with multiple deliveries is the price to pay for saving expensive logging operations.

3.5.3 Fail-Recovery Algorithm: Basic Stubborn Broadcast

We now present an algorithm that implements stubborn best-effort broadcast. Algorithm 3.6 is called basic stubborn broadcast and is straightforward using underlying stubborn communication links.

Correctness. The properties are derived from the properties of stubborn links. In particular, *validity* is derived from the fact that the sender sends the message to every other process in the system.

Performance. The algorithm requires a single communication step for a process to deliver a message and exchanges at least N messages. Of course, stubborn channels may retransmit the same message several times and, in practice, an optimization mechanism is needed to acknowledge the messages and stop the retransmission. Additionally, the algorithms requires a log operation for each delivered message.

Algorithm 3.6 Basic Stubborn Broadcast.

```
Implements:
    StubbornBestEffortBroadcast (sbeb).

Uses:
    StubbornPointToPointLink (sp2p).

upon event ⟨ sbebBroadcast, m ⟩ do
    forall $p_i \in \Pi$ do
        trigger ⟨ sp2pSend, $p_i, m$ ⟩;

upon event ⟨ sp2pDeliver, $p_i, m$ ⟩ do
        trigger ⟨ sbebDeliver, delivered ⟩;
```

3.6 Logged Best Effort Broadcast

We now extende Stubborn Broadcast to prevent multiple delivery of the same messages. In order to achieve this goal, we defining the semantics of message delivery according to message logging. Roughly speaking, a process is said to deliver a message when it logs the message, i.e., it stores it in stable storage. Hence, if it has delivered a message m, a process that crashes and recovers will still be able to retrieve m from stable storage and will be able to execute any associated action accordingly. The corresponding specification is called *logged broadcast*.

3.6.1 Specification

The abstraction we consider here is called *logged broadcast*, to emphasize the fact that the act of "delivering" corresponds to its logging in a local stable storage. The key difference with the best-effort abstraction defined for the fail no-recovery settings is in the interface between modules. Instead of simply triggering an event to "deliver" a message, logged broadcast relies on storing the message in a local log, which can later be read by the layer above: the layer is notified about changes in the log through specific events.

The specification is given in Module 3.5. The act of delivering the message corresponds here to the act of logging the variable *delivered* with m in that variable. Hence, *validity, no duplication* and *no creation* properties are redefined in term of log operations. Note also that we consider here the best-effort case: as we discuss later, stronger abstractions (regular and uniform) can then be designed and implemented on top of this one.

3.6.2 Fail-Recovery Algorithm: Logged Basic Broadcast

Algorithm 3.7, called "Logged Basic Broadcast", implements logged best-effort broadcast. It has many similarities, in its structure, with Algorithm 3.1 ("Basic Broadcsat"). The main differences are the following.

Module:

Name: LoggedBestEffortBroadcast (log-beb).

Events:

Request: ⟨ *log-bebBroadcast*, m ⟩: Used to broadcast message m to all processes.

Indication: ⟨ *log-bebDeliver*, delivered ⟩: Used to notify the upper level of potential updates to the delivered log.

Properties:

LBEB1: *Best-effort validity:* If p_j is correct and p_i does not crash, then every message broadcast by p_i is eventually delivered by p_j.

LBEB2: *No duplication:* No message is delivered more than once.

LBEB3: *No creation:* If a message m is delivered by some process p_j, then m was previously broadcast by some process p_i.

Module 3.5 Interface and properties of logged best-effort broadcast.

1. The algorithm makes use of stubborn communication links between every pair of processes. Remember that these ensure in particular that a message that is sent by a process that does not crash to a correct recipient is supposed to be delivered by its recipient an infinite number of times.
2. The algorithm maintains a log of all delivered messages. When a new message is received for the first time, it is appended to the log (delivered) and the upper layer is notified that the log has changed. If the process crashes and later recovers, the upper layer is also notified (as it may have missed a notification triggered just before the crash).

Correctness. The properties are derived from the properties of stubborn links. In particular, *validity* is derived from the fact that the sender sends the message to every other process in the system. *No duplication* is derived from the fact that the delivery log is checked before delivering new messages.

Performance. The algorithm requires a single communication step for a process to deliver a message and exchanges at least N messages. Of course, stubborn channels may retransmit the same message several times and, in practice, an optimization mechanism is needed to acknowledge the messages and stop the retransmission. Additionally, the algorithms requires a log operation for each delivered message.

3.7 Logged Uniform Reliable Broadcast

In a manner similar to the crash no-recovery case, it is possible to define both reliable and uniform variants of best-effort broadcast for the fail-recovery setting.

Algorithm 3.7 Logged Basic Broadcast.

```
Implements:
    LoggedBest-EffortBroadcast (log-beb).

Uses:
    StubbornPointToPointLink (sp2p).

upon event ⟨ Init ⟩ do
    delivered := ∅;
    store (delivered);

upon event ⟨ Recovery ⟩ do
    retrieve (delivered)
    trigger ⟨ log-bebDeliver, delivered ⟩;

upon event ⟨ log-bebBroadcast, m ⟩ do
    forall $p_i \in \Pi$ do
        trigger ⟨ sp2pSend, $p_i$, m ⟩;

upon event ⟨ sp2pDeliver, $p_i$, m ⟩ do
    if m ∉ delivered then
        delivered := delivered ∪ { m };
        store (delivered);
        trigger ⟨ log-bebDeliver, delivered ⟩;
```

3.7.1 Specification

Module 3.6 defines a logged variant of the uniform reliable broadcast for the fail-recovery model. In this variant, if a process (either correct or not) delivers a message (i.e., logs the variable *delivered* with the message in it), all correct processes should eventually deliver that message (i.e., log it in their variable *delivered*). Not surprisingly, the interface is similar to that of logged best-effort broadcast.

3.7.2 Fail-Recovery Algorithm: Logged Majority-Ack URB

Algorithm 3.8, called Logged "Majority-Ack URB", implements logged uniform broadcast assuming a majority of correct processes. The act of delivering (log-urbDeliver) a message m corresponds to logging the variable *delivered* with m in that variable. Besides *delivered*, the algorithm uses two other variables: *pending* and ack_m. The *pending* set gathers the messages that have been seen by a process but still need to be log-urbDelivered. This variable is logged. The ack_m set gathers, at each process p_i, the set of processes that p_i knows have seen m. The ack_m set is not logged: it can be reconstructed upon recovery. Message are only appended to the *delivered* log when they have been retransmitted by a majority of processes. This, together with the assumption

Module:

Name: LoggedUniformReliableBroadcast (log-urb).

Events:

⟨ *log-urbBroadcast*, m ⟩, ⟨ *log-urbDeliver*, delivered ⟩ with the same meaning and interface as in logged best-effort broadcast.

Properties:

LURB1: *Validity:* If p_j is correct and p_i does not crash, then every message broadcast by p_i is eventually delivered by p_j.

LURB2: *No duplication:* No message is delivered more than once.

LURB3: *No creation:* If a message m is delivered by some process p_j, then m was previously broadcast by some process p_i.

LURB4: *Uniform Agreement:* If a message m is delivered by some process, then m is eventually delivered by every correct process.

Module 3.6 Interface and properties of logged uniform reliable broadcast.

of a correct majority, ensures that at least one correct process has logged the message and will ensure the retransmission to all correct processes.

Correctness. Consider the *agreement* property and assume some process p_i delivers (log-urbDelivers) a message m and does not crash. To do so, a majority of the processes must have retransmitted the message. As we assume a majority of correct processes, at least one correct process must have logged the message (in *pending*). This process will ensure the eventual transmission (sp2pSend) of the message to all correct processes and all correct processes will hence acknowledge the message. Hence, every correct will deliver (log-urbDeliver) m. Consider the *validity* property and assume some process p_i broadcasts (log-urbBroadcasts) a message m and does not crash. Eventually, the message will be seen by all correct processes. As a majority is correct, a majority will retransmit the message: p_i will eventually log-urbDeliver m. The *no duplication* property is trivially ensured by the algorithm whereas the *no creation* property is ensured by the underlying channels.

Performance. Let m be any message that is broadcast (log-urbBroadcast) by some process p_i. A process delivers the message (log-urbDeliver) m after two communication steps and two causally related logging operations. (The logging of *pending* at a majority can be done in parallel).

3.8 Randomized Broadcast

This section considers randomized broadcast algorithms. These algorithms do not provide *deterministic* broadcast guarantees but, instead, only make probabilistic claims about such guarantees, for instance, by ensuring that the guarantees are provided successfully 99% of the times.

Algorithm 3.8 Logged Majority-Ack Uniform reliable Broadcast.

```
Implements:
    LoggedUniformReliableBroadcast (log-urb).

Uses:
    StubbornPointToPointLink (sp2p).

upon event ⟨ Init ⟩ do
    forall $m$ do ack$_m$ := $\emptyset$;
    pending := deliver := $\emptyset$;
    store (pending, delivered);

upon event ⟨ Recovery ⟩ do
    retrieve (pending, delivered);
    trigger ⟨ log-rbDeliver, delivered ⟩;
    forall $m \in$ pending do
        forall $p_i \in \Pi$ do
            trigger ⟨ sp2pSend, $p_i, m$ ⟩;

upon event ⟨ log-urbBroadcast, m ⟩ do
    pending := pending $\cup$ $\{m\}$;
    store (pending);
    ack$_m$ := ack$_m$ $\cup$ self;
    forall $p_i \in \Pi$ do
        trigger ⟨ sp2pSend, $p_i, m$ ⟩;

upon event ⟨ sp2pDeliver, $p_i, m$ ⟩ do
    if $m \notin$ pending then
        pending := pending $\cup$ $\{m\}$;
        store (pending);
        forall $p_i \in \Pi$ do
            trigger ⟨ sp2pSend, $p_i, m$ ⟩;
    if $p_i \notin$ ack$_m$ then
        ack$_m$ := ack$_m$ $\cup$ $\{p_i\}$;
        if | ack$_m$ | $> N/2$ then
            delivered := delivered $\cup$ $\{m\}$;
            store (delivered);
            trigger ⟨ log-urbDeliver, delivered ⟩;
```

Of course, this approach can only be applied to applications that do not require full reliability. On the other hand, as it will be seen, it is often possible to build probabilistic systems with good scalability properties.

3.8.1 The Scalability of Reliable Broadcast

As we have seen throughout this chapter, in order to ensure the reliability of broadcast in the presence of faulty processes (and/or links with omission failures), one needs to collect some form of *acknowledgments*. However, given

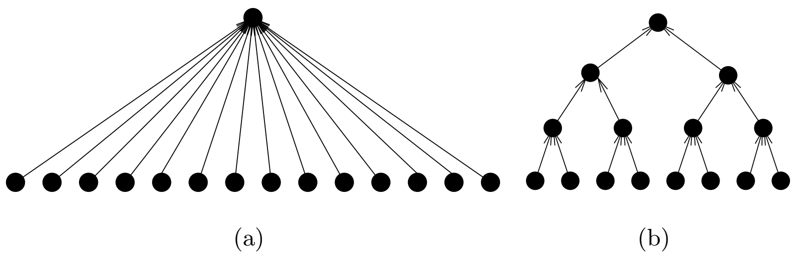

Figure 3.4. Ack implosion and ack tree.

limited bandwidth, memory and processor resources, there will always be a limit to the number of acknowledgments that each process is able to collect and compute in due time. If the group of processes becomes very large (say thousand or even millions of members in the group), a process collecting acknowledgments becomes overwhelmed by that task. This phenomena is known as the *ack implosion* problem (see Figure 3.4a).

There are several ways of mitigating the ack implosion problem. One way is to use some form of hierarchical scheme to collect acknowledgments, for instance, arranging the processes in a binary tree, as illustrated in Figure 3.4b. Hierarchies can reduce the load of each process but increase the latency in the task of collecting acknowledgments. Additionally, hierarchies need to be reconfigured when faults occur (which may not be a trivial task). Furthermore, even with this sort of hierarchies, the obligation to receive, directly or indirectly, an acknowledgment from every other process remains a fundamental scalability problem of reliable broadcast. In the next section we discuss how probabilistic approaches can circumvent this limitation.

3.8.2 Epidemic Dissemination

Nature gives us several examples of how a probabilistic approach can achieve a fast and efficient broadcast primitive. Consider how epidemics are spread among a population: initially, a single individual is infected; this individual in turn will infect some other individuals; after some period, the whole population is infected. Rumor spreading is based exactly on the same sort of mechanism and has shown to be a very effective way to disseminate information.

A number of broadcast algorithms have been designed based on this principle and, not surprisingly, these are often called *epidemic* or *rumor mongering* algorithms.

Before giving more details on these algorithms, we first define the abstraction that these algorithms implement. Obviously, this abstraction is not the reliable broadcast that we have introduced earlier.

Module:

Name: ProbabilisticBroadcast (pb).

Events:

Request: ⟨ *pbBroadcast*, m ⟩: Used to broadcast message m to all processes.

Indication: ⟨ *pbDeliver*, src, m ⟩: Used to deliver message m broadcast by process src.

Properties:

PB1: *Probabilistic validity:* There is a given probability such that for any p_i and p_j that are correct, every message broadcast by p_i is eventually delivered by p_j with this probability.

PB2: *No duplication:* No message is delivered more than once.

PB3: *No creation:* If a message m is delivered by some process p_j, then m was previously broadcast by some process p_i.

Module 3.7 Interface and properties of probabilistic broadcast.

3.8.3 Specification

Probabilistic broadcast is characterized by the properties PB1-3 depicted in Module 3.7.

Note that it is assumed that broadcast messages are implicitly addressed to all processes in the system, i.e., the goal of the sender is to have its message delivered at all processes. Note also that only *validity* is probabilistic.

The reader may find similarities between the specification of probabilistic broadcast and the specification of best-effort broadcast presented in Section 3.2. In fact, both are probabilistic approaches. However, in best-effort broadcast the probability of delivery depends directly on the reliability of the processes: it is in this sense hidden under the probability of process failures. In probabilistic broadcast, it becomes a first class citizen of the specification. The corresponding algorithms are devised with inherent redundancy to mask process faults and ensure delivery with the desired probability.

3.8.4 Randomized Algorithm: Eager Probabilistic Broadcast

Algorithm 3.9, called "Eager Probabilistic Broadcast", implements probabilistic broadcast. The sender selects k processes at random and sends them the message. In turn, each of these processes selects another k processes at random and forwards the message to those processes. Note that, in this algorithm, some or all of these processes may be exactly the same processes already selected by the initial sender.

A step consisting of receiving and gossiping a message is called a *round*. The algorithm performs a maximum number of rounds r for each message.

Algorithm 3.9 Eager Probabilistic Broadcast.

```
Implements:
    ProbabilisticBroadcast (pb).

Uses:
    unreliablePointToPointLinks (up2p).

upon event ⟨ Init ⟩ do
    delivered := $\emptyset$;

function pick-targets (fanout) returns set of processes do
    targets := $\emptyset$;
    while | targets | < fanout do
        candidate := random ($\Pi$);
        if candidate $\notin$ targets $\wedge$ candidate $\neq$ self then
            targets := targets $\cup$ { candidate };
    return targets;

procedure gossip (msg) do
    forall t $\in$ pick-targets (fanout) do
        trigger ⟨ up2pSend, t, msg ⟩;

upon event ⟨ pbBroadcast, m ⟩ do
    gossip ([Gossip, $s_m$, $m$, maxrounds$-$1]);

upon event ⟨ up2pDeliver, $p_i$, [Gossip, $s_m$, $m$, $r$] ⟩ do
    if $m \notin$ delivered then
        delivered := delivered $\cup$ {$m$}
        trigger ⟨ pbDeliver, $s_m, m$ ⟩;
    if $r > 0$ then gossip ([Gossip, $s_m$, $m$, $r - 1$]);
```

The reader should observe here that k, also called the *fanout*, is a fundamental parameter of the algorithm. Its choice directly impacts the probability of reliable message delivery guaranteed by the algorithm. A higher value of k will not only increase the probability of having all the population infected but also will decrease the number of rounds required to have all the population infected. Note also that the algorithm induces a significant amount of redundancy in the message exchanges: any given process may receive the same message more than once. The execution of the algorithm is for instance illustrated in Figure 3.5 for a configuration with a fanout of 3.

The higher the fanout, the higher the load that is imposed on each processes and the amount of redundant information exchanged in the network. Therefore, to select the appropriate k is of particular importance. The reader should also note that there are runs of the algorithm where a transmitted message may not be delivered to all correct processes. For instance, all the k processes that receive the message directly from the sender may select exactly the same k processes to forward the message to. In such case, only these k

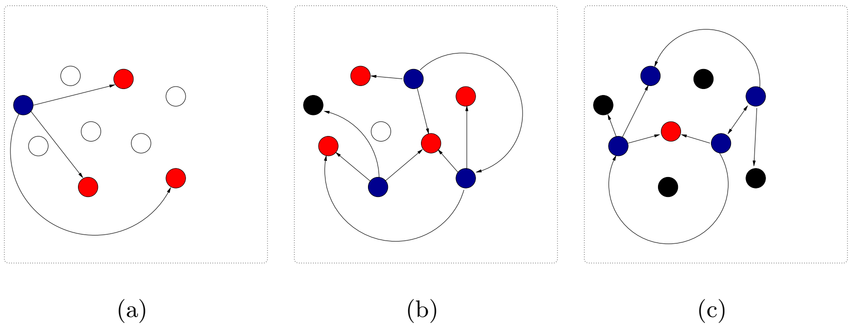

Figure 3.5. Gossip Dissemination.

processes will receive the message. This translates into the very fact that the probability of reliable delivery is not 100%.

It can be shown that, to ensure a high probability of delivering a message to all correct processes, it is enough for the fanout to be in the order of $\log N$, where N is the number of processes in the system. Naturally, the exact value of the fanout and maximum number of rounds to achieve a given probability of success depends not only on the system size but also on the probability of link and process failures.

3.8.5 Randomized Algorithm: Lazy Probabilistic Broadcast

The algorithm described above uses an epidemic approach to the dissemination of messages. However, and as we have discussed, a disadvantage of this approach is that it consumes a non-negligible amount of resources with redundant transmissions. A way to overcome this limitation is to rely on a basic and efficient unreliable communication primitives to disseminate the messages first, and then use a probabilistic approach just as a backup to recover from message omissions. More precisely, we assume here the existence of unreliable point-to-point and broadcast communication abstractions, defined by the primitives *unp2pSend, unp2p2Deliver* and *unBroadcast, unDeliver* respectively. We do not make specific assumptions on these, expect that they could be used to exchange messages efficiently, without corrupting or adding messages to the system, and with *some* reliable delivery guarantees. These could typically correspond to a fair-lossy communication primitive and a broadcast abstraction built on top of it.

A simplified version of an algorithm based on this idea is given in Algorithm 3.10, called “Lazy Probabilistic Broadcast”. The algorithm assumes that each sender is transmitting a stream of numbered messages. Message omissions are detected based on gaps on the sequence numbers of received

Algorithm 3.10 Lazy Probabilistic Broadcast.

```
Implements:
    ProbabilisticBroadcast (pb).

Uses:
    UnreliablePointToPointLinks (up2p), UnreliableBroadcast (unb).

upon event ⟨ Init ⟩ do
    forall $p_i \in \Pi$ do delivered[$p_i$] := 0;
    lsn := 0; pending := $\emptyset$; stored := $\emptyset$;

procedure deliver-pending ($s$) do
    forall [DATA, $s$, $x$, $sn_x$] ∈ pending such that $sn_x$ = delivered[$s$]+1 do
        delivered[$s$] := delivered[$s$]+1; pending := pending \ { [DATA, $s$, $x$,$sn_x$]};
        trigger ⟨ pbDeliver, $s, x$ ⟩;

procedure gossip (msg) do
    forall t ∈ pick-targets (fanout) do
        trigger ⟨ up2pSend, t, msg ⟩;

upon event ⟨ pbBroadcast, m ⟩ do
    lsn := lsn+1;
    trigger ⟨ unBroadcast, [DATA, self, $m$, $lsn$] ⟩;

upon event ⟨ unDeliver, $p_i$, [DATA, $s_m$, $m$, $sn_m$] ⟩ do
    if random() > store-threshold then stored := stored ∪ { [DATA, $s_m$, $m$,$sn_m$] };
    if $sn_m$ = delivered[$s_m$]+1 then
        delivered[$s_m$] := delivered[$s_m$]+1;
        trigger ⟨ pbDeliver, $s_m, m$ ⟩;
    else
        pending := pending ∪ { [DATA, $s_m$, $m$, $sn_m$] };
        forall seqnb ∈ [$sn_m - 1$, delivered[$s_m$] + 1] do
            gossip ([REQUEST, self, $s_m$, seqnb, maxrounds−1]);

upon event ⟨ up2pDeliver, $p_j$, [REQUEST, $p_i$, $s_m$, $sn_m$, $r$] ⟩ do
    if [DATA, $s_m$, $m$, $sn_m$] ∈ stored then
        trigger ⟨ up2pSend, $p_i$, [DATA, $s_m$, $m$, $sn_m$] ⟩;
    else if $r > 0$ then gossip ([REQUEST, $p_i$, $s_m$, $sn_m$, $r - 1$]);

upon event ⟨ up2pDeliver, $p_j$, [DATA, $s_m$, $m$, $sn_m$] ⟩ do
    if $sn_m$ = delivered[$s_m$]+1 then
        delivered[$s_m$] := delivered[$s_m$]+1;
        trigger ⟨ pbDeliver, $s_m, m$ ⟩;
        deliver-pending ($s_m$);
    else
        pending := pending ∪ { [DATA, $s_m$, $m$, $sn_m$] };
```

messages. Each message is disseminated using the unreliable broadcast primitive. For each message, some randomly selected receivers are chosen to store a copy of the message for future retransmission: they store the message for some maximum amount of time. The purpose of this approach is to distribute, among all processes, the load of storing messages for future retransmission.

Omissions can be detected using sequence numbers associated with messages. A process p detects that it has missed a message from a process q when p receives a message from q with a higher timestamp than what p was expecting from q. When a process detects an omission, it uses the gossip algorithm to disseminate a retransmission request. If the request is received by one of the processes that has stored a copy of the message, this process will retransmit the message. Note that, in this case, the gossip algorithm *does not* need to be configured to ensure that the retransmission request reaches all processes: it is enough that it reaches, with high probability, one of the processes that has stored a copy of the missing message.

It is expected that, in most cases, the retransmission request message is much smaller that the original data message. Therefore this algorithm is much more resource effective than the eager probabilistic broadcast algorithm described earlier. On the other hand, it does require the availability of some unreliable but efficient means of communication and this may not be available in settings that include a very large number of processes spread all over the Internet.

Practical algorithms based on this principle make a significant effort to optimize the number and the location of nodes that store copies of each broadcast message. Not surprisingly, best results can be obtained if the physical network topology is taken into account: for instance, an omission in a link connecting a local area network (LAN) with the rest of the system affects all processes in that LAN. Thus, it is desirable to have a copy of the message in each LAN (to recover from local omissions) and a copy outside the LAN (to recover from the omission in the link to the LAN). Similarly, the search procedure, instead of being completely random, may search first for a copy in the local LAN and only after on more distant processes.

Hands-On

We now describe the implementation, in *Appia*, of several of the protocols introduced in this chapter.

Basic Broadcast

The communication stack used to illustrate the protocol is the following:

Application
Basic Broadcast
Perfect Point-to-Point Links

The implementation of this algorithm closely follows the algorithm of Algorithm 3.1 ("Basic Broadcast"). As shown in Listing 3.1, this protocol only handles three classes of events, namely the ProcessInitEvent, used to initialize the set of processes that participate in the broadcast (this event is triggered by the application after reading the configuration file), the ChannelInit event, that is automatically triggered by the runtime when a new channel is created, and the SendableEvent. This last event is associated with transmission requests (if the event flows in the stack downwards) or the reception of events from the layer below (if the event flows upwards). Note that the code in these listing has been simplified. In particular, all exception handling code was deleted from the listings for clarity (but is included in the real code distributed with the tutorial).

The only method that requires some coding is the bebBroadcast() method, which is in charge of sending a series of point-to-point messages to all members of the group. This is performed by executing the following instructions for each member of the group: *i)* the event being sent is "cloned" (this effectively copies the data to be sent to a new event); *ii)* the source and destination address of the point-to-point message are set; *iii)* the event is forwarded to the layer below. There is a single exception to this procedure: if the destination process is the sender itself, the event is immediately delivered to the upper layer. The method to process messages received from the the layer below is very simple: it just forwards the message up.

Listing 3.1. Basic Broadcast implementation.

```
public class BEBSession extends Session {

    private ProcessSet processes;

    public BEBSession(Layer layer) {
        super(layer);
    }


    public void handle(Event event){
        if(event instanceof ChannelInit)
            handleChannelInit((ChannelInit)event);
        else if(event instanceof ProcessInitEvent)
            handleProcessInitEvent((ProcessInitEvent) event);
        else if(event instanceof SendableEvent){
            if(event.getDir()==Direction.DOWN)
                // UPON event from the above protocol (or application)
                bebBroadcast((SendableEvent) event);
            else
                // UPON event from the bottom protocol (or perfect point2point links)
                pp2pDeliver((SendableEvent) event);
        }
    }

    private void handleProcessInitEvent(ProcessInitEvent event) {
        processes = event.getProcessSet();
        event.go();
    }

    private void handleChannelInit(ChannelInit init) {
        init.go();
    }

    private void bebBroadcast(SendableEvent event) {
        SampleProcess[] processArray = this.processes.getAllProcesses();
        SendableEvent sendingEvent = null;
        for(int i=0 ; i<processArray.length ; i++){
            // source and destination for data message
            sendingEvent = (SendableEvent) event.cloneEvent();
            sendingEvent.source = processes.getSelfProcess().getInetWithPort();
            sendingEvent.dest = processArray[i].getInetWithPort();
            // set the event fields
            sendingEvent.setSource(this); // the session that created the event
            if(i == processes.getSelfRank())
                sendingEvent.setDir(Direction.UP);
            sendingEvent.init();
            sendingEvent.go();
        }
    }

    private void pp2pDeliver(SendableEvent event) {
        event.go();
    }
}
```

Lazy Reliable Broadcast

The communication stack used to illustrate the protocol is the following:

Application
Reliable Broadcast
Perfect Failure Detector
Best Effort Broadcast
Perfect Point to Point Links

The implementation of this algorithm, shown in Listing 3.2, closely follows the Algorithm 3.2 ("Lazy Reliable Broadcast"). The protocol accepts four events, namely the ProcessInitEvent, used to initialize the set of processes that participate in the broadcast (this event is triggered by the application after reading the configuration file), the ChannelInit event, that is automatically triggered by the runtime when a new channel is created, the Crash event, triggered by the PFD when a node crashes, and the SendableEvent. This last event is associated with transmission requests (if the event flows in the stack downwards) or the reception of events from the layer below (if the event flows upwards). Note that the code in these listing has been simplified. In particular, all exception handling code was deleted from the listings for clarity (but is included in the real code distributed with the tutorial).

In order to detect duplicates, each message needs to be uniquely identified. In this implementation, the protocol use the rank of the sender of the message and a sequence number. This information needs to be pushed into the message header when a message is sent, and then popped again when the message is received. Note that during the retransmission phase, it is possible for the same message, with the same identifier, to be broadcast by different processes.

In the protocol, to broadcast a message consists only in pushing the message identifier and forward the request to the Best-Effort layer. To receive the message consists in popping the message identifier, check for duplicates, and to log and deliver the message when it is received for the first time. Upon a crash notification, all messages from the crashed node are broadcast again. Note that when a node receives a message for the first time, if the sender is already detected to be crashed, the message is immediately retransmitted.

Listing 3.2. Lazy reliable broadcast implementation.

```
public class RBSession extends Session {
    private ProcessSet processes;
    private int seqNumber;
    private LinkedList[] from;
    private LinkedList delivered;

    public RBSession(Layer layer) {
        super(layer);
        seqNumber = 0;
    }

    public void handle(Event event){
        // (...)
    }

    private void handleChannelInit(ChannelInit init) {
        init.go();
        delivered = new LinkedList();
    }

    private void handleProcessInitEvent(ProcessInitEvent event) {
        processes = event.getProcessSet();
        event.go();
        from = new LinkedList[processes.getSize()];
        for (int i=0; i<from.length; i++)
            from[i] = new LinkedList();
    }

    private void rbBroadcast(SendableEvent event) {
        SampleProcess self = processes.getSelfProcess();
        MessageID msgID = new MessageID(self.getProcessNumber(),seqNumber);
        seqNumber++;
        ((ExtendedMessage)event.getMessage()).pushObject(msgID);
        bebBroadcast(event);
    }

    private void bebDeliver(SendableEvent event) {
        MessageID msgID = (MessageID) ((ExtendedMessage)event.getMessage()).peekObject();
        if ( ! delivered.contains(msgID) ){
            delivered.add(msgID);
            SendableEvent cloned = (SendableEvent) event.cloneEvent();
            ((ExtendedMessage)event.getMessage()).popObject();
            event.go();
            SampleProcess pi = processes.getProcess((InetWithPort) event.source);
            int piNumber = pi.getProcessNumber();
            from[piNumber].add(event);
            if ( ! pi.isCorrect() ){
                SendableEvent retransmit = (SendableEvent) cloned.cloneEvent();
                bebBroadcast(retransmit);
            }
        }
    }

    private void bebBroadcast(SendableEvent event) {
        event.setDir(Direction.DOWN);
        event.setSource(this);
        event.init();
        event.go();
    }

    private void handleCrash(Crash crash) {
        int pi = crash.getCrashedProcess();
        System.out.println("Process␣"+pi+"␣failed.");
        processes.getProcess(pi).setCorrect(false);
        SendableEvent event = null;
        ListIterator it = from[pi].listIterator();
        while(it.hasNext()){
            event = (SendableEvent) it.next();
            bebBroadcast(event);
        }
    from[pi].clear();
    }
}
```

Hands-On Exercise

Exercise 3.1 *This implementation of the Reliable Broadcast Algorithm has a* delivered *set that is never garbage collected. Modify the implementation to remove messages that no longer need to be maintained in the* delivered *set.*

All-Ack URB

The communication stack used to illustrate the protocol is the following:

Application
Uniform Reliable Broadcast
Perfect Failure Detector
Best Effort Broadcast
Perfect Point to Point Links

The implementation of this protocol is shown in Listing 3.3. Note that the code in these listing has been simplified. In particular, all exception handling code was deleted from the listings for clarity (but is included in the real code distributed with the tutorial).

The protocol uses two variables received and delivered to register which messages have already been received and delivered respectively. These variables only store message identifiers. When a message is received for the first time, it is forwarded as specified in the algorithm. To keep track on who has already acknowledged (forwarded) a given message a hash table is used. There is an entry in the hash table for each message. This entry keeps the data message itself (for future delivery) and a record of who has forwarded the message.

When a message has been forwarded by every correct process it can be delivered. This is checked every time a new event is handled (as both the reception of messages and the crash of processes may trigger the delivery of pending messages).

Listing 3.3. All ack uniform reliable broadcast implementation.

```
public class URBSession extends Session {
  private ProcessSet processes;
  private int seqNumber;
  private LinkedList received, delivered;
  private Hashtable ack;


  public URBSession(Layer layer) {
    super(layer);
  }

  public void handle(Event event) {
    // (...)
    urbTryDeliver();
  }

  private void urbTryDeliver() {
    Iterator it = ack.values().iterator();
    MessageEntry entry=null;
    while( it.hasNext() ){
      entry = (MessageEntry) it.next();
      if(canDeliver(entry)){
        delivered.add(entry.messageID);
        urbDeliver(entry.event, entry.messageID.process);
      }
    }
  }

  private boolean canDeliver(MessageEntry entry) {
    int procSize = processes.getSize();
    for(int i=0; i<procSize; i++)
      if(processes.getProcess(i).isCorrect() && (! entry.acks[i]) )
        return false;
    return ( (! delivered.contains(entry.messageID)) && received.contains(entry.messageID) );
  }

  private void handleChannelInit(ChannelInit init) {
    init.go();
    received = new LinkedList();
    delivered = new LinkedList();
    ack = new Hashtable();
  }
  private void handleProcessInitEvent(ProcessInitEvent event) {
    processes = event.getProcessSet();
    event.go();
  }

  private void urbBroadcast(SendableEvent event) {
    SampleProcess self = processes.getSelfProcess();
    MessageID msgID = new MessageID(self.getProcessNumber(),seqNumber);
    seqNumber++;
    received.add(msgID);
    ((ExtendedMessage) event.getMessage()).pushObject(msgID);
    event.go ();
  }

  private void bebDeliver(SendableEvent event) {
    SendableEvent clone = (SendableEvent) event.cloneEvent();
    MessageID msgID = (MessageID) ((ExtendedMessage) clone.getMessage()).popObject();
    addAck(clone,msgID);
    if ( ! received.contains(msgID) ){
      received.add(msgID);
      bebBroadcast(event);
    }
  }

  private void bebBroadcast(SendableEvent event) {
    event.setDir(Direction.DOWN);
    event.setSource(this);
    event.init();
    event.go();
  }

  private void urbDeliver(SendableEvent event, int sender) {
    event.setDir(Direction.UP);
    event.setSource(this);
    event.source = processes.getProcess(sender).getInetWithPort();
    event.init();
    event.go();
  }

  private void handleCrash(Crash crash) {
    int crashedProcess = crash.getCrashedProcess();
    System.out.println("Process␣"+crashedProcess+"␣failed.");
    processes.getProcess(crashedProcess).setCorrect(false);
  }

  private void addAck(SendableEvent event, MessageID msgID){
    int pi = processes.getProcess((InetWithPort)event.source).getProcessNumber();
    MessageEntry entry = (MessageEntry) ack.get(msgID);
    if(entry == null){
      entry = new MessageEntry(event, msgID, processes.getSize());
      ack.put(msgID,entry);
    }
    entry.acks[pi] = true;
  }
}
```

Hands-On Exercises

Exercise 3.2 *Modify the implementation to keep track just of the last message sent from each process, in the* received *and* delivered *variables.*

Exercise 3.3 *Change the protocol to exchange acknowledgements when the sender is correct and only retransmit the payload of a message when the sender is detected to have crashed (just like in the Lazy Reliable Protocol).*

Majority-Ack URB

The communication stack used to illustrate the protocol is the following (note that a Perfect Failure Detector is no longer required):

Application
Indulgent Uniform Reliable Broadcast
Best Effort Broadcast
Perfect Point to Point Links

The protocol works in the same way as the protocol presented in the prevous section, but without being aware of crashed processes. Besides that, the only difference from the previous implementation is the `canDeliver()` method, which can be shown in Listing 3.4.

Listing 3.4. Indulgent Uniform reliable broadcast implementation.

```
public class IURBSession extends Session {

  private boolean canDeliver(MessageEntry entry) {
    int N = processes.getSize(), numAcks = 0;
    for(int i=0; i<N; i++)
      if(entry.acks[i])
        numAcks++;
    return (numAcks > (N/2)) && ( ! delivered.contains(entry.messageID) );
  }

  // Except for the method above, and for the handling of the crash event, same
        // as the previous protocol
}
```

Hand-On Exercises.

Exercise 3.4 *Note that if a process does not acknowledge a message, copies of that message may have to be stored for a long period (in fact, if a process crashes, copies need to be stored forever). Try to devise a scheme to ensure that no more than $N/2+1$ copies of each message are preserved in the system (that is, not all members should be required to keep a copy of every message).*

Probabilistic Reliable Broadcast

This protocol is based on probabilities and is used to broadcast messages in large groups. Instead of creating Perfect Point to Point Links, it use Unreliable Point to Point Links (UP2PL) to send messages just for a subset of the group. The communication stack used to illustrate the protocol is the following:

Application
Probabilistic Broadcast
Unreliable Point to Point Links

The protocol has two configurable parameters: *i) fanout* is the number of processes for which the message will be gossiped; *maxrounds*, is the maximum number of rounds that the message will be retransmitted.

The implementation of this protocol is shown on Listing 3.5. The gossip() method invokes the pickTargets() method to choose the processes which the message is going to be sent and sends the message to those targets. The pickTargets() method chooses targets randomly from the set of processes. Each message carries its identification (as previous reliable broadcast protocols) and the remaining number of rounds (when the message is gossiped again, the number of rounds is decremented).

Listing 3.5. Probabilistic broadcast implementation.

```
public class PBSession extends Session {

  private LinkedList delivered;
  private ProcessSet processes;
  private int fanout, maxRounds, seqNumber;

  public PBSession(Layer layer) {
    super(layer);
    PBLayer pbLayer = (PBLayer) layer;
    fanout = pbLayer.getFanout();
    maxRounds = pbLayer.getMaxRounds();
    seqNumber = 0;
  }

  public void handle(Event event){
    // (...)
  }

  private void handleChannelInit(ChannelInit init) {
    init.go();
    delivered = new LinkedList();
  }

  private void handleProcessInitEvent(ProcessInitEvent event) {
    processes = event.getProcessSet();
    fanout = Math.min (fanout, processes.getSize ());
    event.go();
  }

  private void pbBroadcast(SendableEvent event) {
    MessageID msgID = new MessageID(processes.getSelfRank(),seqNumber);
    seqNumber++;
    gossip(event, msgID, maxRounds−1);
  }

  private void up2pDeliver(SendableEvent event) {
    SampleProcess pi = processes.getProcess((InetWithPort)event.source);
    int round = ((ExtendedMessage) event.getMessage()).popInt();
    MessageID msgID = (MessageID) ((ExtendedMessage) event.getMessage()).popObject();
    if ( ! delivered.contains(msgID) ){
      delivered.add(msgID);
      SendableEvent clone = null;
      clone = (SendableEvent) event.cloneEvent();
      pbDeliver(clone,msgID);
    }
    if(round > 0)
      gossip(event,msgID,round−1);
  }

  private void gossip(SendableEvent event, MessageID msgID, int round){
      int[] targets = pickTargets();
      for(int i=0; i<fanout; i++){
        SendableEvent clone = (SendableEvent) event.cloneEvent();
        ((ExtendedMessage) clone.getMessage()).pushObject(msgID);
        ((ExtendedMessage) clone.getMessage()).pushInt(round);
        up2pSend(clone,targets[i]);
      }
  }

  private int[] pickTargets() {
    Random random = new Random(System.currentTimeMillis());
    LinkedList targets = new LinkedList();
    Integer candidate = null;
    while(targets.size() < fanout){
      candidate = new Integer(random.nextInt(processes.getSize()));
      if ( ( ! targets.contains(candidate) ) && (candidate.intValue() != processes.getSelfRank()) )
        targets.add(candidate);
    }
    int[] targetArray = new int[fanout];
    ListIterator it = targets.listIterator();
    for(int i=0; (i<targetArray.length) && it.hasNext(); i++)
      targetArray[i] = ((Integer)it.next()).intValue();
    return targetArray;
  }

  private void up2pSend(SendableEvent event, int dest) {
    event.setDir(Direction.DOWN);
    event.setSource(this);
    event.dest = processes.getProcess(dest).getInetWithPort();
    event.init();
    event.go();
  }

  private void pbDeliver(SendableEvent event, MessageID msgID) {
    event.setDir(Direction.UP);
    event.setSource(this);
    event.source = processes.getProcess(msgID.process).getInetWithPort();
    event.init();
    event.go();
  }
}
```

Hands-On Exercises

Exercise 3.5 *The up2pDeliver() method perform two different functions:* i) *delivers the message to the application (if it was not delivered yet) and* ii) *gossips the message to other processes. Change the code such that a node gossip just when it receives a message for the first time. Discuss the impact of the changes.*

Exercise 3.6 *Change the code to limit* i) *the number of messages each node can store;* ii) *the maximum throughput (messages per unit of time) of each node.*

Test Application

All the implementations covered by this chapter may be experimented using the same application, called SampleAppl. An optional parameter in the command line allows the user to select which protocol stack the application will use. The general format of the command line is the following:

```
java demo/tutorialDA/SampleAppl -f <cf> -n <rank> -qos <prot>
```

The `cf` parameter is the name of a text file with the information about the set of processes, namely the total number N of processes in the system and, for each of these processes, its "rank" and "endpoint". The rank is a unique logical identifier of each process (an integer from 0 to $N-1$). The "endpoint" is just the host name or IP address and the port number of the process. This information is used by low-level protocols (such as TCP or UDP) to establish the links among the processes. The configuration file has the following format:

```
<number_of_processes>
<rank> <host_name> <port>
...
<rank> <host_name> <port>
```

For example, the following configuration file could be used to define a group of three processes, all running in the local machine:

```
3
0 localhost 25000
1 localhost 25100
2 localhost 25200
```

The `rank` parameter identifies the rank of the process being launched (and implicitly, the address to be used by the process, taken from the configuration file).

The `prot` parameter specifies which broadcast abstraction is used by the application. One of the following values can be selected: *beb* to use Best Effort Broadcast protocol; *rb* to use Lazy Reliable Broadcast protocol; *urb* to use Uniform Reliable Broadcast protocol; *iurb* to use Indulgent Uniform Reliable Broadcast protocol; and *pb <fanout> <maxrounds>* to use the Probabilistic Broadcast protocol with a specified fanout and maximum number of message rounds.

After all processes are launched, a message can be sent from one process to the other processes just by typing a `bcast <string>` in the command line and pressing the `Enter` key. The application also accepts another command, the `startpfd` to activate the perfect failure detector (as decribed in the previous chapter); do not forget to initiate the PFD at every processes by issuing the `startpfd` request on the command line every time the stack uses this service.

Exercises

Exercise 3.7 *Consider a process p that rbBroadcasts a message m in our lazy reliable broadcast implementation (Algorithm 3.2). Can p rbDeliver m before bebBroadcasting it.*

Exercise 3.8 *Modify our lazy reliable broadcast algorithm (Algorithm 3.2) to reduce the number of messages sent in case of failures.*

Exercise 3.9 *Some of our algorithms have the processes continuously fill their different buffers without emptying them. Modify them to remove unnecessary messages from the following buffers:*

1. $from[p_i]$ *in the lazy reliable broadcast algorithm*
2. delivered *in all reliable broadcast algorithms*
3. pending *in the all-ack uniform reliable broadcast algorithm*

Exercise 3.10 *What do we gain if we replace bebBroadcast with rbBroadcast in our majority-ack uniform reliable broadcast algorithm?*

Exercise 3.11 *Consider our lazy reliable broadcast and all-ack uniform broadcast algorithms: both use a perfect failure detector. What happens if each of the following properties of the failure detector is violated:*

1. *accuracy*
2. *completeness*

Exercise 3.12 *Our all-ack uniform reliable broadcast algorithm can be viewed as an extension of our eager reliable broadcast algorithm. Would we gain anything by devising a uniform reliable broadcast algorithm that would be an extension of our lazy reliable algorithm, i.e., can we have the processes not relay messages unless they suspect the sender?*

Exercise 3.13 *Can we devise a uniform reliable broadcast with an eventually perfect failure detector but without the assumption of a correct majority of processes?*

Exercise 3.14 *Give the specification of a logged reliable broadcast abstraction (i.e., a weaker variant Module 3.6) and an algorithm that implements it (i.e., a simpler variant of Algorithm 3.7).*

Exercise 3.15 *Our probabilistic broadcast algorithm considers that the connectivity is the same among every pair of processes. In practice, it may happen that some processes are at shorter distance and connected by more reliable links than others. For instance, the underlying network topology could be a set of local-area networks connected by long-haul links. Propose methods to exploit the topology in gossip algorithms.*

Algorithm 3.11 Simple optimization of lazy reliable broadcast.

```
upon event ⟨ rbBroadcast, m ⟩ do
    delivered := delivered ∪ {m}
    trigger ⟨ rbDeliver, self, m ⟩;
    trigger ⟨ bebBroadcast, [DATA, self, m] ⟩;
```

Solutions

Solution 3.7 The answer is yes. Every process anyway rbDelivers the messages as soon as it bebDelivers them. This does not add any guarantee with respect to rbDelivering the messages before bebBroadcasting them. The event that we would need to change to Algorithm 3.2 would simply be the rbBroadcast even, as depicted in Algorithm 3.11.
□

Solution 3.8 In our lazy reliable broadcast algorithm, if a process p rbBroadcasts a message and then crashes, N^2 messages are relayed by the remaining processes to retransmit the message of process p. This is because a process that bebDelivers the message of p does not know whether the other processes have bebDelivered this message or not. However, it would be sufficient in this case if only one process, for example process q, relays the message of p.

In practice one specific process, call it leader process p_l, might be more likely to bebDeliver messages: the links to and from this process would be fast and very reliable, the process runs on a reliable computer, etc. A process p_i would forward its messages to the leader p_l, which would coordinate the broadcast to every other process. If the leader is correct, every process will eventually bebDelivers and rbDelivers every message. Otherwise, we revert to the previous algorithm, and every process would be responsible of bebBroadcasting the messages that it bebDelivered. □

Solution 3.9 From $from[p_i]$ in the lazy reliable broadcast algorithm: The array $from$ is used exclusively to store messages that are retransmitted in the case of a failure. Therefore they can be removed as soon as they have been retransmitted. If p_i is correct, they will eventually be bebDelivered. If p_i is faulty, it does not matter if the other processes do not bebDeliver them.

From *delivered* in all reliable broadcast algorithms: Messages cannot be removed. If a process crashes and its messages are retransmitted by two different processes, then a process might rbDeliver the same message twice if it empties the *deliver* buffer in the meantime. This would violate the *no duplication* property.

From *pending* in the uniform reliable broadcast algorithm: Messages can actually be removed as soon as they have been urbDelivered. □

Solution 3.10 Nothing, because the uniform reliable broadcast algorithm does not assume and hence does not use the guarantees provided by the reliable broadcast algorithm.

Consider the following scenario which illustrates the difference between using bebBroadcast and using rbBroadcast. A process p broadcasts a message and crashes. Consider the case where only one correct process q receives the message (bebBroadcast). With rbBroadcast, all correct processes would deliver the message. In the urbBroadcast algorithm, q adds the message in $forward$ and then bebBroadcasts it. As q is correct, all correct processes will deliver it, and thus, we have at least the same guarantee as with rbBroadcast. □

Solution 3.11 Consider our uniform reliable broadcast algorithm using a perfect failure detector and a system of three processes: p_1, p_2 and p_3. Assume furthermore that p_1 urbBroadcasts a message m. If *strong completeness* is not satisfied, then p_1 might never urbDeliver m if any of p_2 or p_3 crash and p_1 never suspects them or bebDelivers m from them: p_1 would wait indefinitely for them to relay the message. In the cases of regular and uniform reliable broadcast algorithms, the *validity property* can be violated. Assume now that *strong accuracy* is violated and p_1 falsely suspects p_2 and p_3 to have crashed. Process p_1 eventually urbDelivers m. Assume that p_1 crashes afterwards. It might be the case that p_2 and p_3 never bebDelivered m and have no way of knowing about m and urbDeliver it: *uniform agreement* is violated. □

Solution 3.12 The advantage of the lazy scheme is that processes do not need to relay messages to ensure *agreement* if they do not suspect the sender to have crashed. In this failure-free scenario, only $N - 1$ messages are needed for all the processes to deliver a message. In the case of uniform reliable broadcast (without a majority), a process can only deliver a message when it knows that every correct process has seen that message. Hence, every process should somehow convey that fact, i.e., that it has seen the message. A lazy scheme would be of no benefit here. □

Solution 3.13 No. We explain why for the case of a system of four processes $\{p_1, p_2, p_3, p_4\}$ using what is called a *partitioning* argument. The fact that the correct majority assumption does not hold means that 2 out of the 4 processes may fail. Consider an execution where process p_1 broadcasts a message m and assume that p_3 and p_4 crash in that execution without receiving any message neither from p_1 nor from p_2. By the *validity* property of uniform reliable broadcast, there must be a time t at which p_1 urbDelivers message m. Consider now an execution that is similar to this one except that p_1 and p_2 crash right after time t whereas p_3 and p_4 are correct: say they have been falsely suspected, which is possible with an eventually perfect failure detector. In this execution, p_1 has urbDelivered a message m whereas p_3 and p_4

Module:

Name: Logged Reliable Broadcast (log-rb).

Events:

⟨ *log-rbBroadcast*, m ⟩, ⟨ *log-rbDeliver*, delivered ⟩ with the same meaning and interface as in logged best-effort broadcast.

Properties:

LURB1: *Validity:* If p_j is correct and p_i does not crash, then every message broadcast by p_i is eventually delivered by p_j.

LURB2: *No duplication:* No message is delivered more than once.

LURB3: *No creation:* If a message m is delivered by some process p_j, then m was previously broadcast by some process p_i.

LURB4: *Agreement:* If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

Module 3.8 Interface and properties of logged reliable broadcast.

have no way of knowing about that message m and eventually urbDelivering it: *uniform agreement* is violated. □

Solution 3.14

Module 3.6 defines a logged variant of reliable broadcast. In this variant, if a correct process delivers a message (i.e., logs the variable *delivered* with the message in it), all correct processes should eventually deliver that message (i.e., log it in their variable *delivered*).

Algorithm 3.12 implements logged reliable broadcast using stubborn channels. To broadcast a message, a process first delivers it and then sends it to all (using stubborn channels). When a message is received for the first time, it is delivered and sent to all. Upon recovery, a process retrieves the messages it has delivered and sends them to all.

Correctness. Consider the *agreement* property and assume some process correct p_i delivers a message m. If it does not crash, then p_i sends the message to all and all correct processes will deliver the message by the properties of the stubborn channels. If it crashes, there is a time after which p_i recovers, retrieves m and sends it to all. Again, all correct processes will deliver the message by the properties of the stubborn channels. The *validity* property follows directly from the stubborns channels. The *no duplication* property is trivially ensured by the algorithm whereas the *no creation* property is ensured by the underlying channels.

Performance. Let m be any message that is broadcast by some process p_i. A process delivers the message m immediately and the other processes deliver it after one communication step.

□

Algorithm 3.12 Reliable Broadcast with Log.

```
Implements:
    LoggedReliableBroadcast (log-rb).

Uses:
    StubbornPointToPointLink (sp2p).

upon event ⟨ Init ⟩ do
    delivered := $\emptyset$;
    store (delivered);

upon event ⟨ Recovery ⟩ do
    retrieve (delivered);
    trigger ⟨ log-rbDeliver, delivered ⟩;
    forall $m \in$ delivered do
        forall $p_i \in \Pi$ do
            trigger ⟨ sp2pSend, $p_i, m$ ⟩;

upon event ⟨ log-rbBroadcast, m ⟩ do
    delivered := delivered $\cup$ $\{m\}$;
    store (delivered);
    trigger ⟨ log-rbDeliver, delivered ⟩;
    forall $p_i \in \Pi$ do
        trigger ⟨ sp2pSend, $p_i, m$ ⟩;

upon event ⟨ sp2pDeliver, $p_i, m$ ⟩ do
    if $m \notin$ delivered then
        delivered := delivered $\cup$ $\{m\}$;
        store (delivered);
        trigger ⟨ log-rbDeliver, delivered ⟩;
        forall $p_i \in \Pi$ do
            trigger ⟨ sp2pSend, $p_i, m$ ⟩;
```

Solution 3.15 One approach consists in assigning weights to the links connecting processes. Weights reflect the reliability of the links. We could easily adapt our algorithm to avoid redundant transmission by gossiping though more reliable links with lower probability. An alternative approach consists in organizing the nodes in a hierarchy that reflects the network topology in order to reduce the traffic across domain boundaries. □

Historical Notes

- The requirements for a reliable broadcast communication abstraction seem to have originated from the domain of aircraft control and the Sift system in 1978 (Wensley 1978).
- Later on, several distributed computing libraries offered communication primitives with reliable broadcast semantics. These include the V System (Cherriton and Zwaenepoel 1985), Delta-4 (Powell, Barret, Bonn, Chereque, Seaton, and Verissimo 1994), Isis and Horus (Birman and Joseph 1987a; van Renesse and Maffeis 1996). Amoeba (Kaashoek, Tanenbaum, Hummel, and Bal 1989), Psync (Peterson, Bucholz, and Schlichting 1989), Transis (Amir, Dolev, Kramer, and Malki 1992), Totem (Moser, Melliar-Smith, Agarwal, Budhia, Lingley-Ppadopoulos, and Archambault 1995), xAMp (Rodrigues and Veríssimo 1992), OGS (Felber and Guerraoui 2000), Appia (Miranda, Pinto, and Rodrigues 2001), among others.
- Algorithms for reliable broadcast message delivery were presented in a very comprehensive way in 1994 (Hadzilacos and Toueg 1994). The problem of the uniformity of a broadcast was discussed in 1984 (Hadzilacos 1984) and then in 1993 (Neiger and Toueg 1993).
- The idea of applying epidemic dissemination to implement probabilistically reliable broadcast algorithms was explored in various papers (Golding and Long 1992; Birman, Hayden, Ozkasap, Xiao, Budiu, and Minsky 1999; Eugster, Handurukande, Guerraoui, Kermarrec, and Kouznetsov 2001; Kouznetsov, Guerraoui, Handurukande, and A.-M.Kermarrec 2001; Kermarrec, Massoulie, and Ganesh 2000; Xiao, Birman, and van Renesse 2002). The specification of probabilistic broadcast was defined later (Eugster, Guerraoui, and Kouznetsov 2004).
- The exploitation of topology features in probabilistic algorithms was proposed through an algorithm that assigns weights to link between processes (Lin and Marzullo 1999). A similar idea, but using a hierarchy instead of weight was proposed later to reduce the traffic across domain boundaries (Gupta, Kermarrec, and Ganesh 2002).
- The first probabilistic broadcast algorithm that did not depend of any global membership was given in 2001 (Eugster, Handurukande, Guerraoui, Kermarrec, and Kouznetsov 2001). The notion of message ages was also introduced in 2001 (Kouznetsov, Guerraoui, Handurukande, and A.-M.Kermarrec 2001) for purging messages and ensuring the scalability of process buffers. The idea of flow control in probabilistic broadcast was developed in 2002 (Rodrigues, Handurukande, Pereira, Guerraoui, and Kermarrec 2003).

4. Shared Memory

I always tell the truth, even when I lie.
(Tony Montana - Scarface)

This chapter presents shared memory abstractions. These are distributed programming abstractions that encapsulate read-write forms of storage among processes. These abstractions are called *registers* because they resemble those provided by multi-processor machines at the hardware level, though in many cases, including in this chapter, they are be implemented over processes that communicate through message passing and do not share any hardware device. help understand how to implement distributed shared working spaces.

We study here different variants of register abstractions. These differ according to the number of processes that are allowed to read and write on them, as well as on the semantics of their read operations in the face of concurrency and failures. We distinguish two kinds of semantics: *regular* and *atomic*. We will first consider the (1,N) *regular* register abstraction. The notation (1,N) means here that one specific process can write and any process can read. Then we will consider the (1,N) *atomic* register and finally the (N,N) atomic register abstractions. We will consider these abstractions for three of the distributed system models identified in Chapter 2: the fail-stop, fail-silent, and fail-recovery models.

4.1 Introduction

4.1.1 Sharing Information in a Distributed System

In a multiprocressor machine, processes typically communicate through registers provided at the hardware level. The set of these registers constitute the shared memory of the processes. The act of building a register abstraction among a set of processes that communicate by message passing is sometimes called a *shared memory emulation.* The programmer using this abstraction can develop shared memory algorithms without being aware that, behind the scenes, processes are actually communicating by exchanging messages, i.e., there is no physical shared memory. Such emulation is very appealing because programming with a shared memory is usually considered significantly easier than with message passing, precisely because the programmer can ignore various concurrency and failure issues.

Studying register specifications and algorithms is also useful in implementing distributed file systems as well as shared working spaces for collaborative work. For example, the abstraction of a distributed file that can be accessed through read and write operations is similar to the notion of register. Not surprisingly, the algorithms that one needs to devise in order to build a distributed file system can be directly inspired from those used to implement register abstractions. Similarly, when building a shared working space in collaborative editing environments, one ends up devising register-like distributed algorithms.

In the following, we will study two semantics of registers: *regular* and *atomic* ones. When describing a register abstraction, we will distinguish the case where it can be read and (or) written by exactly one process, and read and (or) written by all (i.e., any of the N processes in the system).

4.1.2 Register Overview

Assumptions. Registers store values that are accessed through two operations: *read* and *write*. The operations of a register are invoked by the processes of the system to exchange information through the register. When a process invokes any of these operations and gets back a reply, we say that the process completes the operation. Each process accesses the registers in a sequential manner: if a process invokes some operation (read or write on some register), the process does not invoke any further operation unless the previous one is complete.

To simplify, we also assume that every register (a) is supposed to contain only positive integers, and (b) is supposed to be initialized to 0. In other words, we assume through the latter assumption that some write operation was initially invoked on the register with 0 as a parameter and completed before any other operation is invoked. Also, for presentation simplicity but still without loss of generality, we will also assume that (c) the values written in the register are uniquely identified, say by using some unique timestamps provided by the processes. (Just like we assumed in the previous chapter that messages that are broadcast are uniquely identified.)

Some of the register abstractions and algorithms we will present make the assumption that specific processes can write and specific processes can read. For example, the simplest case is a register with one writer and one reader, denoted by $(1,1)$: the writer is a specific process known in advance and so is the reader. We will also consider registers with one writer and N readers (the writer is here a specific process and any process can be a reader). More generally, a register with X writers and Y readers is also called a (X, Y) register. The extreme case is of course the one with N writers and N readers: any process can be a writer and a reader at the same time.

Signature and Semantics. Basically, a read is supposed to return the value in the register and a write is supposed to update the value of the register. More precisely:

1. A read operation does not take any input parameter and has one ouput parameter. This output parameter contains the presumably current value of the register and constitutes the reply of the read invocation. A read does not modify the content of the register.

2. A write operation takes an input parameter and returns a simple indication (*ack*) that the operation has taken place. This indication constitutes the reply of the write invocation. The write operation aims at modifying the content of the register.

If a register is used (read and written) by a single process, and we assume there is no failure, it is reasonable to define the specification of the register through the simple following properties:

- **Liveness.** Every operation eventually completes.
- **Safety.** Every read returns the *last* value written.

In fact, even if a register is used by a set of processes one at a time (i.e., we also say in a serial manner) and without crashing, we could still define the specification of the register using those simple properties. By serial access we mean that a process does not invoke an operation on a register if some other process has invoked an operation and did not receive any reply.

Failure Issues. If we assume that processes might fail, say by crashing, we cannot require that any process that invokes an operation eventually completes that operation. Indeed, a process might crash right after invoking an operation and would not have the time to complete this operation. We say that the operation has failed. (Remember that failures are unpredictable and this is precisely what makes distributed computing challenging).

However, it makes sense to require that if a process p_i invokes some operation and does not subsequently crash, then p_i gets back a reply to its invocation, i.e., completes its operation. That is, any process that invokes a read or write operation, and does not crash, is supposed to eventually return from that invocation. Its operation should not fail. This requirement makes the register *fault-tolerant*. It is also sometimes said to be *robust* or *wait-free*.

If we assume that processes access a register in a serial manner, we may, at first glance, still want to require from a read operation that it returns the last value written. We need however to be careful here with failures in defining the very notion of *last*. To illustrate the underlying issue, consider the following situation.

- Assume that a process p_1 invokes a write on the register with value v_1 and terminates its write. Later on, some other process p_2 invokes a write operation on the register with a new value v_2, and then p_2 crashes before the operation terminates: before it crashes, p_2 did not get any indication that the operation has indeed taken place, i.e., the operation has failed. Now, if even later on, process p_3 invokes a read operation on the register, then what is the value that is supposed to be returned to p_3? should it be v_1 or v_2?

In fact, we will consider both values to be valid replies. Intuitively, p_2 might have or not had the time to terminate the writing operation. In other words, when we require that a read returns the last value written, we consider the two following cases as possible:

1. The value returned has indeed been written by the last process that completed its write, even if some process has invoked a write later but crashed. In this case, no future read should be returning the value of the failed write; Everything happens as if the failed operation was never invoked.
2. The value returned was the input parameter of the last write operation that was invoked, even by some process that crashed before the completion of the actual operation. Everything happens as if the operation that failed has completed.

In fact, the difficulty underlying the problem of failure just discussed has actually to do with a failed write (the one of the crashed process p_2) being concurrent with a read (i.e., the one that comes from p_3 after the crash): this happens even if a process does not invoke an operation while some other process is still waiting for a reply. The difficulty is related to the context of concurrent invocations discussed below.

Concurrency Issues. What should we expect from a value returned by a read operation that is concurrent with some write operation? What is the meaning of the *last* write in this context? Similarly, if two write operations were invoked concurrently, what is the last value written? Can a subsequent read return one of the values and then a read that comes even later return the other?

In this chapter, we will give the specifications of register abstractions (i.e., *regular* and *atomic*) that differ mainly in the way we address these questions, as well algorithms that implement these specifications. Roughly speaking, a regular register ensures minimal guarantees in the face of concurrent and failed operations. An atomic register is stronger and provides strong properties even in the face of concurrency and failures. To make the specifications more precise, we first introduce below some definitions that aim to capture the intuitions discussed above (remember that, by default, we consider that a process does nor recover after a crash; later in the chapter we will consider the fail-recovery model).

4.1.3 Completeness and Precedence

We first define the notions of *completeness* of an operation execution and *precedence* between operation executions, e.g., read or write executions. Note that when there is no possible ambiguity, we simply talk about *operations* to mean operation *executions*.

These notions are defined using the events that occur at the *boundary* of an operation: the *request invocation* (read or write invocation) and the *return confirmation (ack)* or the actual *reply value* in the case of a read invocation. Each of these events is assumed to occur at a single indivisible point in time. (Remember that we assume a fictional notion of global time, used to reason about specifications and algorithms. This global time is however not directly accessible to the processes.)

- We say that an operation is *complete* if *both* events defining the operation have occured.
  This basically means that the process which invoked the operation *op* did not crash before being informed that *op* is terminated, i.e., before the confirmation event occured.
- A *failed* operation is one that was invoked, but the process which invoked it crashed before obtaining any reply.
- An operation *op* (e.g., read or write) is said to *precede* an operation *op'* (e.g., read or write) if:
  - the event corresponding to the return of *op* occurs before (i.e., precedes) the event corresponding to the invocation of *op'*;

  It is important to notice here that, for an operation *op*, invoked by some process p_1 to *precede* an operation *op'* (invoked by a different process) p_2, *op* must be complete.
- If two operations are such that one precedes the other, we say that the operations are *sequential*. Otherwise we say that they are *concurrent*.

Basically, every execution of a register can be viewed as a partial order of its read and write operations. If only one process invokes operations, then the order is total. When there is no concurrency and all operations are complete, the order is also total.

- When a read operation r returns a value v, and that value v was the input parameter of some write operation w, we say that r (resp. v) has (was) *read from* w.
- A value v is said to be written when the write of v is complete.

Module:

Name: (1,N)RegularRegister (on-rreg).

Events:

Request: $\langle$ *on-rRedRead*, reg $\rangle$: Used to invoke a read operation on register reg.
Request: $\langle$ *on-rregWrite*, reg, v $\rangle$: Used to invoke a write operation of value v on register reg.

Confirmation: $\langle$ *on-rregReadReturn*, reg, v $\rangle$: Used to return v as a response to the read invocation on register reg and indicates that the operation is complete.
Confirmation: $\langle$ *on-rregWriteReturn*, reg $\rangle$: Indicates that the write operation has taken place at register reg and is complete.

Properties:

RR1: *Termination:* If a correct process invokes an operation, the process eventually returns from the invocation.

RR2: *Validity:* A read returns the last value written, or the value concurrently written.

Module 4.1 Interface and properties of a (1,N) regular register.

4.2 (1,N) Regular Register

We give here the specification and underlying algorithms of a (1,N) *regular* register, i.e., one specific process, say p_1 can invoke a write operation on the register and any process can invoke a read operation on that register.

4.2.1 Specification

The interface and properties of a (1,N) regular register are given in Module 4.1. In short, a read that is not concurrent with any write returns the last value written. Otherwise (i.e., if there is a concurrent write), the read is allowed to return the last value written or the value concurrently written. Note that if a process invokes a write and crashes (without recovering), the write is considered to be concurrent with any read that did not precede it. Hence, such a read can return the value that was supposed to be written by the failed write or the previous value written, i.e., the last value written before the failed write was invoked. Note also that, in any case, the value returned must be read from some write operation invoked on the register. That is, a value read must in any case be a value that some process has tried to write (even if the write was not complete): it cannot be invented out of thin air. This can be the initial value of the register, which we assume to have been written initially by the writer.

Example. To illustrate the specification of a regular register, we depict in Figure 4.1 and Figure 4.2 two executions. The first is not permitted by a regular register whereas the second is. In the first case, even when there is no concurrency, the read does not return the last value written.

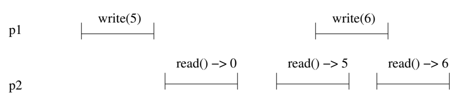

Figure 4.1. Non-regular register execution

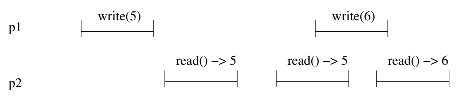

Figure 4.2. Regular register execution

4.2.2 Fail-Stop Algorithm: Read-One-Write-All Regular Register

Algorithm 4.1 implements a (1,N) regular register. The simplicity of this algorithm lies in its relying on a perfect failure detector (fail-stop model). The crash of a process is eventually detected by all correct processes (*strong completeness*), and no process is detected to have crashed until it has really crashed (*strong accuracy*).

The algorithm has each process store a copy of the *current* register value in a variable that is local to the process. In other words, the value of the register is replicated at all processes. The writer updates the value of all processes it does not detect to have crashed. When the write of a new value is complete, all processes that did not crash have the new value. The reader simply returns the value it has stored locally. In other words, the reader *reads one value* and the writer *writes all values*. Henve the name of Algorithm 4.1 (read-one-write-all algorithm).

Besides a perfect failure detector, our algorithm makes use of two underlying communication abstractions: perfect point-to-point links as well as a best-effort broadcast.

Correctness. The *termination* property is straightforward for any read invocation. A process simply reads (i.e., returns) its local value. For a write invocation, termination follows from the properties of the underlying communication abstractions (perfect point-to-point communication and best-effort broadcast) and the *completeness* property of a perfect failure detector (every crashed process is eventually detected by every correct process). Any process that crashes is detected and any process that does not crash sends back an acknowledgement which is eventually delivered by the writer.

Algorithm 4.1 Read-one-write-all regular register algorithm.

```
Implements:
    (1,N)RegularRegister (on-rreg).

Uses:
    BestEffortBroadcast (beb).
    PerfectPointToPointLinks (pp2p).
    PerfectFailureDetector ($\mathcal{P}$).

upon event ⟨ Init ⟩ do
    forall $r$ do
        value[$r$] := 0;
        writeSet[$r$] := $\emptyset$;
    correct := $\Pi$;

upon event ⟨ crash, $p_i$ ⟩ do
    correct := correct \ {$p_i$};

upon event ⟨ on-rregRead, reg ⟩ do
    trigger ⟨ on-rregReadReturn, reg, value[reg] ⟩;

upon event ⟨ on-rregWrite, reg, val ⟩ do
    trigger ⟨ bebBroadcast, [WRITE, reg, val] ⟩;

upon event ⟨ bebDeliver, $p_j$, [WRITE, reg, val] ⟩ do
    value[reg] := val;
    trigger ⟨ pp2pSend, $p_j$, [ACK, reg] ⟩;

upon event ⟨ pp2pDeliver, $p_j$, [ACK, reg] ⟩ do
    writeSet[reg] := writeSet[reg] $\cup$ {$p_j$};

upon exists $r$ such that correct $\subseteq$ writeSet[$r$] do
    writeSet[$r$] := $\emptyset$;
    trigger ⟨ on-rregWriteReturn, r ⟩;
```

Consider *validity*. Assume that there is no concurrency and all operations are complete. Consider a read invoked by some process p_i and assume furthermore that v is the last value written. By the *accuracy* property of the perfect failure detector, at the time when the read is invoked, all processes that did not crash have value v. These include p_i which indeed returns v, i.e., the last value written.

Assume now that the read is concurrent with some write of a value v and the value written prior to v was v' (this could be the initial value 0). By the properties of the communication abstractions, no message is altered and no value can be stored at a process unless the writer has invoked a write operation with this value as a parameter. Hence, at the time of the read, the value can either be v or v'.

Performance. Every write operation requires one communication round-trip between the writer and all processes, and at most $2N$ messages. A read operation does not require any remote communication: it is purely local.

4.2.3 Fail-Silent Algorithm: Majority-Voting Regular Register

It is easy to see that if the failure detector is not perfect, the read-one-write-all algorithm (i.e., Algorithm 4.1) might not ensure the *validity* property of the register. We depict this possibility through the execution of Figure 4.3. Even without concurrency and without any failure, process p_2 returns a value that was not the last value written. This might happen if p_1, the process that has written that value, has falsely suspected p_2 to have crashed, and p_1 did not make sure p_2 has locally stored the new value, i.e., 6.

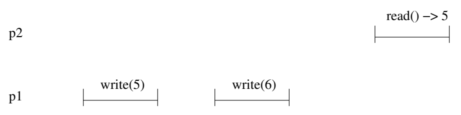

Figure 4.3. A non-regular register execution

In the following, we give a regular register algorithm in a fail-silent model. This algorithm does not rely on any failure detection scheme. Instead, the algorithm assumes a majority of correct processes. We leave it as an exercise to show that this majority assumption is actually needed, even when an eventually perfect failure detector can be used.

The general principle of the algorithm consists for the writer and readers to use a set of *witness* processes that keep track of the most recent value of the register. The witnesses must be chosen in such a way that at least one witness participates in any pair of such operations and does not crash in the meantime. Sets of witnesses must intuitively form *quorums*: their intersection should not be empty. This is ensured by the use of majorities and Algorithm 4.2 is called the *a majority-voting algorithm.* The algorithm indeed implements a (1,N) regular register where one specific process is the writer, say p_1, and any process can be the reader.

Similarly to our previous read-one-write-all algorithm (i.e., Algorithm 4.1), our majority voting algorithm (i.e., Algorithm 4.2) also has each process store a copy of the *current* register value in a local variable. Furthermore, the algorithm relies on a timestamp (we also say sequence number) associated with the value, i.e., which each value stored locally at a process. This timestamp is defined by the writer, i.e., p_1, and intuitively represents the version number (or the age) of the value. It measures the number of times the write operation has been invoked.

- For p_1 (the unique writer) to write a new value, p_1 defines a new timestamp by incrementing the one it already had and associates it with the value to

Algorithm 4.2 Majority-voting regular register algorithm.

```
Implements:
    (1,N)RegularRegister (on-rreg).
Uses:
    BestEffortBroadcast (beb).
    perfectPointToPointLinks (pp2p).

upon event ⟨ Init ⟩ do
    forall $r$ do
        ts[$r$] := sn[$r$] := v[$r$] := acks[$r$] := reqid[$r$] := 0; readSet[$r$] := $\emptyset$;

upon event ⟨ on-rregWrite, r, val ⟩ do
    ts[$r$] := ts[$r$] + 1; acks[$r$] := 0; reqid[$r$] := reqid[$r$] + 1;
    trigger ⟨ bebBroadcast, [WRITE, r, reqid, ts[$r$], val] ⟩;

upon event ⟨ bebDeliver, $p_j$, [WRITE, r, id, tsamp, val] ⟩ do
    if tstamp > sn[$r$] then
        v[$r$] := val; sn[$r$] := tstamp;
        trigger ⟨ pp2pSend, $p_j$, [ACK, r, id] ⟩;

upon event ⟨ pp2pDeliver, $p_j$, [ACK, r, id] ⟩ do
    if id=reqid[$r$] then acks[$r$] := acks[$r$] + 1;

upon exists $r$ such that acks[$r$] > $N/2$) do
    trigger ⟨ on-rregWriteReturn, r ⟩;

upon event ⟨ on-rregRead, r ⟩ do
    reqid[$r$] := reqid[$r$] +1; readSet[$r$] := $\emptyset$;
    trigger ⟨ bebBroadcast, [READ, r, reqid[$r$]] ⟩;

upon event ⟨ bebDeliver, $p_j$, [READ, r, id] ⟩ do
    trigger ⟨ pp2pSend, $p_j$,[READVALUE, r, id, $sn[r], v[r]$] ⟩;

upon event ⟨ pp2pDeliver, $p_j$, [READVALUE, r, id, version, val] ⟩ do
    if id=reqid[$r$] then readSet[$r$] := readSet[$r$] $\cup$ {(version, val)};

upon exists $r$ such that |readSet[$r$]| > $N/2$) do
    v[$r$] := highest(readSet[$r$]);
    trigger ⟨ on-rregReadReturn, r, v[$r$] ⟩;
```

be written. Then p_1 sends a message to all processes, and has a majority adopt this value (i.e., store it locally), as well as the value's timestamp. Process p_1 considers the write to be complete (and hence returns the indication *ack*) when p_1 has received an acknowledgement from a majority of processes indicating that they have indeed adopted the new value and the corresponding timestamp. It is important at this point to notice that a process p_i will only adopt a value sent by the writer, and consequently sends back an acknowledgement, if p_i has not already adopted a more recent value (with a higher timestamp). Process p_1 might have adopted an old value if for instance p_1 has sent a value v_1, then later a value v_2, and process p_i receives v_2 and then v_1. This would mean that p_i was not in the majority that made it possible for p_1 to terminate its writing of v_1, before proceeding to the writing of v_2.

- To read a value, the reader process (it can be any process) selects the value with the highest timestamp among a majority: the processes in this majority act as witnesses of what was written before. The two majorities do not need to be the same. Choosing the highest timestamp ensures that the last value is chosen, provided there is no concurrency. In our majority voting algorithm (Algorithm 4.2), the reader uses a function *highest* that returns the value with the highest timestamp among a set of pairs (timestamp, value) among the set of all pairs returned by a majority. Note that every request is tagged with a unique identifier, and that the corresponding replies carry this identifier. In this way, the reader, can figure out whether a given reply message matches a given request message (and is not an old reply). This is important here since the reader could for instance confuse two messages: one for an old read invocation and one for a new one. This might lead to violate the *validity* property of the register.

Correctness. The *termination* property follows from the properties of the underlying communication abstractions and the assumption of a majority of correct processes. Consider now *validity.*

Consider first the case of a read that is not concurrent with any write. Assume furthermore that a read is invoked by some process p_i and the last value written by p_1, say v, has timestamp sn_1 at p_1. This means that, at the time when the read is invoked, a majority of the processes have timestamp sn_1 and there is no higher timestamp in the system. This is because the writer uses increasing timestamps.

Before reading a value, i.e., returning from the read operation, p_i consults a majority of processes and hence gets at least one value with timestamp sn_1. This is because majorities always intersect (i.e., they form quorums). Process p_i hence returns value v with timestamp sn_1, which is indeed the last value written.

Consider now the case where the read is concurrent with some write of value v and timestamp sn_1, and the previous write was for value v' and timestamp $sn_1 - 1$. If any process returns sn_1 to p_i, p_i will return v, which is a valid reply. Otherwise, at least one process will return $sn_1 - 1$ and p_i will return v', which is also a valid reply.

Performance. Every write operation requires one communication round-trip between the writer and a majority of the processes and every read requires one communication round-trip between the reader and a majority of the processes. In both operations, at most $2N$ messages are exchanged.

4.3 (1,N) Atomic Register

We give here the specification and underlying algorithms of a $(1, N)$ atomic register. The generalization to multiple writers will be discussed in the next section.

4.3.1 Specification

With a regular register specification, nothing prevents a process from reading a value v and then v', even if the writer process has written v' and then v, as long as the writes and the reads are concurrent. Furthermore, consider a register on which only one write operation was invoked by the writer p_1, say with some value v, and p_1 crashed before returning from the operation and does not recover, i.e., the operation is not complete. A subsequent reader might read v whereas another, coming even later, might not, i.e., might return the initial value of the register. In short, an atomic register is a regular register that prevents such behaviors.

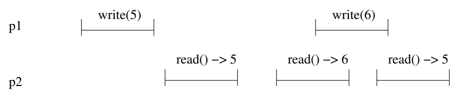

Figure 4.4. Non-atomic register execution

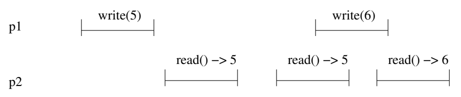

Figure 4.5. Atomic register execution

The interface and properties of a (1,N) atomic register are given in Module 4.2. A (1,N) atomic register is a regular register that, in addition to the properties of a regular register (Module 4.1) ensures a specific *ordering* property which, roughly speaking, prevents an old value to be read once a new value has been read.

Typically, with a (1,N) atomic register, a reader process cannot read a value v', after some value v was read (possibly by some other process), if v' was written before v. In addition, consider a register on which one write operation was invoked and the writer that invoked this operation, say with some value v, crashed before returning from the operation, i.e., the operation is not complete. Once a subsequent reader reads v, no subsequent reader can read the initial value of the register.

Module:

Name: (1,N)AtomicRegister (on-areg).

Events:

Request: ⟨ *on-aregRead*, *reg* ⟩: Used to invoke a read operation on register *reg*.
Request: ⟨ *on-aregWrite*, reg, v ⟩: Used to invoke a write operation of value v on register *reg*.

Confirmation: ⟨ *on-aregReadReturn*, reg, v ⟩: Used to return v as a response to the read invocation on register *reg* and indicates that the operation is complete.
Confirmation: ⟨ *on-aregWriteReturn*, *reg* ⟩: Indicates that the write operation has taken place at register *reg* and is complete.

Properties:

AR1: *Termination:* If a correct process invokes an operation, the process eventually returns from the invocation (same as RR1).

AR2: *Validity:* A read returns the last value written, or the value concurrently written (same as RR2).

AR3: *Ordering:* If a read returns v_2 after a read that precedes it has returned v_1, then v_1 cannot be written after v_2.

Module 4.2 Interface and properties of a (1,N) atomic register.

The execution depicted in Figure 4.5 is that of an atomic register whereas the execution depicted in Figure 4.4 is not. In the execution of Figure 4.4, the *ordering* property of an atomic register should prevent the read of process p_2 to return 6 and then 5, given that 5 was written before 6.

It is important to notice that none of our previous algorithms implements a (1,N) atomic register. We illustrate this through the execution depicted in Figure 4.6 as a counter example for our read-one-write-all regular register algorithm (Algorithm 4.1), and the execution depicted in Figure 4.7 as a counter example for our majority-voting regular register algorithm (Algorithm 4.2).

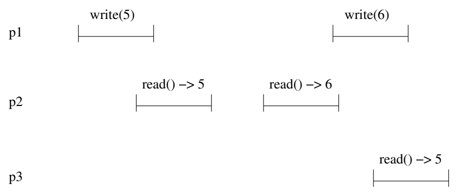

Figure 4.6. Violation of atomicity in the read-one-write-all regular register algorithm

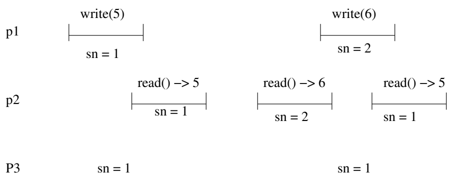

Figure 4.7. Violation of atomicity in the majority-voting regular register algorithm

- The scenario of Figure 4.6 can indeed occur with Algorithm 4.1 if p_1, during its second write, communicates the new value 6 to p_2 before p_3, and furthermore, before p_2 reads locally 6 but after p_3 reads locally 5. This can happen even if the read of p_2 precedes the read of p_3.
- The scenario of Figure 4.7 can occur with Algorithm 4.2 if p_2 has accessed p_1 in its second *read* and p_3 in its third *read* before p_1 accesses any of p_2 and p_3 in its second *write*. Clearly, this can also occur for Algorithm 4.1.

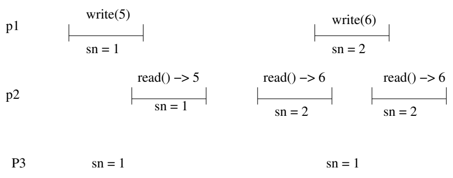

Figure 4.8. An (1,N) atomic register execution

In the following, we give algorithms that implement the (1,N) atomic register abstraction. We first describe how to automatically transform any (fail-stop or fail-silent) (1,1) regular algorithm into a (1,N) atomic register algorithm. Such a transformation does not lead to an efficient implementation but it is modular and helps understand the fundamental difference between atomic and regular registers. It does not however lead to efficient algorithms. We later describe how to extend our regular register algorithms in an ad-hoc way and obtain an efficient (1,N) atomic register algorithms.

4.3.2 Transformation: From (1,N) Regular to (1,N) Atomic

For pedagogical reasons, we divide the problem of transforming any (1,N) regular register into a (1,N) atomic register algorithm in two parts. We first explain how to transform any (1,1) regular register algorithm into a (1,1) atomic register algorithm and then how to transform any (1,1) atomic register algorithm into a (1,N) atomic register algorithm. It is important to notice that these transformations do not use any other means of communication between processes than the underlying registers.

From (1,N) Regular to (1,1) Atomic. The first transformation is given in Algorithm 4.3 and its underlying idea is simple. To build a (1,1) atomic register with p_1 as a writer and p_2 as a reader, we make use of one (1,N) regular register the writer of which is also p_1 and the reader is also p_2. Furthermore, the writer p_1 maintains a timestamp that it increments and associates with every new value to be written. The reader also maintains a timestamp, together with a variable to locally store the latest value read from the register. Intuitively, the goal of storing this value is to make sure an old value is not returned after a new one has been returned.

In Algorithm 4.3, the underlying regular register is denoted by *regReg* and the atomic register to be implemented is denoted by *reg*.

- To write a value v, in the atomic register *reg*, the writer p_1 increments its timestamp and writes it, together with v in the underlying regular register *regReg*.
- To read a value in the atomic register *reg*, the reader p_2 reads the value in the underlying regular register *regReg* as well as the associated timestamp. If the timestamp read is higher than the one previously locally stored by the reader, then p_2 stores the new timestamp read, together with the new value read, and returns the latest. Otherwise, the reader simply returns the value it already had locally stored.

Correctness. The *termination* property of the atomic register follows from the one of the underlying regular register.

Consider *validity.* Assume first a read that is not concurrent with any write and the last value written by p_1, say v, is associated with sequence number sn_1. The sequence number stored by p_2 is either sn_1, if p_2 has already read v in some previous read or a strictly lower value. In both cases, by the *validity* property of the regular register, a read by p_2 will return v. Consider now the case where the read is concurrent with some write of value v and sequence number sn_1, and the previous write was for value v' and sequence number $sn_1 - 1$. The sequence number stored by p_2 cannot be strictly higher than sn_1. Hence, by the *validity* property of the underlying regular register, p_2 will return either v or v', both are valid replies.

Consider the *ordering* property. Assume p_1 writes value v and then v'. Assume p_2 returns v' for some read and consider any subsequent read of p_2. The sequence number stored locally at p_2 is either the one associated with v' or a higher one. By the transformation algorithm, there is no way p_2 can return v.

Performance. Interestingly, writing in the atomic register requires only a local computation (incrementing a timestamp) in addition to writing in the regular register. Similarly, reading from the atomic register requires only a local computation (performing a test and possibly some affectations) in addition to reading from the regular register. This observation means that

Algorithm 4.3 From (1,N) regular to (1,1) atomic registers.

```
Implements:
    (1,1)AtomicRegister (oo-areg).

Uses:
    (1,N)RegularRegister(on-rreg).

upon event ⟨ Init ⟩ do
    forall $r$ do
        ts[$r$] := sn[$r$] := v[$r$] := 0;

upon event ⟨ oo-aregWrite, r, $v$ ⟩ do
    ts[$r$] := ts[$r$] + 1;
    trigger ⟨ on-rregWrite, $r$, ts[$r$], v[$r$] ⟩;

upon event ⟨ on-rregWriteReturn, $r$ ⟩ do
    trigger ⟨ oo-aregWriteReturn, $r$ ⟩;

upon event ⟨ oo-aregRead, r ⟩ do
    trigger ⟨ on-rregRead, $r$ ⟩;

upon event ⟨ on-rregReadRet, $r$, (tstamp,val) ⟩ do
    if tstamp > sn[$r$] then
        sn[$r$] := tstamp; v[$r$] := val;
    trigger ⟨ oo-aregReadReturn, $r$, v[$r$] ⟩;
```

no messages need to be added to an algorithm that implements a (1,1) regular register in order to implement a (1,1) atomic register.

From (1,1) Atomic to (1,N) Atomic. We describe here an algorithm that implements the abstraction of a (1,N) atomic register out of (1,1) atomic registers. To get an intuition of the transformation, think of a teacher, i.e., the writer, who needs to communicate some information to a set of students, i.e., the readers, through the abstraction of a traditional black-board. In some sense, a board is typically a (1,N) register, as far as only the teacher writes on it. It is furthermore atomic as it is made of a single physical entity.

Assume however that the teacher cannot physically gather all students within the same classroom and hence cannot use one physical board for all. Instead, this global board needs to be emulated using one or several electronic boards (e-boards) that could also be written by one person but could only be read by one person, say every student can have one or several of such boards at-home that only this student can read.

It makes sense to have the teacher write every new information on at least one board per student. This is intuitively necessary for the students to eventually read the information provided by the teacher, i.e., to ensure the *validity* property of the register. This is however not enough if we want to guarantee the *ordering* property of an atomic register. Indeed, assume that the teacher writes two consecutive information X and then Y. It might happen that a student reads Y and then later on, some other student reads X, say because the information flow from the teacher to the first student is faster than the flow to the second teacher. This case of *ordering* violation is in a sense similar to the situation of Figure 4.6.

One way to cope with this issue is, for every student, before terminating the reading of some information, to transmit this information to all other students, through other e-boards. That is, every student would devote, besides the e-board devoted for the teacher to provide her with new information, another one for every other student to write new information in. Whenever a student reads some information from the teacher, she first writes this information in the boards of all other students before returning the information. Old and new information are distinguished using timestamps.

The transformation we give in Algorithm 4.4 uses a number of $(1, 1)$ atomic registers to build one $(1, N)$ atomic register, denoted by *reg*. The writer of the latter register *reg* is p_1. The $(1, 1)$ registers are used in the following way:

1. A series of N of (1,1) atomic registers, with identities stored in variables $writer[reg, 1]$, $writer[reg, 2]$, ..., $writer[reg, N]$. These registers are used to communicate between the writer, i.e., p_1, and each of the N readers. In all these registers, the writer is p_1. The reader of register $writer[reg, k]$ is p_k.
2. A series of N^2 of (1,1) atomic registers, with identities stored in variables $readers[reg, 1, 1]$, ..., $readers[reg, i, j]$, ... , $readers[reg, N, N]$. These registers are used to communicate between the readers. In any register with identifier $readers[reg, i, j]$, the reader is p_i and the writer is p_j.

The algorithm also relies on a timestamp ts that indicates the version of the current value of the register. We make also use here of a function *highest* that returns the pair (timestamp, value) with the highest timestamp among a set of such pairs.

Correctness. By the *termination* property of the underlying (1,1) atomic registers and the fact that the transformation algorithm contains no loop or wait statement, every operation eventually returns. Similarly, by the *validity* property of the underlying (1,1) atomic registers, and the fact that the value with the largest timestamp is chosen to be returned, we also derive the *validity* of the (1,N) atomic register. Consider now the *ordering* property. Consider a write w_1 of a value v_1 with timestamp s_1 that precedes a write w_2 with value v_2 and timestamp s_2 ($s_1 < s_2$). Assume that some read operation returns v_2: by the algorithm, for any j in $[1, N]$, p_i has written (s_2, v_2) in $r_{i,j}$. By the *ordering* property of the underlying $(1, 1)$ registers, every subsequent read will return a value with a timestamp at least as high as s_2, i.e., there is no way to return v_1.

Algorithm 4.4 From (1,1) atomic to (1,N) atomic registers.

```
Implements:
    (1,N)AtomicRegister (on-areg).

Uses:
    (1,1)AtomicRegister (oo-areg).

upon event ⟨ Init ⟩ do
    i := rank (self); // rank of local process (integer in range $1 \dots N$)
    forall $r$ do
        ts[$r$] := acks[$r$] := readval[$r$] := 0; readSet[$r$] := $\emptyset$;
        for $j = 1$ to $N$ do // assign namespace of (1,1) atomic registers
            writer[$r, j$] := $(r-1)(N^2+N)+j$;
            for $k = 1$ to $N$ do readers[$r, j, k$] := $(N^2+N)(r-1)+jN+k$;

upon event ⟨ on-aregWrite, r, $v$ ⟩ do
    ts[$r$] := ts[$r$] + 1;
    for $j = 1$ to $N$ do trigger ⟨ oo-aregWrite, writer[$r, j$], (ts[$r$], v) ⟩;

upon event ⟨ oo-aregWriteReturn, writer[$r, j$] ⟩ do
    acks[$r$] := acks[$r$] + 1;

upon exists $r$ such that acks[$r$] = N do
    acks[$r$] := 0; trigger ⟨ on-aregWriteReturn, $r$ ⟩;

upon event ⟨ on-aregRead, r ⟩ do
    readSet[$r$] := $\emptyset$; for $j = 1$ to $N$ do trigger ⟨ oo-aregRead, readers[$r, i, j$] ⟩;

upon event ⟨ oo-aregReadReturn, readers[$r, i, j$], (tstamp, v) ⟩ do
    readSet[$r$] := readSet[$r$] $\cup$ {(tstamp, v)};

upon exists $r$ such that |readSet[$r$]| = $N$) do
    trigger ⟨ oo-aregRead, writer[$r, i$] ⟩;

upon event ⟨ oo-aregReadReturn, writer[$r, i$], (tstamp, v) ⟩ do
    (maxts,reaval[$r$]) := highest (readSet[$r$] $\cup$ (tstamp ,v));
    for $j = 1$ to $N$ do
        trigger ⟨ oo-aregWrite, readers[$r, j, i$], maxts, readval[$r$] ⟩;

upon event ⟨ oo-aregWriteReturn, readers[$r, j, i$] ⟩ do
    acks[$r$] := acks[$r$] + 1;

upon exists $r$ such that acks[$r$] = $N$ do
    acks[$r$] := 0; trigger ⟨ on-aregReadReturn, $r$, readval[$r$] ⟩;
```

Performance. Every write operation into the (1,N) register requires N writes into (1,1) registers. Every read from the (1,N) register requires one read from N (1,1) registers and one write into N (1,1) registers.

We give in the following two ad-hoc (1,N) atomic register algorithms. The first one is a a fail-stop and the second is a fail-silent algorithm. These are adaptations of the read-one-write-all and majority-voting (1,N) regular register algorithms, respectively. Both algorithms require less messages than those we would obtain through the automatic transformations described above.

4.3.3 Fail-Stop Algorithm: Read-One-Impose-All (1,N) Atomic Register

If the goal is to implement a register with one writer and multiple readers, i.e., (1,N), the read-one-write-all regular register algorithm (Algorithm 4.1) does clearly not work: the scenario depicted in Figure 4.6 illustrates this case.

To cope with this case, we define an extension to the read-one-write-all regular register algorithm (Algorithm 4.1) that circumvents the problem by having the reader also imposes, on all other processes, the value it is about to return. In other words, the read operation acts also as a write. The resulting algorithm, named read-one-impose-all, is depicted in Algorithm 4.5 The writer uses a timestamp to date the values it is writing: it is this timestamp that ensures the *ordering* of every execution. A process that is asked to store a value that is older than the one it has returns an acknowledgement but does not modify its value. We will discuss the need for this test, as well as the need for the timestamp, through an exercise at the end of this chapter.

Correctness. Termination and *validity* are ensured as in Algorithm 4.1. Consider now *ordering*. Assume p_1 writes a value v and then v', which is associated with some timestamp sn. Assume furthermore that some reader p_i reads v' and, later on, some other process p_j invokes another read operation. At the time where p_i completes its read, all processes that did not crash have a timestamp that is at least as hight as sn. By the read-one impose-all algorithm, there is no way p_j will later on change its value with v, as this has a lower timestamp because it was written by p_1 before v'.

Performance. Every write or read operation requires one communication round-trip between the writer or the reader and all processes. At most $2N$ messages are needed in both cases.

4.3.4 Fail-Silent Algorithm: Read-Majority Impose-Majority (1,N) Atomic Register

In the following, we consider a fail-silent model. We describe an adaptation of our majority-voting (1,N) regular register (Algorithm 4.2) that implements a (1,N) atomic register algorithm.

This adaptation, called read-majority impose-majority, is depicted in Algorithm 4.6. The implementation of the write operation is similar to that of the majority-voting algorithm (Algorithm 4.2): the writer makes simply sure a majority adopts its value. The implementation of the read operation is however different. A reader selects the value with the highest timestamp among a majority, as in the majority-voting algorithm, but now also makes

Algorithm 4.5 Read-one-impose-all (1,N) atomic register algorithm.

```
Implements:
    (1,N)AtomicRegister (on-areg).

Uses:
    BestEffortBroadcast (beb).
    PerfectPointToPointLinks (pp2p).
    PerfectFailureDetector ($\mathcal{P}$).

upon event ⟨ Init ⟩ do
    correct := $\Pi$;
    forall $r$ do
        v[$r$]:= ts[$r$] := sn[$r$] := readval[$r$] := rqid[$r$] := 0;
        reading[$r$] := false; writeSet[$r$] := $\emptyset$;

upon event ⟨ crash, $p_i$ ⟩ do
    correct := correct \ {$p_i$};

upon event ⟨ on-aregRead, r ⟩ do
    rqid[$r$] := rqid[$r$] + 1; reading[$r$] := true; readval[$r$] := $v[r]$;
    trigger ⟨ bebBroadcast, [WRITE, r, reqid, sn[$r$], v[$r$]] ⟩;

upon event ⟨ on-aregWrite, r, val ⟩ do
    rqid[$r$] := rqid[$r$] + 1; ts[$r$] := ts[$r$] + 1;
    trigger ⟨ bebBroadcast, [WRITE, r, reqid[$r$], ts[$r$], val] ⟩;

upon event ⟨ bebDeliver, $p_j$,[WRITE, r, id, tstamp, val] ⟩ do
    if tstamp > sn[$r$] then
        v[$r$] := val; sn[$r$] := tstamp;
        trigger ⟨ pp2pSend, $p_j$, [ACK, r, id] ⟩;

upon event ⟨ pp2pDeliver, $p_j$, [ACK, r, id] ⟩ do
    if id = reqid[$r$] then writeSet[$r$] := writeSet[$r$] ∪ {$p_j$};

upon exists $r$ such that correct ⊆ writeSet[$r$] do
    writeSet[$r$] := $\emptyset$;
    if (reading[$r$] = true) then
        reading[$r$] := false;
        trigger ⟨ on-aregReadReturn, r, readval[$r$] ⟩;
    else
        trigger ⟨ on-aregWriteReturn, $r$ ⟩;
```

sure a majority adopts this value before completing the read operation: this is key to ensuring the *ordering* property of an atomic register.

It is important to notice that the majority-voting algorithm can be seen as a particular case of the read-majority impose-majority algorithm in the following sense: given that there is only one reader in the majority-voting algorithm, the reader simply adopts itself the value read and makes sure to include itself in the majority.

Algorithm 4.6 Read-Majority Impose-Majority (1,N) atomic register algorithm.

```
Implements:
    (1,N)AtomicRegister (on-areg).

Uses:
    BestEffortBroadcast (beb).
    perfectPointToPointLinks (pp2p).

upon event ⟨ Init ⟩ do
    forall $r$ do
        ts[$r$] := sn[$r$] := v[$r$] := acks[$r$] := reqid[$r$] := 0; ;
        reading[$r$] := false; readSet[$r$] := $\emptyset$;

upon event ⟨ on-aregWrite, r, val ⟩ do
    reqid[$r$] := reqid[$r$] + 1; ts[$r$] := ts[$r$] + 1; acks[$r$] := 0;
    trigger ⟨ bebBroadcast, [WRITE, r, reqid, ts[$r$], val] ⟩;

upon event ⟨ bebDeliver, $p_j$, [WRITE, r, id, t, val] ⟩ do
    if t > sn[$r$] then
        v[$r$] := val; sn[$r$] := t; trigger ⟨ pp2pSend, $p_j$, [ACK, r, id] ⟩;

upon event ⟨ pp2pDeliver, $p_j$, [ACK, r, id] ⟩ do
    if reqid[$r$] = id then acks[$r$] := acks[$r$] + 1;

upon exists $r$ such that acks[$r$] > $N/2$ do
    if reading[$r$] = true then
        reading[$r$] := false; trigger ⟨ on-aregReadReturn, r, $v[r]$ ⟩;
    else trigger ⟨ on-aregWriteReturn, r ⟩;

upon event ⟨ on-aregRead, r ⟩ do
    reqid[$r$] := reqid[$r$] +1; readSet[$r$] := $\emptyset$;
    trigger ⟨ bebBroadcast, [READ, r, reqid[$r$]] ⟩;

upon event ⟨ bebDeliver, $p_j$, [READ, r, id] ⟩ do
    trigger ⟨ pp2pSend, $p_j$, [READVALUE, r, id, $sn[r], v[r]$] ⟩;

upon event ⟨ pp2pDeliver, $p_j$, [READVALUE, r, id, $snb, val$] ⟩ do
    if reqid[$r$] = id then readSet[$r$] := readSet[$r$] $\cup$ {($snb, val$)};

upon exists $r$ such that |readSet[$r$]| > $N/2$ do
    ($tstamp, val$) := highest(readSet[$r$]); acks[$r$] := 0; reading[$r$] := true;
    reqid[$r$] := reqid[$r$] +1; trigger ⟨ bebBroadcast, [WRITE, r, reqid[$r$], tstamp, val] ⟩;
```

Correctness. Termination and *validity* are ensured as in Algorithm 4.2. Consider now the *ordering* property. Assume that a *read* invocation r_1, by process p_i, returns a value v_1 from a *write* invocation w_1, by process p_1 (the only writer), a *read* invocation r_2, by process p_j, returns a different value v_2 from a *write* invocation w_1, also by process p_1, and r_1 precedes r_2. Assume by contradiction that w_2 precedes w_1. By the algorithm, the sequence number that p_1 associated with v_1, ts_k, is strictly higher than the one p_1 associated with v_2, $ts_{k'}$. Given that r_1 precedes r_2, then when r_2 was invoked, a majority have a timestamp that is at least $ts_{k'}$. Hence p_j cannot return v_2, because v_2 has a strictly lower sequence number than v_1. A contradiction.

Performance. Every write operation requires one communication round-trip between p_1 and a majority of the processes. $2N$ messages are exchanged. Every read requires two communication round-trips between the reader and a majority of the processes. $4N$ messages are exchanged.

4.4 (N,N) Atomic Register

4.4.1 Multiple Writers

So far, we have focused on registers with a single writer. That is, our specifications of regular and atomic registers do not provide any guarantees when multiple processes write in a register. It is natural to ask what should be ensured in the case of multiple writers.

One difficulty underlying this question has to do with defining the *validity* property in the case of multiple writers. Indeed, this property requires that a read that is not concurrent with any write should return the *last* value written. But if two processes have written different values concurrently, say v and v', before some other process invokes a read operation, then what should this read return? Assume we make it possible for the reader to return either v or v', do we allow a concurrent reader, or even a reader that comes later, to return the other value?

In the following, we address these questions and we generalize the specification of atomic registers to multiple writers.

4.4.2 Specification

In short, a (N,N) atomic register ensures that failed writes either appear as if they were never invoked or if they were complete, i.e., if they were invoked and terminated. (Clearly, failed read operations do always appear as if they were never invoked.) In addition, even in the face of concurrency, the values returned by reads could have been returned by a serial execution (called a *linearization* of the actual execution) where any operation takes place at some instant between its invocation and reply instants. The execution is in this sense *linearizable* (i.e., it has one linearization). A (N,N) atomic register is a generalization of a (1,N) atomic register in the following sense: every execution of (1,N) atomic register is an execution of a (N,N) atomic register. The interface and properties of a (N,N) atomic register are given in Module 4.3.

Module:

Name: (N,N) Atomic Register (nn-areg).

Events:

Same as for a regular register (just replave "on-" by "nn-" on the interface).

Properties:

NAR1: *Termination:* Same as RR1.

NAR2: *Atomicity:* Every failed operation appears to be complete or does not appear to have been invoked at all, and every complete operation appears to have been executed at some instant between its invocation and reply events.

Module 4.3 Interface and properties of a (N,N) atomic register.

To study the implementation of (N,N) atomic registers, we adopt the same modular approach as for the (1,N) case. We first describe a general transformation that implements a (N,N) atomic register using (1,N) atomic registers. This transformation does not rely on any other way of exchanging information among the processes, besides the underlying (1,N) atomic registers. This helps understand the fundamental difference between both abstractions. We will also study ad-hoc (N,N) atomic register algorithms in various models.

4.4.3 Transformation: From (1,N) atomic to (N,N) atomic registers

To get an intuition of this transformation, think of emulating a general (atomic) board to be used by a set of teachers to provide information to a set of students. All teachers are able to write and read information on the common board. However, what is available are simply boards where only one teacher can write information. If every teacher uses her own board to write information, then it would not be clear for a student which information to select and still guarantee the atomicity of the common board, i.e., the illusion of one physical common board that all teachers share. The difficulty is actually for any given student to select the latest information written. Indeed, if some teacher A writes X and then some other teacher B writes later Y, then a student that comes afterward should read Y. But how can the student know that Y is indeed the latest information, given that what is available are simply individual boards, one for each teacher?

This can in fact be ensured by having the teachers coordinate their writing to create a causal precedence among the information they write. Teacher B that writes Y could actually read the board of teacher A and, when finding X, associate with Y some global timestamp that denotes the very fact that Y is indeed more recent than X. This is the key to the transformation algorithm we present below.

The transformation algorithm, depicted in Algorithm 4.7, uses N (1,N) atomic registers, whose identities are stored in variables $writer[reg, 1]$, ..., $writer[reg, N]$, to build one (N,N) atomic register, denoted by *reg*. Every register $writer[reg, i]$ contains a value and an associated timestamp. Basically, to write a value v in reg, process p_j reads all (1,N) registers and selects the highest timestamp, which it increments and associates with the value v to be written. Then p_j writes in $writer[reg, j]$ the value with the associated timestamp.

To read a value from reg, process p_j reads all registers $writer[reg, 1]$, $writer[reg, 2]$, ..., $writer[reg, N]$, and returns the value with the highest timestamp. To distinguish values that are associated with the same timestamp, p_j uses the identity of the processes that have originally written those values and order them accordingly, i.e., p_j uses the indices of the registers from which it read these timestamps. We define this way a total order among the timestamps associated with the values, and we abstract away this order within a function *highest* that returns the value with the highest order. We also make use of a similar function, called *highest*, but with a different signature, that returns the value with highest timestamp, among a set of triplets (timestamp, value, process identity).

Correctness. The *termination* property of the (N,N) register follows from that of the (1,N) register, whereas *atomicity* follows from the total order used to write values: this order respects the real-time order of the processes.

Performance. Every write operation into the (N,N) atomic register requires N reads from each of the (1,N) registers and one write into a (1,N) register. Every read from the (N,N) register requires N reads from each of the (1,N) registers.

1. Assume we apply the transformation of Algorithm 4.7 to the read-one-impose-all fail-stop algorithm (Algorithm 4.5) in order to obtain a (N,N) atomic register algorithm. Every read in the (N,N) register would involve N parallel communication round trips between the reader and all other processes. Furthermore, every write operation in the (N,N) register would involve N parallel communication round-trips between the writer and all other processe (to determine the highest timestamp) and then another communication round-trip between the writer and all other processes (to perform the actual writing).
2. Similarly, assume we apply the transformation of Algorithm 4.7 to "read-majority impose-majority algorith (Algorithm 4.6) in order to obtain a (N,N) atomic register algorithm. Every read in the (N,N) register would involve N communication round trips between the reader and a majority of processes (to determine the latest value), and then N other communication round trips between the reader and a majority of processes (to impose that value). Furthermore, every write operation in the (N,N) register would involve N parallel communication round-trips between the

Algorithm 4.7 From (1,N) atomic to (N,N) atomic registers.

```
Implements:
    (N,N)AtomicRegister (nn-areg).

Uses:
    (1,N)AtomicRegisters(o-areg).

upon event ⟨ Init ⟩ do
    i := rank (self); // rank of local process (integer in range $1 \ldots N$)
    forall $r$ do
        writeval[$r$] := 0; tSet[$r$] := readSet[$r$] := $\emptyset$;
        for $j = 1$ to $N$ do // assign namespace of (1,N) atomic registers
            writer[$r, j$] := $(r - 1)N + j$;

upon event ⟨ nn-aregWrite, r, v ⟩ do
    writeval[$r$] := v; tSet[$r$] := $\emptyset$;
    for $j = 1$ to $N$ do trigger ⟨ on-aregRead, writer[$r, j$] ⟩;

upon event ⟨ on-aregReadReturn, writer[$r, j$], (tstamp,val) ⟩ do
    tSet[$r$] := tSet[$r$] ∪ { tstamp, $j$ };

upon exists $r$ such that | tSet[$r$]| = $N$ do
    trigger ⟨ on-aregWrite, writer[$r, i$], (highest(tSet[$r$]) +1, writeval[$r$]) ⟩;

upon event ⟨ on-aregWriteReturn, writer[$r, i$] ⟩ do
    trigger ⟨ nn-aregWriteReturn, $r$ ⟩;

upon event ⟨ nn-aregRead, reg ⟩ do
    for $j = 1$ to $N$ do trigger ⟨ on-aregRead, writer[$reg, j$] ⟩;

upon event ⟨ on-aregReadRetutn, writer$w$[$r, j$], (tstamp, val) ⟩ do
    readSet[$r$] := readSet[$r$] ∪ {tstamp, val, $j$};

upon exists $r$ such that |readSet[$r$]| = $N$ do
    trigger ⟨ nn-aregReadReturn, $r$, highest (readSet[$r$]) ⟩;
```

writer and a majority (to determine the highest timestamp) and then another communication round-trip between the writer and a majority (to perform the actual writing).

We present in the following ad-hoc algorithms that require less messages. We describe first a fail-stop algorithm and then a fail-silent algorithm.

4.4.4 Fail-Stop Algorithm: Read-All-Impose-All (N,N) Atomic Register

We describe below an adaptation of our (1,N) read-one-impose-all (1-N) atomic register algorithm (Algorithm 4.5) to deal with multiple writers. To get an idea of the issue introduced by multiple writers, it is important to first figure out why the read-one-impose-all algorithm cannot afford multiple writers. Consider indeed the case of two processes trying to write in a register implemented using the read-one-impose-all algorithm: say processes p_1 and p_2. Assume furthermore that p_1 writes value X, then Y, and later on (after the write of Y is terminated), p_2 writes value Z. If some other process, say p_3, reads the register after the writing of Z is over, p_3 will get value Y, and this is because Y has a higher timestamp: Y has timestamp 2 whereas Z has timestamp 1.

Intuitively, the problem is that the timestamps are generated independently by the processes, which was clearly not the case with a single writer.

What we actually expect from the timestamps is that (a) they be totally ordered, and (b) they reflect the precedence relation between operations. They should not be generated independently by multiple writers, but should in our example reflect the fact that the writing of Y precedes the writing of Z. In the case of multiple writers, we have to deal with the problem of how to determine a timestamp in a distributed fashion. The idea is to have every writer consult first other writers and determine its timestamp by choosing the highest, i.e., we add one communication round-trip between the writer and all processes (that did not crash). (The idea of consulting other writers is key to our transformation above from (1,N) to (N,N) atomic.) It is important to notice that two values might be stored in different processes with the same timestamp. Two consecutive readers that come after the writes might return different values, without any write having been invoked in the meantime. To address this issue, the idea is to use the identity of the processes in the comparison, i.e., use the lexicographical order. (The idea of making use of process identities in the comparions was also key in our transformation from (1,N) to (N,N) atomic.)

The resulting algorithm, called read-all impose-all, depicted in Algorithm 4.8 and 4.9, is an extension of read-one-impose-all algorithm (Algorithm 4.5) that implements a (N,N) atomic register.

Correctness. The *termination* property of the register follows from the *completenes* property of the failure detector and the underlying channels. The *atomicity* property follows from the *accuracy* property of the failure detector.

Performance. Every read in the (N,N) register requires 2 communication round-trips: 4N messages are exchanged. Every write requires 1 communication round-trip: 2N messages are exchanged.

4.4.5 Fail-Silent Algorithm: Majority Voting (N,N) Atomic Register

We describe here how to obtain an algorithm that implements a (N,N) atomic register in a fail-silent model as an extension of our majority-voting algorithm, i.e., Algorithm 4.6. Let us first understand first the issue of multiple writers in Algorithm 4.6. Consider again the case of two processes trying to write in

Algorithm 4.8 Read-All Impose-All (N,N) atomic register algorithm (1/2).

```
Implements:
    (N,N)AtomicRegister (nn-areg).

Uses:
    BestEffortBroadcast (beb). PerfectPointToPointLinks (pp2p).
    PerfectFailureDetector ($\mathcal{P}$).

upon event ⟨ Init ⟩ do
    correct := $\Pi$; i := rank (self);
    forall $r$ do
        writeSet[$r$] := readSet[$r$] := $\emptyset$;
        reqid[$r$] := readval[$r$] := writeval[$r$] := v[$r$] := ts[$r$] := mrank[$r$] :=0;

upon event ⟨ crash, $p_i$ ⟩ do
    correct := correct \ {$p_i$};

upon event ⟨ nn-aRegRead, reg ⟩ do
    reqid[$r$] := reqid[$r$] +1; reading[$r$] := true; readSet[$r$] := $\emptyset$; readval[$r$] := v[$r$];
    trigger ⟨ bebBroadcast, [WRITE, r, reqid[$r$], ($ts$[$r$], i), $v$[$r$]] ⟩;

upon event ⟨ nn-aRegWrite, r, val ⟩ do
    reqid[$r$] := reqid[$r$] +1; writeval[$r$] := val;
    trigger ⟨ bebBroadcast, [READ, r, reqid[$r$]] ⟩;

upon event ⟨ bebDeliver, $p_j$, [READ, r, id] ⟩ do
    trigger ⟨ pp2pSend, $p_j$, [READVALUE, r, id, ($ts$[$r$], mrank[$r$]), $v$[$r$]] ⟩;

upon event ⟨ pp2pDeliver, $p_j$, [READVALUE, r, id, (t,rk), val] ⟩ do
    if id = reqid[$r$] then readSet[$r$] := readSet[$r$] ∪ {$((t, rk), val)$};

upon exists $r$ such that |correct| ≤ |readSet[$r$]| do
    reqid[$r$] := reqid[$r$] +1; ((t,rk), val) := highest(readSet[$r$]); writeSet[$r$] := := $\emptyset$;
```

a register implemented using Algorithm 4.1: say processes p_1 and p_2. Assume furthermore that p_1 writes value X, then Y, and later on (after the write of Y is terminated), p_2 tries to write value Z. Process p_2 will be blocked waiting for acknowledgements from a majority of the processes and *termination* will be violated because at least one acknowledgement will be missing: remember that, in Algorithm 4.2, a (witness) process does not send back an acknowledgement for a request to write a value with a lower timestamp than what the process has. Assume we modify Algorithm 4.2 to alleviate the need for this test and have the witness processes reply in all cases. We will ensure that Z is written. Nevertheless, if some other process reads the register afterward, it will get value Y, and this is because Y has a higher timestamp: Y has timestamp 2 whereas Z has timestamp 1.

We describe in Algorithm 4.10 the events that need to be modified or added to Algorithm 4.6 in order to deal with multiple writers.

Algorithm 4.9 Read-All Impose-All (N,N) atomic register algorithm (2/2).

```
upon event ⟨ bebDeliver, $p_j$,[WRITE, r, id, (t,j), val] ⟩ do
    if (t,j) > (ts[r], mrank[r]) then
        v[$r$] := $val$; $sn$[$r$] := $t$; mrank[r] := j;
        trigger ⟨ pp2pSend, $p_j$, [ACK, r, id] ⟩;

upon event ⟨ pp2pDeliver, $p_j$, [ACK, r, id] ⟩ do
    if id = reqid[$r$] then writeSet[$r$] := writeSet[$r$] $\cup$ {$p_j$};

upon exists $r$ such that correct $\subseteq$ writeSet[$r$] do
    writeSet[$r$] := $\emptyset$;
    if (reading[$r$] = true) then
        reading[$r$] := false;
        trigger ⟨ nn-aregReadReturn, r, readval[$r$] ⟩;
    else
        trigger ⟨ nn-aregWriteReturn, $r$ ⟩;
```

More precisely, the read procedure of our (N,N) atomic register algorithm is similar to that of Algorithm 4.6. The write procedure is different in that the writer first determines a timestamp to associate with the new value to be written by reading at a majority of processes. It is also important to notice that the processes distinguish values with the same timestamps using process identifiers. We assume that every value written is tagged with the identity of the originator process. A value v is considered more recent than a value v', if v has a strictly higher timestamp, or they have the same timestamp and v was written by p_i whereas v' was written by p_j such that $i > j$. We assume here a new function *highest()* and new comparator operator that counts for this stronger ordering scheme.

Correctness. The *termination* property of the register follows from the correct majority assumption and the underlying channels. The *atomicity* property follows from the quorum property of the majority.

Performance. Every read or write in the (N,N) register requires 2 communication round-trips between the reader or the writer and a majority of the processes. In each case, 4N messages are exchanged.

4.5 (1,N) Logged Regular Register

So far, we considered register specifications and implementations under the assumption that processes that crash do not recover. In other words, processes that crash, even if they recover, are excluded from the computation: they can neither read nor write in a register. Furthermore, they cannot help other processes reading or writing by storing and witnessing values. We revisit here this assumption and take into account processes that recover after crashing.

Algorithm 4.10 Read-Majority Impose-Majority (N,N) atomic register algorithm.

```
Implements:
    (N,N)AtomicRegister (nn-areg).

Uses:
    BestEffortBroadcast (beb). PerfectPointToPointLinks (pp2p).

upon event ⟨ Init ⟩ do
    i := rank (self);
    forall $r$ do
        writeSet[$r$] := readSet[$r$] := $\emptyset$;
        reqid[$r$] := readval[$r$] := writeval[$r$] := v[$r$] := ts[$r$] := mrank[$r$] :=0;

upon event ⟨ nn-aRegRead, reg ⟩ do
    reqid[$r$] := reqid[$r$] +1; reading[$r$] := true; readSet[$r$] := $\emptyset$;
    trigger ⟨ bebBroadcast, [READ, r, reqid[$r$]] ⟩;

upon event ⟨ nn-aRegWrite, r, val ⟩ do
    reqid[$r$] := reqid[$r$] +1; writeval[$r$] := val; readSet[$r$] := $\emptyset$;
    trigger ⟨ bebBroadcast, [READ, r, reqid[$r$]] ⟩;

upon event ⟨ bebDeliver, $p_j$, [READ, r, id] ⟩ do
    trigger ⟨ pp2pSend, $p_j$, [READVALUE, r, id, ($ts$[$r$], mrank[$r$]), $v$[$r$]] ⟩;

upon event ⟨ pp2pDeliver, $p_j$, [READVALUE, r, id, (t,rk), val] ⟩ do
    if id = reqid[$r$] then readSet[$r$] := readSet[$r$] $\cup$ $\{((t, rk), val)\}$;

upon exists $r$ such that |readSet[$r$]| $> N/2$ do
    reqid[$r$] := reqid[$r$] +1; ((t,rk),val) := highest (readSet[$r$]); writeSet[$r$] := $\emptyset$;
    if reading[$r$] then
        readval[$r$] := val;
        trigger ⟨ bebBroadcast, [WRITE, r, reqid[$r$], (t+1, i), readval[$r$]] ⟩;
    else
        trigger ⟨ bebBroadcast, [WRITE, r, reqid[$r$], (t+1, i), writeval[$r$]] ⟩;

upon event ⟨ bebDeliver, $p_j$,[WRITE, r, id, (t,j), val] ⟩ do
    if (t,j) > (ts[r], mrank[r]) then
        $v$[$r$] := $val$; $sn$[$r$] := $t$; mrank[r] := j; trigger ⟨ pp2pSend, $p_j$, [ACK, r, id] ⟩;

upon event ⟨ pp2pDeliver, $p_j$, [ACK, r, id] ⟩ do
    if id = reqid[$r$] then writeSet[$r$] := writeSet[$r$] $\cup$ $\{p_j\}$;

upon exists $r$ such that |writeSet[$r$]| $> N/2$ do
    if (reading[$r$] = true) then
        reading[$r$] := false; trigger ⟨ nn-aregReadReturn, r, readval[$r$] ⟩;
    else trigger ⟨ nn-aregWriteReturn, $r$ ⟩;
```

4.5.1 Precedence in the Fail-Recovery Model

To define the notion of a register in a fail-recovery model, we first revisit the notion of precedence introduced earlier, assuming by default, fail-no-recovery models.

Module:

Name: (1,N)LoggedRegularRegister (on-log-rreg).

Events:

Request: ⟨ *on-log-rregRead*, *reg* ⟩: Used to invoke a read operation on register *reg*.

Request: ⟨ *on-log-rregWrite*, reg, v ⟩: Used to invoke a write operation of value v on register *reg*.

Confirmation: ⟨ *on-log-rregReadReturn*, reg, v ⟩: Used to return v as a response to the read invocation on register *reg* and indicates that the operation is complete.

Confirmation: ⟨ *on-log-rregWriteReturn*, *reg* ⟩: Indicates that the write operation has taken place at register *reg* and is complete.

Properties:

SRR1: *Termination:* If a process invokes an operation and does not crash, the process eventually returns from the invocation.

SRR2: *Validity:* A read returns the last value written, or the value concurrently written.

Module 4.4 Interface and properties of a (1,N) logged regular register.

- We say that an operation *op* (e.g., read or write) *precedes* an operation *op'* (e.g., read or write) if any of the two following conditions hold.

1. the event corresponding to the return of *op* occurs before (i.e., precedes) the event corresponding to the invocation of *op'*;
2. the operations are invoked by the same process and the event corresponding to the invocation of *op'* occurs after the event corresponding to the invocation of *op*.

It is important to notice here that, for an operation *op*, invoked by some process p_1 to *precede* an operation *op'* invoked by the same process, *op* does not need to be complete. In this case, a process might have invoked *op*, crash, recover, and invoke op'. This was clearly not possible in a crash-non-recovery models.

4.5.2 Specification

The interface and properties of a (1,N) regular register in a fail-recovery model, called here a *logged register*, are given in Module 4.4. Logged atomic registers ((1,N) and (N,N)) can be specified accordingly.

The *termination* property is similar to what we considered before, though expressed here in a different manner. Indeed the notion of correctness used in earlier register specifications has a different meaning here. It does not make much sense to require that a process that invokes some operation, crashes, and then recovers, still gets back a reply to the operation. Our *termination* property requires however that if a process invokes an operation and does not crash, it eventually gets a reply.

On the the other hand, the *validity* property is expressed as in earlier specifications but now has a different meaning. Assume the writer p_1 crashes before completing the write of some value X (no previous write was invoked before), then recovers and invokes the writing of value Y. Assume that p_2 concurrently invokes a read operation on the same register. It is valid that this read operation returns 0: value X is not considered to have been written. Now assume that p_2 invokes another read operation that is still concurrent with the writing of Y. It is not valid for p_2 to return 5. In other words, there is only one last value written before 6: this can be 0 or 5, but not both of them.

4.5.3 Fail-Recovery Algorithm: Logged Majority Voting

Considering a fail-recovery model where all processes can crash, it is easy to see that even a $(1, 1)$ regular register algorithm cannot be implemented unless the processes have access to stable storage and a majority is correct. We thus make the following assumptions.

1. Every process has access to a local stable storage. This is supposed to be accessible through the primitives *store*, which atomically logs information in the stable storage, and *retrieve*, which gets back that information from the storage. Information that is logged in the stable storage is not lost after a crash.
2. A majority of the processes are correct. Remember that a correct process in a fail-recovery model is one that either never crashes, or eventually recovers and never crashes again.

Intuitively, we might consider transforming our majority voting regular register (i.e., Algorithm 4.2) to deal with process crashes and recoveries simply by logging every new value of any local variable to stable storage, upon modification of that variable, and then retrieving all variables upon recovery. This would include messages to be delivered, i.e., the act of delivering a message would coincide with the act of storing it in stable storage. However, and as we discussed earlier in this manuscript, one should be careful with such an automatic transformation because access to stable storage is an expensive operation and should only be used when necessary.

In particular, we describe in Algorithm 4.11 an implementation of a $(1, N)$ regular register that logs the variables that are persistent across invocations (e.g., the value of the register at a given process and the timestamp), in one atomic operation, and retrieve these variables upon recovery. We discuss the need of this atomicity, through an exercise given at the end of the chapter.

The algorithm makes use of stubborn communication channels and stubborn broadcast communication abstractions. Remember that stubborn communication primitives ensure that if a message is sent to a correct process (even in the fail-recovery sense), the message is delivered an infinite number of times, unless the sender crashes. This ensures that the process, even if it crashes and recovers a finite number of times, will eventually process every message sent to it.

Note that upon recovery, every process first executes its initialization procedure and then its recovery one. Note also that we do not log the variables that are only persistent across events, e.g., the variable that counts the number of acknowledgements that a writer has for instance received. The communication pattern of Algorithm 4.11 is similar to the one of the majority-voting regular register algorithm for the fail-silent model (Algorithm 4.2). What we furthermore add here are logs. For every write operation, the writer logs the new timestamp and the value to be written, then a majority of the processes log the new value with its timestamp.

Correctness. The *termination* property follows from the properties of the underlying stubborn communication abstractions and the assumption of a majority of correct processes.

Consider now *validity.* Consider first the case of a read that is not concurrent with any write. Assume furthermore that a read is invoked by some process p_i and the last value written by p_1, say v, has timestamp sn_1 at p_1. Because the writer logs every timestamp and increments the timestamp for every write, at the time when the read is invoked, a majority of the processes have logged v and timestamp sn_1 and there is no higher timestamp in the system. Before reading a value, i.e., returning from the read operation, p_i consults a majority of processes and hence gets at least one value with timestamp sn_1. Process p_i hence returns value v with timestamp sn_1, which is indeed the last value written.

Consider now the case where the read is concurrent with some write of value v and timestamp sn_1, and the previous write was for value v' and timestamp $sn_1 - 1$. If the latter write had failed before p_1 logged v' than no process will ever see v'. Otherwise, p_1 would have first completed the writing of v' upon recovery. If any process returns sn_1 to p_i, p_i will return v, which is a valid reply. Otherwise, at least one process will return $sn_1 - 1$ and p_i will return v', which is also a valid reply.

Performance. Every write operation requires one communication round-trip between p_1 and a majority of the processes and every read requires one communication round-trip between the reader process and a majority of the processes. In both cases, at most $2N$ messages are exchanged. Every write requites one log at p_1 and then at least a majority of *logs* (possibly parallel ones). Thus, every write requites two causally related logs. It is important to notice that stubborn channels are implemented by retransmitting messages periodically, and this retransmission can be stopped by a writer and a reader

Algorithm 4.11 Majority voting (1,N) logged regular register algorithm.

```
Implements:
    (1,N)LoggedRegularRegister (on-logrreg)
Uses:
    StubbornBestEffortBroadcast (sbeb). StubbornPointToPointLinks (sp2p).

upon event ⟨ Init ⟩ do
    forall $r$ do
        ts[$r$] := sn[$r$] := v[$r$] := acks[$r$] := reqid[$r$] := 0;
        readSet[$r$] := $\emptyset$; writing[$r$] = false;

upon event ⟨ Recovery ⟩ do
    retrieve (ts, sn, v, writing);
    forall $r$ do
        if writing[$r$] $\neq 0$ then
            acks[$r$] = 0; reqid[$r$] := reqid[$r$] + 1;
            trigger ⟨ sbebBroadcast, [WRITE, r, reqid[$r$], ts[$r$], v[$r$]] ⟩;

upon event ⟨ on-logrregWrite, r, val ⟩ do
    acks[$r$] := 0; reqid[$r$] := reqid[$r$] + 1; ts[$r$] := ts[$r$] + 1; writing[$r$] := true
    store (ts[$r$], v[$r$], writing[$r$]);
    trigger ⟨ sbebBroadcast, [WRITE, r, reqid[$r$], ts[$r$], v[$r$]] ⟩;

upon event ⟨ sbebDeliver, $p_j$, [WRITE, r, id, t, val] ⟩ do
    if t > sn[$r$] then
        v[$r$] := val; sn[$r$] := t; store(sn[$r$], v[$r$]);
    trigger ⟨ sbp2pSend, $p_j$, [ACK, r, id] ⟩;

upon event ⟨ sbp2pDeliver, $p_j$, [ACK, r, id] ⟩ do
    if id = reqid[$r$] then acks[$r$] := acks[$r$] + 1;

upon exists $r$ such that acks[$r$] > $N/2$ do
    writing[$r$] = false; trigger ⟨ on-logrregWriteReturn, r ⟩; store (writing[$r$]);

upon event ⟨ on-logrregRead, r ⟩ do
    reqid[$r$] := reqid[$r$] + 1; readSet[$r$] := $\emptyset$;
    trigger ⟨ sbebBroadcast, [READ, r, id] ⟩;

upon event ⟨ sbebDeliver, $p_j$, [READ, r, id] ⟩ do
    trigger ⟨ sp2pSend, $p_j$,[READVALUE, r, id, sn[$r$], v[$r$]] ⟩;

upon event ⟨ sp2pDeliver, $p_j$, [READVALUE, r, id, snb, val] ⟩ do
    if id = reqid[$r$] then readSet[$r$] := readSet[$r$] $\cup$ { (snb, val) };

upon exists $r$ such that | readSet[$r$]| > $N/2$ do
    trigger ⟨ on-logrregReadReturn, r, highest(readSet[$r$]) ⟩;
```

that receives a reply of some process or receives enough replies to complete its operation.

Interestingly, Algorithm 4.10 and Algorithm 4.6 extend Algorithm 4.11 to implement respectively a (1,N) and a (N,N) atomic registers in a fail-recovery model.

Hands-On

To-be-done. To-be-done.

To-be-done. To-be-done.

To-be-done. To-be-done.

To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done.

Exercises

Exercise 4.1 *Explain why every process needs to maintain a copy of the register value in the read-one-write-all algorithm (Algorithm 4.1) as well as in the majority-voting algorithm (Algorithm 4.2).*

Exercise 4.2 *Use the idea of the tranformation from (1,N) regular to (1,1) atomic registers (Algorithm 4.3) to adapt the read-one-write-all algorithm (i.e., Algorithm 4.1) to implement a (1,1) Atomic Register.*

Exercise 4.3 *Use the idea of the tranformation from (1,N) regular to (1,1) atomic registers (Algorithm 4.3) to adapt the majority voting algorithm (Algorithm 4.2) to implement a (1,1) Atomic Register.*

Exercise 4.4 *Explain why a timestamp is needed in the majority-voting algorithm (Algorithm 4.2) but not in the read-one-write-all algorithm (Algorithm 4.1).*

Exercise 4.5 *Explain why, in Algorithm 4.12, the reader* p_2 *needs always include its own value and timestamp when selecting a majority.*

Exercise 4.6 *Does any implementation of a regular register require a majority of correct processes in an asynchronous system? What if an eventually perfect failure detector is available?*

Exercise 4.7 *Explain why, in Algorithm 4.11 for instance, if the store primitive is not atomic, it is important not to log the timestamp without having logged the value. What if the value is logged without having logged the timestamp.*

Exercise 4.8 *Explain why in Algorithm 4.11, the writer needs to store its timestamp in stable storage.*

Solutions

Solution 4.1 We consider each algorithm separately.

Algorithm 4.1. In this algorithm, a copy of the register value needs to be stored at every process because we assume that any number of processes can crash and any process can read. Indeed, assume that the value is not stored at some process p_k. It is easy to see that after some write operation, all processes might crash except p_k. In this case, there is no way for p_k to return the last value written.

Algorithm 4.2. In this algorithm, a copy of the register value needs also to be maintained at all processes, even if we assume only one reader. Assume that some process p_k does not maintain a copy. Assume furthermore that the writer updates the value of the register: it can do so only by accessing a majority. If p_k is in that majority, then the writer would have stored the value in a majority of the processes minus one. It might happen that all processes in that majority, except p_k, crash: the rest of the processes plus p_k also constitutes a majority. A subsequent read in this majority might not get the last value written. □

Solution 4.8 The read-one-write-all algorithm (i.e., Algorithm 4.1) does not need to be transformed to implement an atomic register if we consider only *one* reader: indeed the scenario of Figure 4.6, which violates *ordering* involves two readers. As is, the algorithm implements a $(1, 1)$ atomic register where any process can write and one specific process, say p_2, can read. In fact, if we assume a single reader, say p_2, the algorithm can even be optimized in such a way that the writer does simply try to store its value in p_2, and gives up if the writer detects the crash of p_2. Basically, only the reader p_2 needs to maintain the register's value and the writer p_1 would not need to send a message to all. □

Solution 4.3 Consider now our majority voting algorithm, i.e., Algorithm 4.2. This algorithm does not implement a (1,1) atomic register but can easily be extended to satisfy the *ordering* property by adding a simple local computation at the reader p_2. It suffices indeed for p_2 to update its value and timestamp whenever p_2 selects the value with the highest timestamp before returning it. Then p_2 has simply to make sure that it includes its own value in the set from which it selects new values. The scenario of Figure 4.7 occurs precisely because the reader has no memory of the previous value read.

A solution is depicted in Algorithm 4.12, an extension of Algorithm 4.2 that implements a (1,1) atomic register. We assume here that the reader includes itself in every read majority. Note that in this case, we assume that the function *select* returns a pair *(timestamp,value)* (with the highest timestamp), rather than simply a value. With this algorithm, the scenario of Figure 4.7 cannot happen, whereas the scenario depicted in Figure 4.8 could. As

Algorithm 4.12 Majority-voting (1,1) atomic register algorithm.

```
Implements:
    (1,1)AtomicRegister (oo-areg).
Uses:
    BestEffortBroadcast (beb).
    perfectPointToPointLinks (pp2p).

upon event ⟨ Init ⟩ do
    forall $r$ do
        ts[$r$] := sn[$r$] := v[$r$] := acks[$r$] := reqid[$r$] := 0; $readSet[r] := \emptyset$;

upon event ⟨ oo-aregWrite, r, val ⟩ do
    ts[$r$] := ts[$r$] + 1; acks[$r$] := 0; reqid[$r$] := reqid[$r$] + 1;
    trigger ⟨ bebBroadcast, [WRITE, r, reqid[$r$], ts[$r$], val] ⟩;

upon event ⟨ bebDeliver, $p_j$, [WRITE, r, id, tstamp, val] ⟩ do
    if tstamp > sn[$r$] then
        v[$r$] := val; sn[$r$] := tstamp;
        trigger ⟨ pp2pSend, $p_j$, [ACK, r,id] ⟩;

upon event ⟨ pp2pDeliver, $p_j$, [ACK, r, id] ⟩ do
    if id=reqid[$r$] then acks[$r$] := acks[$r$] + 1;

upon exists $r$ such that acks[$r$] > $N/2$ do
    trigger ⟨ oo-aregWriteReturn, r ⟩;

upon event ⟨ oo-aregRead, r ⟩ do
    readSet[$r$] := ∅; reqid[$r$] := reqid[$r$] +1;
    trigger ⟨ bebBroadcast, [READ, r, reqid[$r$]] ⟩;

upon event ⟨ bebDeliver, $p_j$, [READ, r, id] ⟩ do
    trigger ⟨ pp2pSend, $p_j$,[READVALUE, r, id, sn[$r$], v[$r$]] ⟩;

upon event ⟨ pp2pDeliver, $p_j$, [READVALUE, r, id, snb,val] ⟩ do
    if id=reqid[$r$] then readSet[$r$] := readSet[$r$] ∪ $\{(snb, val)\}$;

upon exists $r$ such that | readSet[$r$]| > $N/2$ do
    $(v, ts)$ := highest(readSet[$r$]); $v[r] := v$; $sn[r] := ts$;
    trigger ⟨ oo-aregReadRetutn, r, $v$ ⟩;
```

in the original majority voting algorithm (Algorithm 4.2), every write operation requires one communication round-trip between p_1 and a majority of the processes and every read requires one communication round-trip between p_2 and a majority of the processes. In both cases, $2N$ messages are exchanged. □

Solution 4.4 The timestamp of Algorithm 4.2 is needed precisely because we do not make use of a perfect failure detector. Without the use of any timestamp, p_2 would not have any means to compare different values from the accessed majority. In particular, if p_1 writes a value v and then a value v', and does not access the same majority in both cases, p_2, which is supposed to return v', might not see which one is the latest. Such a timestamp is not needed in Algorithm 4.1, because the writer accesses all processes that did not crash. The writer can do so because of its relying on a perfect failure detector. □

Solution 4.5

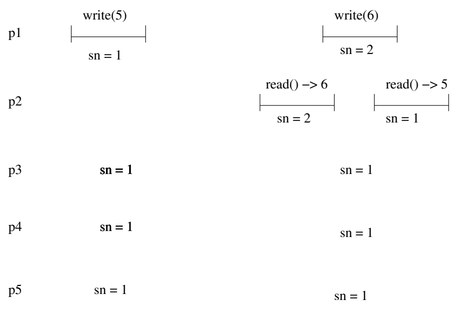

Figure 4.9. Violation of ordering

Unless it includes its own value and timestamp when selecting a majority, the reader p_2 might violate the ordering property as depicted in the scenario of Figure 4.9. This is because, in its first read, p_2 accesses the writer, p_1, which has the latest value, and in its second read, it accesses a majority with timestamp 1 and old value 5. □

Solution 4.6 The argument we use here is a *partitioning* argument and it is similar to the argument used earlier in this manuscript to show that uniform reliable broadcast requires a majority of correct processes even if the system is augmented with an eventually perfect failure detector.

We partition the system into two disjoint sets of processes X and Y, such that $\mid X \mid \lceil n/2 \rceil$: p_1, the writer of the register, is in X and p_2, the reader of the register, is in Y. The assumption that there is no correct majority means here that there are runs where all processes of X crash and runs where all processes of Y crash.

Basically, the writer p_1 might return from having written a value, say v, even if none of the processes in Y has witnessed this value. The other processes, including p_2, were considered to have crashed, even if they were not. If the processes of X, and which might have witnessed v later, crash, the reader, p_2, has no way of knowing about v and might not return the last value written.

Assuming an eventually perfect detector does not help. This is because, even with such a failure detector, the processes of X, including the writer p_1

might return from having written a value, say v, even if no process in Y has witnessed v. The processes of Y have been falsely suspected and there is no way to know whether the suspicions are true or false. □

Solution 4.7 Assume p_1 writes a value v, then a value v', and then a value v''. While writing v, assume p_1 accesses some process p_k and not p'_k whereas, while writing v', p_1 accesses p'_k and not p_k. While writing v'', p_1 also accesses p_k which logs first the timestamp and then crashes without logging the associated value, then recovers. When reading, process p_2 might select the old value v because it has a higher timestamp, violating validity.

On the other hand, logging the timestamp without logging the value is not necessary (although desirable to minimize accesses to stable storage). In the example depicted above, p_2 would not be able to return the new value because it still has an old timestamp. But that is okay because the value was not completely written and there is no obligation to return it. □

Solution 4.8 The reason for the writer to log its timestamp in Algorithm 4.11 is the following. If it crashes and recovers, the writer should not use a smaller timestamp than the one associated with the current value of the register. Otherwise, the reader might return an old value and violates the validity property of the register. □

Historical Notes

- Register specifications were first given in (Lamport 1977; Lamport 1986a; Lamport 1986b), for the case of a concurrent system with one writer. The original notion of atomic register was similar to the one we introduced here. There is slight difference however in the way we gave our definition because we had to take into account the possibility for the processes to fail (independently of each other). We had thus to deal explicitly with the notion of failed operations (in particular failed write). The original definition was given in the context of a multiprocessor machine where processes do not fail independently.
- In the fail-stop model, our notion of atomicity is similar to the notion of linearizability introduced in (Herlihy and Wing 1990). In the fail-recovery model, we had to consider a slightly different notion to take into account the fact that a write operation that was interrupted by a failure has to appear as it was never invoked or if it was terminated before the next invocation of the same process (which might have recovered) take place.
- Our notion of regular register also corresponds to the notion of *regular* register initially introduced in (Lamport 1977; Lamport 1986a; Lamport 1986b). For the case of multiple-writers the notion of regular register was generalized in three different ways in (Shao, Pierce, and Welch 2003), all are stronger than our notion of regular register.
- Various forms of register transformations were given in (Vitanyi and Awerbuch 1986; Vidyasankar 1988; Vidyasankar 1990; Israeli and Li 1993).
- In this chapter, we considered registers that can contain any integer value and did not make any assumption on the possible range of this value. In (Lamport 1977), registers with values of a limited range were considered, i.e., the value in the register cannot be greater than some specific value V. In (Lamport 1977; Peterson 1983; Vidyasankar 1988), several transformation algorithms were described to emulate a register with a given range value into a register with a larger range value.
- Fail-silent register implementations over a crash-stop message passing system and assuming a correct majority were first given in (Attiya, Bar-Noy, and Dolev 1995) for the case of a single writer. They were then generalized for the case of multiple writers in (Lynch and Shvartsman 1997; Lynch and Shvartsman 2002).
- Failure detection lower bounds for registers were given in (Delporte-Gallet, Fauconnier, and Guerraoui 2002).
- Implementation of registers when processes can crash and recover were given in (Boichat, Dutta, Frolund, and Guerraoui 2001; Guerraoui and Levy 2004).

5. Consensus

Life is what happens to you while you are making other plans.
(John Lennon)

This chapter considers the *consensus* abstraction. The processes use this abstraction to agree on a common value out of values they initially propose. We consider four variants of this abstraction: *regular, uniform, logged* and *randomized.*

We will show later in this manuscript (Chapter 6 and Chapter 7) how consensus abstractions can be used to build more sophisticated forms of agreements.

5.1 Regular Consensus

5.1.1 Specification

Consensus (sometimes we say regular consensus) is specified in terms of two primitives: *propose* and *decide.* Each process has an initial value that it proposes for the agreement, through the primitive *propose.* The proposed values are private to the processes and the act of proposing is local. This act typically triggers broadcast events through which the processes exchange their proposed values in order to eventually reach agreement. All correct processes have to decide on a single value, through the primitive *decide.* This decided value has to be one of the proposed values. Consensus must satisfy the properties C1–4 listed in Module 5.1.

In the following, we present two different algorithms to implement consensus. Both algorithms are fail-stop: they rely on a perfect failure detector abstraction. The first algorithm uses a small number of communication steps but a large number of messages. The second, on the other hand, uses less messages but a large number of communication steps.

Module:

Name: (regular) Consensus (c).

Events:

Request: ⟨ *cPropose*, v ⟩: Used to propose a value for consensus.

Indication: ⟨ *cDecide*, v ⟩: Used to indicate the decided value for consensus.

Properties:

C1: *Termination:* Every correct process eventually decides some value.

C2: *Validity:* If a process decides v, then v was proposed by some process.

C3: *Integrity:* No process decides twice.

C4: *Agreement:* No two correct processes decide differently.

Module 5.1 Interface and properties of consensus.

5.1.2 Fail-Stop Algorithm: Consensus with Flooding

Algorithm 5.1 uses, besides a perfect failure detector, a best-effort broadcast communication abstraction.

The basic idea of the algorithm is the following. The processes follow sequential rounds. Each process keeps the set of proposed values (proposals) it has seen, and this set is typically augmented when moving from a round to the next (and new proposed values are known). In each round, every process disseminates its own set to all processes using the best-effort broadcast abstraction, i.e., the process floods the system with all proposals it has seen in previous rounds. When a process receives a proposal set from another process, it merges this set with its own. That is, in each round, every process computes the union of all sets of proposed values it received so far. Roughly speaking, a process decides a specific value in its set when it knows it has gathered all proposals that will ever possibly be seen by any correct process. We explain in the following (1) when a round terminates and a process moves from a round to the next, (2) when a process knows it is safe to decide, and (3) how a process selects the value to decide.

1. Every message is tagged with the round number in which the message was broadcast. A round terminates, at a given process p_i, when p_i has received a message from every process that has not been suspected by p_i in that round. That is, a process does not leave a round unless it receives messages, tagged with that round, from all processes that have not been suspected to have crashed in that round.
2. A consensus decision is reached when a process knows it has the same set of proposed values as all correct processes. In a round where a new failure is detected, a process p_i is not sure of having exactly the same set of values as the other processes. This might happen because the crashed

Algorithm 5.1 A flooding consensus algorithm.

```
Implements:
    Consensus (c);

Uses:
    BestEffortBroadcast (beb);
    PerfectFailureDetector ($\mathcal{P}$);

upon event ⟨ Init ⟩ do
    correct := correct-this-round[0] := $\Pi$;
    decided := $\perp$; round := 1;
    for $i = 1$ to $N$ do
        correct-this-round[i] := $\emptyset$;
        proposal-set[i] := $\emptyset$;

upon event ⟨ crash, $p_i$ ⟩ do
    correct := correct $\setminus \{p_i\}$;

upon event ⟨ cPropose, $v$ ⟩ do
    proposal-set[round] := $\{v\}$;
    trigger ⟨ bebBroadcast, [MySet, round, proposal-set] ⟩;

upon event ⟨ bebDeliver, $p_i$, [MySet, r, set] ⟩ do
    correct-this-round[r] := correct-this-round[r] $\cup \{p_i\}$;
    proposal-set[r] := proposal-set[r] $\cup$ set;

upon correct $\subset$ correct-this-round[round] $\wedge$ (decided = $\perp$) do
    if (correct-this-round[round] = correct-this-round[round-1]) then
        decided := min (proposal-set[round]);
        trigger ⟨ cDecide, decided ⟩;
        trigger ⟨ bebBroadcast, [Decided, decided] ⟩;
    else
        round := round +1;
        trigger ⟨ bebBroadcast, [MySet, round, proposal-set[round-1]] ⟩;

upon event ⟨ rbDeliver, $p_i$, [Decided, v] ⟩ $\wedge$ $p_i \in$ correct $\wedge$ (decided = $\perp$) do
    decided := v; trigger ⟨ cDecide, v ⟩;
    trigger ⟨ bebBroadcast, [Decided, decided] ⟩;
```

process(es) may have broadcast some values to the other processes but not to p_i. In order to know when it is safe to decide, each process keeps a record of the processes it did not suspect in the previous round, and from how many processes it has received a proposal in the current round. If a round terminates with the same number of non-suspected processes as in the previous round, a decision can be made. In a sense, the messages broadcast by all processes that moved to the current round did all reach their destination.

3. To make a decision, a process can then apply any deterministic function to the set of accumulated values, provided this function is the same at all

processes (i.e., it is agreed upon by all processes in advance). In our case, the process decides the minimum value: we implicitely assume here that the set of all possible proposals is totally ordered and the order is known by all processes. (The processes could also pick the value proposed by the process with the lowest identity for instance.) A process that decides, simply disseminates the decision to all processes using the best-effort broadcast abstraction.

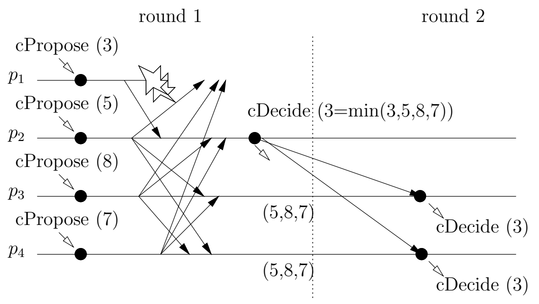

Figure 5.1. Sample execution of the flooding consensus algorithm.

An execution of the algorithm is illustrated in Figure 5.1. Process p_1 crashes during the first round after broadcasting its proposal. Only p_2 sees that proposal. No other process crashes. Therefore, p_2 sees the proposals of all processes and can decide. This is because the set of processes from which it receives proposals in the first round is the same as the initial set of processes which start the algorithm. Process p_2 takes the *min* of the proposals and decides the value 3. Processes p_3 and p_4 detect the crash of p_1 and cannot decide. So they advance to the next round, namely round 2. Note that if any of these processes decided the *min* of the proposals it had after round 1, they would have decided differently, i.e., value 5. Since p_2 has decided, p_2 disseminates its decision through a best-effort broadcast. When the decision is delivered, processes p_3 and p_4 also decide 3.

Correctness. Validity and *integrity* follow from the algorithm and the properties of the communication abstractions. *Termination* follows from the fact that at round N at the latest, all processes decide. *Agreement* is ensured because the *min* function is deterministic and is applied by all correct processes on the same set.

Performance. If there are no failures, the algorithm requires a single communication step: all processes decide at the end of round 1, if no process is suspected to have crashed. Each failure may cause at most one additional communication step. Therefore, in the worst case the algorithm requires N steps, if $N-1$ processes crash in sequence. If there are no failures, the algo-

Algorithm 5.2 A hierarchical consensus algorithm.

```
Implements:
    Consensus (c);

Uses:
    BestEffortBroadcast (beb);
    PerfectFailureDetector ($\mathcal{P}$);

upon event ⟨ Init ⟩ do
    suspected := $\emptyset$; round := 1;
    proposal := nil; prop-round :=0;
    for i = 1 to $N$ do
        delivered[$i$] := broadcast[$i$] := false;

upon event ⟨ crash, $p_i$ ⟩ do
    suspected := suspected $\cup$\{ rank($p_i$) \};

upon event ⟨ cPropose, $v$ ⟩ do
    proposal := $v$;

upon (round = rank (self)) $\wedge$ (proposal $\neq$ nil) $\wedge$ (broadcast[round] = false) do
    broadcast[round] := true;
    trigger ⟨ cDecide, proposal ⟩;
    trigger ⟨ bebBroadcast, [DECIDED, round, proposal] ⟩;

upon (round $\in$ suspected) $\vee$ (delivered[round] = true) do
    round := round + 1;

upon event ⟨ bebDeliver, $p_i$, [DECIDED, r,v] ⟩ do
    if (r < rank (self)) $\wedge$ (r > prop-round) then
        proposal := v; prop-round := r;
    delivered[$r$] := true;
```

rithm exchanges $2N^2$ messages. There is an additional N^2 message exchanges for each round where a process crashes.

5.1.3 Fail-Stop Algorithm: Hierarchical Consensus

Algorithm 5.2 is an alternative way to implement regular consensus. This algorithm is interesting because it uses less messages than our flooding algorithm and enables one process to decide before exchanging any messages with the rest of the processes (0-latency). However, to reach a global decision, where all processes decide, the algorithm requires N communication steps, even in situations when no failure occurs. Algorithm 5.2 is particularly adequate if consensus is used as a service implemented by a set of server processes where the clients are happy to get a value as fast as possible, even if the servers did not all decide that value yet.

Algorithm 5.2 makes use of the fact that processes can be ranked according to their identity, and this rank is used to totally order them, a priori, i.e., $p_1 > p_2 > p3 > .. > p_N$. In short, the algorithm ensures that the correct process with the highest rank in the hierarchy, i.e., the process with the lowest identity, imposes its value on all the other processes. Basically, if p_1 does not crash, then p_1 will impose its value to all: all correct processes will decide the value proposed by p_1. If p_1 crashes initially and p_2 is correct, then the algorithm ensures that p_2's proposal will be decided. A non-trivial issue that the algorithm handles is the case where p_1 is faulty, but does not initially crash, whereas p_2 is correct. The issue has to do with the fact that p_1's decision message might only reach process p_3 but not p_2.

Algorithm 5.2 also works in rounds and uses a best effort broadcast abstraction. In the kth round, process p_k decides its proposal, and broadcasts it to all processes: all other processes in round k wait to deliver the message of p_k or to suspect p_k. None of these processes broadcast any message in this round. When a process p_k receives the proposal of p_i, in round $i < k$, p_k adopts this proposal as its own new proposal.

Consider the example depicted in Figure 5.2. Process p_1 decides 3 and broadcasts its proposal to all processes, but crashes. Processes p_2 and p_3 detect the crash before they deliver the proposal of p_1 and advance to the next round. Process p_4 delivers the value of p_1 and changes its own proposal accordingly, i.e., p_4 adopts p_1's value. In round 2, process p_2 decides and broadcasts its own proposal. This causes p_4 to change its proposal again, i.e., now p_4 adopts p_2's value. From this point on, there are no further failures and the processes decide in sequence the same value, namely p_2's value (5).

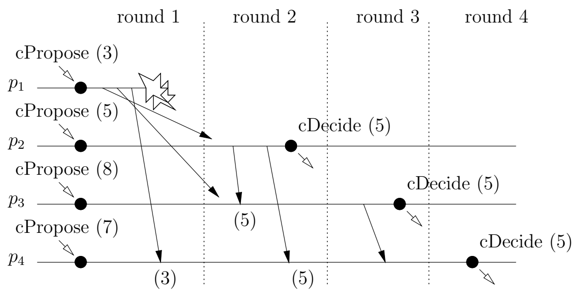

Figure 5.2. Sample execution of hierarchical consensus.

Correctness. The *validity* and *integrity* properties follow from the algorithm and the use of an underlying best effort broadcast abstraction. *Termination* follows from the *completeness* property of the perfect failure detector and the *validity* property of best effort broadcast: no process will remain indefinitely blocked in a round and every correct process p_i will eventually reach round i and decide in that round. Concerning *agreement*, let p_i be the correct process

Module:

Name: UniformConsensus (uc).

Events:

$\langle$ *ucPropose*, v $\rangle$, $\langle$ *ucDecide*, v $\rangle$: with the same meaning and interface of the consensus interface.

Properties:

C1-C3: from consensus.

C4': *Uniform Agreement:* no two processes decide differently..

Module 5.2 Interface and properties of uniform consensus.

with the highest rank which decides some value v. By the algorithm, every process p_j, such that $j > i$, decides v: no process will suspect p_i because p_i is correct. This is guaranteed by the *accuracy* property of the perfect failure detector. Hence, every process will adopt and decides p_i's decision.

Performance. The algorithm exchanges $(N-1)$ messages in each round and can clearly be optimized such that it exchanges only $N(N-1)/2$ messages: a process does not need to send a message to processes with a higher rank. The algorithm also requires N communication steps to terminate.

5.2 Uniform Consensus

5.2.1 Specification

As with (regular) reliable broadcast, we can define a uniform variant of consensus. The uniform specification is presented in Module 5.2: correct processes decide a value that must be consistent with values decided by processes that might have decided before crashing. In short, uniform consensus ensures that no two processes decide different values, whether they are correct or not.

None of the consensus algorithms we presented so far ensure uniform agreement. Roughly speaking, this is because some of the processes decide too early: without making sure that their decision has been seen by enough processes. Should the deciding processes crash, other processes might have no choice but to decide something different. To illustrate this for our hierarchical consensus algorithm, i.e., Algorithm 5.2, remember that process p_1 decides its own proposal in a unilateral way without making sure its proposal is seen by any other process. Hence, if process p_1 crashes immediately after deciding, it is likely that the other processes decide a different value. To illustrate this for our flooding consensus algorithm, i.e., Algorithm 5.1, consider a scenario where process p_1 , at the end of round 1, receives messages from all processes. Assume furthermore that p_1 decides its own value as this turns out to be the lowest value. Assume however that p_1 crashes after deciding and its message does not reach any other process. The rest of the processes move to round 2 without having received p_1's message. Again, the processes are likely to decide some other value.

In the following, we present two uniform consensus algorithms for the fail-stop model. Each algorithm can be viewed as a uniform variant of one of our regular consensus algorithms above. The first algorithm is a flooding uniform consensus algorithm whereas the second is a hierarchical uniform consensus algorithm. Subsecquently, we present also an algorithm for the fail-noisy model.

5.2.2 Fail-Stop Algorithm: Uniform Consensus with Flooding

Algorithm 5.3 implements uniform consensus. The processes follow sequential rounds. As in our flooding regular consensus algorithm, each process gathers a set of proposals that it has seen and disseminates its own set to all processes using a best-effort broadcast primitive. An important difference with Algorithm 5.3 is that all processes wait for round N before deciding.

Correctness. *Validity* and *integrity* follow from the algorithm and the properties of best-effort broadcast. *Termination* is ensured here because all correct processes reach round N and decide in that round. *Uniform agreement* is ensured because all processes that reach round N have the same set of values.

Performance. The algorithm requires N communication steps and $N(N-1)^2$ messages for all correct processes to decide.

5.2.3 Fail-Stop Algorithm: Hierarchical Uniform Consensus

Algorithm 5.4 is round-based, hierarchical, and is in this sense similar to our hierarchical regular consensus algorithm. Algorithm 5.4 uses both a best-effort broadcast abstraction to exchange messages and a reliable broadcast abstraction to disseminate a decision.

Every round has a leader: process p_i is leader of round i. When a process receives a proposal from the leader, it ignores its own initial value and follows the leader. Unlike our hierarchical regular consensus algorithm, however, the leader does not decide immediately. Instead, it waits until all other processes have confirmed the value the leader has proposed. If the leader of a round fails, the correct processes detect this and proceeds to the next round. The leader is consequently changed.

Correctness. *Validity* and *integrity* follow trivially from the algorithm and the properties of the underlying communication abstractions. Consider *termination.* If some correct process decides, it decides through the reliable broadcast abstraction, i.e., by rbDelivering a decision message. By the properties of this broadcast abstraction, every correct process rbDelivers the decision message and decides. Hence, either all correct processes decide or no correct process

Algorithm 5.3 A flooding uniform consensus algorithm.

```
Implements:
    UniformConsensus (c);

Uses:
    BestEffortBroadcast (beb).
    PerfectFailureDetector ($\mathcal{P}$);

upon event ⟨ Init ⟩ do
    correct := $\Pi$; round := 1; decided := $\perp$; proposal-set := $\emptyset$;
    for i = 1 to $N$ do delivered[$i$] := $\emptyset$;

upon event ⟨ crash, $p_i$ ⟩ do
    correct := correct \{$p_i$};

upon event ⟨ ucPropose, $v$ ⟩ do
    proposal-set := {$v$};
    trigger ⟨ bebBroadcast, [MySet, round, proposal-set] ⟩;

upon event ⟨ bebDeliver, $p_i$, [MySet, r, newSet] ⟩ do
    proposal-set := proposal-set $\cup$ newSet;
    delivered[r] := delivered[r] $\cup$ {$p_i$};

upon (correct $\subseteq$ delivered[round]) $\wedge$ (decided = $\perp$) do
    if round = N then
        decided := min (proposal-set);
        trigger ⟨ ucDecide, decided) ⟩;
    else
        round := round + 1;
        trigger ⟨ bebBroadcast, [MySet, round, proposal-set] ⟩;
```

decides. Assume by contradiction that there is at least one correct process and no correct process decides. Let p_i be the correct process with the highest rank. By the *completeness* property of the perfect failure detector, every correct process suspects the processes with higher ranks than p_i (or bebDelivers their message). Hence, all correct processes reach round i and, by the *accuracy* property of the failure detector, no process suspects process p_i or moves to a higher round, i.e., all correct processes wait until a message from p_i is bebDelivered. In this round, process p_i hence succeeds in imposing a decision and decides. Consider now *agreement* and assume that two processes decide differently. This can only be possible if two processes rbBroadcast two decision messages with two propositions. Consider any two processes p_i and p_j, such that $j > i$ and p_i and p_j rbBroadcast two decision values v and v'. Because of the accuracy property of the failure detector, process p_j must have adopted v before reaching round j.

Performance. If there are no failures, the algorithm terminates in 3 communication steps: 2 steps for the first round and 1 step for the reliable broadcast.

Algorithm 5.4 A hierarchical uniform consensus algorithm.

```
Implements:
      UniformConsensus (uc);

Uses:
      PerfectPointToPointLinks (pp2p);
      ReliableBroadcast (rb).
      BestEffortBroadcast (beb).
      PerfectFailureDetector ($\mathcal{P}$);

upon event ⟨ Init ⟩ do
      proposal := decided := $\perp$; round := 1;
      suspected := ack-set := $\emptyset$;
      prop-round := 0;

upon event ⟨ crash, $p_i$ ⟩ do
      suspected := suspected $\cup$ { $rank(p_i)$ };

upon event ⟨ ucPropose, $v$ ⟩ do
      proposal := $v$;

upon (round = $rank$(self)) $\wedge$ (proposal $\neq \perp$) $\wedge$ (decided = $\perp$) do
      trigger ⟨ bebBroadcast, [PROPOSE, round, proposal] ⟩;

upon event ⟨ bebDeliver, $p_i$, [PROPOSE, r, v] ⟩do
      ack-set := ack-set $\cup$ { $rank(p_i)$ };
      if r < $rank$(self) $\wedge$ r < prop-round then
            proposal := v; prop-round := r;

upon event (round $\in$ suspected) $\vee$ (round $\in$ ack-set) do
      round := round + 1;

upon event (ack-set $\cup$ suspected = $\Pi$) do
      trigger ⟨ rbBroadcast, [DECIDED, proposal] ⟩;

upon event ⟨ rbDeliver, $p_i$, [DECIDED, v] ⟩ $\wedge$ (decided = $\perp$) do
      decided := v;
      trigger ⟨ ucDecide, v ⟩;
```

It exchanges $3(N-1)$ messages. Each failure of a leader adds 2 additional communication steps and $2(N-1)$ additional messages.

5.2.4 Fail-Noisy Algorithm: Carefull Leader Algorythm

The consensus and uniform consensus algorithms we have given so far are all fail-stop: they rely on the assumption of a perfect failure detector. It is easy to see that, in any of those algorithms, a false failure suspicion (i.e., a violation of the *accuracy* property of the failure detector) might lead to violation of the *agreement* property of consensus (see exercice at the end of this chapter).

That is, if a process is suspected to have crashed whereas the process is actually correct, agreement would be violated and two processes might decide differently. On the other hand, in any of those algorithms, not suspecting a crashed process (i.e., violating the *completeness* property of the failure detector) might lead to violation of the *termination* property of consensus.

In fact, there is no solution to consensus in a fail-silent model if at least one process can crash. Note that this does not mean that a perfect failure detector is always necessary, as we discuss below. In fact, we will provide an algorithm to solve consensus in the fail-noisy model. In this section, we will also show that any fail-noisy algorithm that solves consensus also solves uniform consensus, and no fail-silent algorithm can solve abortable consensus (resp no fail-noisy algorithm can solve consensus) without a correct majority of processes.

Given the complexity of solving consensusin the fail-noisy model, we will use, as a building block, an intermediate abstraction, that we call *abortable consensus.*

Abortable Consensus. Roughly, abortable consensus is weaker variant of consensus that does not require processes to always decide: This consensus variant can be implemented in a fail-silent model, provided a majority of the processes are correct. We later describe how, given such abstraction, uniform consensus can be obtained in a fail-noisy model, i.e., if an eventually perfect failure detector is available. This modular approach helps better understand the various issues underlying solving consensus without a perfect failure detector, i.e., in a fail-noisy model.

Just like consensus, abortable consensus has a single *propose* operation. This operation takes one input parameter, i.e., a proposal for a consensus decision. The operation is also supposed to return a value. Unlike consensus however, the value returned is not necessarily a value that was proposed by some process. It could also be a specific indication $\perp$, meaning that consensus has aborted. It is important to notice that the specific value $\perp$ is not value that could be proposed to consensus. We use the following terminology to define the abortable consensus abstraction.

- When a process invokes the propose operation with v as an argument, we say that the process *proposes v.*
- When a process returns from the invocation with $v \neq \perp$, we say that the process *decides v.*
- When a process returns from the invocation with $\perp$, we say that the process aborts.

Just like with consensus, we require that, when a process decides v, then v was proposed by some process: it cannot be invented out of thin air. Furthermore, once a process has decided a value v, no other process can decide a different value v'. We explain now intuitively when a process can abort and when it has to decide. We will come back to these situations in more details later in the section. Roughly speaking:

Module:

Name: AbortableConsensus (ac).

Events:

Request: ⟨ *acPropose*, v, *tstamp* ⟩: Used to propose a value v with timestamp *tstamp*.

Indication: ⟨ *acDecide*, x ⟩: Used to return x, either a decision value or $\perp$, as a response to the proposition.

Properties:

ER1: *Termination:* Every correct process that proposes eventually decides or aborts.

ER2: *Decision:* If a single process proposes an infinite number of times, it eventually decides.

ER3: *Agreement:* No two processes decide differently.

ER4: *Validity:* Any value decided must have been proposed.

Module 5.3 Interface and properties of abortable consensus.

- A process might abort if another process tries concurrently to propose a value.
- If only one process keeps proposing, it eventually decides. Underlying this idea lies the very fact that abortable consensus is typically an abstraction that processes might (need to) use in a repeated fashion.

Module 5.3 describes the interface and specification of abortable consensus.

Fail-Silent Read-Write Abortable Consensus Algorithm. We describe here a fail-silent algorithm that implements abortable consensus. The algorithm assumes a majority of correct processes (we discuss the need for this majority in the exercice section). We do not make use of any failure detection scheme.

In short, the idea underlying the algorithm is the following. Each process stores an estimate of the proposal value as well as a corresponding timestamp. A process p_i that proposes a value first determines a timestamp to associate with that value: this is simply done by having the process increments its previous timestamp with the value N. Then the process proceeds in two phases: a *read* and then a *write* phase. The aim of the first phase is to check if there already is some estimate of the decision in the system, whereas the aim of the second phase is to reach a decision. Any of these phases can abort, in which case p_i simply stops the algorithm and returns back the abort value $\perp$. The other processes act during these phases as witnesses for p_i.

We describe below the two phases of the algorithm for the process that is proposing a value, say p_i, as well as the tasks of the witness processes.

- **Read**. The aim of this phase, described in Algorithm 5.5, is twofold.

1. First, the aim is for p_i to check, among a majority of witness processes, the estimates already stored in the processes, and the timestamps that those processes have already seen. If any of those timestamps is higher than the timestamp that p_i is proposing, then p_i aborts. Otherwise, p_i selects the value with the higest timestamp, or its own proposal if no such value has been stored, and then proceeds to the *write* phase. We make use here of a function *highest* that returns the estimate value with the highest timestamp among a set of (timestamp, value) pairs.
2. The second aim is for p_i to get a promise from a majority of processes that no other process will succeed in a read or write phase with a lower timestamp.

- **Write**. The aim of this phase, described in Algorithm 5.5, is also twofold.

1. The first aim is for p_i to store an estimate value among a majority of witness processes, and then decide that value. While doing so, p_i might figure out that some process in the majority has seen a higher timestamp than the one p_i is proposing. In this case, p_i simply aborts. Otherwise, p_i decides.
2. The second aim is for p_i to get a promise from a majority of processes that no other process will succeed in a read or write phase with a striclty lower timestamp.

Correctness. The *termination* and *validity* properties follow from the properties of the channels and the assumption of a majority of correct processes.

Consider now the *decision* property. Let p_i be the process that keeps on proposing an infinite number of times, and let t be the time after which no other process proposes a value. Assume by contradiction that no process decides. By the algorithm, p_i keeps on incrementing its timestamp until it gets to a timestamp no process has ever used. By the properties of the channels and the algorithm, there is a time t' higher than t after which p_i decides. A contradiction.

Consider now *agreement.* Let p_i be the process which decides with the smallest timestamp t_i. Assume p_i decides value v_i. By induction on the timestamp, any process that decides with a higher timestamp t_j, does it with v_i. Clearly, $t_j \neq t_i$, otherwise, by the algorithm and the use of a majority, some process will abort the *read* phase of p_i or p_j. Assume the induction property up to some timestamp $t_j > t_i$ and consider $t_j + 1$. By the algorithm, p_j selects the value with the highest timestamp from a majority, and this must be v_i in a majority, with the highest timestamp.

Performance. Every propose operation requires 2 communication round-trips between the process that is proposing a value and a majority of the processes. Hence, at most $4N$ messages are exchanged.

Variant. It is easy to see how our abortable consensus algorithm can be transformed to alleviate the need for a majority of correct processes if a perfect failure detector is available (i.e., in a fail-stop model). Roughly speaking,

Algorithm 5.5 Abortable consensus algorithm

```
Implements:
    AbortableConsensus (ac).

Uses:
    BestEffortBroadcast (beb). PerfectPointToPointLinks (pp2p).

upon event ⟨ Init ⟩ do
    estimate := value := $\perp$;
    readSet := $\emptyset$; wAcks := rstamp := wstamp := 0;

upon event ⟨ acPropose, val, tstamp ⟩ do
    readSet:= $\emptyset$; estimate := val; ts = tstamp;
    trigger ⟨ bebBroadcast, [READ, ts] ⟩;

upon event ⟨ bebDeliver, $p_j$,[READ, t] ⟩ do
    if rstamp $\geq$ t or wstamp $\geq$ t then
        trigger ⟨ pp2pSend, $p_j$,[READNACK] ⟩;
    else
        rstamp := t; trigger ⟨ pp2pSend, $p_j$, [READACK, wstamp, v] ⟩;

upon event ⟨ pp2pDeliver, $p_j$, [READNACK] ⟩ do
    trigger ⟨ acDecide, $\perp$ ⟩;

upon event ⟨ pp2pDeliver, $p_j$, [READACK, t, v] ⟩ do
    readSet := readSet $\cup$ $\{(t, v)\}$

upon (| readSet | > $N/2$) do
    value := highest(readSet); if value $\neq\perp$ then estimate := value;
    wAcks := 0; trigger ⟨ bebBroadcast, [WRITE, ts, estimate] ⟩;

upon event ⟨ bebDeliver, $p_j$, [WRITE, t, v] ⟩ do
    if rstamp > $t$ or wstamp > $t$ then
        trigger ⟨ pp2pSend, $p_j$,[WRITENACK] ⟩;
    else
        value := v; wstamp := t; trigger ⟨ pp2pSend, $p_j$, [WRITEACK] ⟩;

upon event ⟨ pp2pDeliver, $p_j$, [WRITENACK] ⟩ do
    trigger ⟨ acDecide, $\perp$ ⟩;

upon event ⟨ pp2pDeliver, $p_j$, [WRITEACK] ⟩ do
    wAcks := wAcks +1;

upon (|wAcks | > $N/2$) do
    trigger ⟨ acDecide, estimate ⟩;
```

instead of relying on a majority to read and write a value, a process would do so at all processes that it did not suspect to have crashed. Later in this chapter, we will give an algorithm that implements abortable consensus in a fail-recovery model.

Algorithm 5.6 Careful Leader Algorithm.

```
Implements:
    UniformConsensus (uc).

Uses:
    AbortableConsensus (ac);
    BestEffortBroadcast (beb);
    EventualLeaderDetector ($\Omega$).

upon event ⟨ Init ⟩ do
    proposal := $\bot$; leader := proposed := decided := false;
    ts := rank (); // process rank $(1 \ldots N)$

upon event ⟨ trust, $p_i$ ⟩ do
    if $p_i$ = self then leader := true;
    else leader := false;

upon event ⟨ ucPropose, v, ts ⟩ do
    proposal := v;

upon leader $\wedge$ (proposed = false) $\wedge$ (proposal $\neq \bot$) do
    proposed := true; trigger ⟨ acPropose, proposal, ts ⟩;

upon event ⟨ acDecide, result ⟩ do
    if result $\neq \bot$ then trigger ⟨ bebBroadcast, [DECIDED, result] ⟩;
    else proposed := false;
    ts := ts $+N$;

upon event ⟨ bebDeliver, $p_i$, [DECIDED, v] ⟩ $\wedge$ (decided = false) do
    decided := true; trigger ⟨ ucDecide, v ⟩;
```

Fail-Noisy Careful Leader Consensus. Algorithm 5.6 implements uniform consensus. It uses, besides an eventually perfect leader election abstraction and a best-effort broadcast communication abstraction, abortable consensus.

Intuitively, the value that is decided in the consensus algorithm is the value that is decided in the underlying abortable consensus. Two processes that concurrently propose values to abortable consensus might abort. If only one process keeps proposing for sufficiently long however, this process will succeed. This will be ensured in our algorithm by having only leaders propose values. Eventually, only one leader is elected and this will be able to successfully propose a value. Once this is done, the leader broadcasts a message to all processes informing them of the decision.

Correctness. Validity and *integrity* follow from the algorithm and the properties of the underlying communication abstractions. Consider *termination* and assume some process is correct. By the algorithm, only a process that is leader can propose a value to abortable consensus. By the assumption of

Module:

Name: LoggedConsensus (lc).

Events:

Request: ⟨ *lcPropose*, v ⟩: Used to propose a value for logged consensus.

Indication: ⟨ *lcDecide*, v ⟩: Used to indicate the decided value for logged consensus.

Properties:

C1: *Termination:* Unless it crashes, every process eventually decides some value.

C2: *Validity:* If a process decides v, then v was proposed by some process.

C3: *Agreement:* No two processes decide differently.

Module 5.4 Interface and properties of logged consensus.

the underlying eventually perfect leader election, there is a time after which exactly one correct process is eventually elected and remains leader forever. Let p_i be that process. Process p_i will permanently keep on proposing values. By the properties of abortable consensus, p_i will decide and broadcast the decision. By the properties of the best-effort communication primitive, all correct processes eventually deliver the decision message and decide. Consider now agreement and assume that some process p_i decides some value v. This means that v was decided in the underlying abortable consensus. By the properties of abortable consensus, no other process can decide any different value. Any other process p_j that decides, does necessarily decide v.

Performance. We consider here our implementation of abortable consensus assuming a majority of correct processes. If there is a single leader and this leader does not crash, then 4 communication steps and $4(N-1)$ are needed for this leader to decide. Therefore, 5 communication steps and $5(N-1)$ messages are needed for all correct processes to decide.

5.3 Logged Consensus

5.3.1 Specification

We consider here the fail-recovery model and we introduce the logged consensus abstraction in Module 5.4.

5.3.2 Fail-Recovery Algorithm: Logged Careful Reader

As for the fail-noisy model, we will use an auxiliary abstraction, in this case, the abortable logged consensus abstraction.

Module:

Name: LoggedAbortableConsensus (lac).

Events:

Request: $\langle$ *lacPropose*, v $\rangle$: Used to propose a value v.

Indication: $\langle$ *lacDecide*, x $\rangle$: Used to return x, either a decision value or $\perp$, as a response to the proposition.

Properties:

ER1: *Termination:* If a process proposes and does not crash, it eventually decides or aborts.

ER2: *Decision:* If a single correct process proposes an infinite number of times, it eventually decides.

ER3: *Agreement:* No two processes decide differently.

ER4: *Validity:* Any value decided must have been proposed.

Module 5.5 Interface and properties of logged abortable consensus.

Logged Abortable Consensus. The interface and properties of logged abortable consensus are depicted in Module 5.5.

Logged Abortable Consensus Algorithm. We give now an algorithm that implements logged abortable consensus. The algorithm we describe here, depicted in Algorithm 5.7 and 5.8 is similar to the algorithm presented for the fail-silent model (Algorithm 5.5), with three major differences:

1. We use stubborn links and stubborn broadcast instead of perfect-links and best-effort broadcast.
2. We indeed also assume a majority of correct processes but remember that the notion of correct is different in a fail-recovery model: a process is said to be correct in this case if eventually it is permanently up.
3. The updates of the timestamps and estimate values are now logs, i.e., updates on stable storage. The timestamps and estimate values are retreived upon recovery.

Logged Careful Reader Algorithm. Interestingly, assuming a logged abortable consensus instead of abortable consensus, Algorithm 5.6 implements logged consensus (instead of uniform consensus).

5.4 Randomized Consensus

In this section, we discuss how randomization can also be used to solve a slightly weaker variant of consensus without resourcing to a failure detector. This variant of consensus, which we call randomized consensus, ensures integrity, (uniform) agreement and validity properties of (uniform) consensus,

Algorithm 5.7 Logged abortable consensus algorithm (1/2)

```
Implements:
    LoggedAbortableConsensus (lac).

Uses:
    StubbornEffortBroadcast (sbeb). StubbornPointToPointLinks (sp2p).

upon event ⟨ Init ⟩ do
    estimate := value := ⊥; ts := rank (); // process rank (1 . . . N)
    readSet := ∅; wAcks := rstamp := wstamp := 0;

upon event ⟨ Recover ⟩ do
    retrieve (value, rstamp, wstamp);

upon event ⟨ lacPropose, val ⟩ do
    readSet:= ∅; estimate := val; ts := ts +N;
    trigger ⟨ sbebBroadcast, [READ, ts] ⟩;

upon event ⟨ sbebDeliver, $p_j$,[READ, t] ⟩ do
    if rstamp ≥ t or wstamp ≥ t then
        trigger ⟨ sp2pSend, $p_j$,[READNACK] ⟩;
    else
        rstamp := t; store (rstamp); trigger ⟨ sp2pSend, $p_j$, [READACK, wstamp, v] ⟩;

upon event ⟨ sp2pDeliver, $p_j$, [READNACK] ⟩ do
    trigger ⟨ lacDecide, ⊥ ⟩;

upon event ⟨ sp2pDeliver, $p_j$, [READACK, t, v] ⟩ do
    readSet := readSet ∪ $\{(t, v)\}$

upon (| readSet | > N/2) do
    value := highest(readSet); if v ≠⊥ then estimate := value;
    wAcks := 0; trigger ⟨ sbebBroadcast, [WRITE, ts, estimate] ⟩;
```

plus a termination properties which stipulates that, with probability 1, every correct process eventually decides.

5.4.1 Specification

Each process has an initial value that it proposes to the others through the primitive *rcPropose* (we simply write propose when there is no confusion). All correct processes have to decide on a single value that has to be one of the proposed values: the decision primitive is denoted by *rcdecide*) (we simply write decide when there is no confusion). Randomized consensus ensures the properties RC1–4 listed in Module 5.6.

Algorithm 5.8 Logged abortable consensus algorithm (2/2)

```
upon event ⟨ sbebDeliver, $p_j$, [WRITE, t, v] ⟩ do
    if rstamp > $t$ or wstamp > $t$ then
        trigger ⟨ sp2pSend, $p_j$,[WRITENACK] ⟩;
    else
        value := v; wstamp := t; store (value, wstamp); trigger ⟨ sp2pSend, $p_j$, [WRITEACK] ⟩;

upon event ⟨ sp2pDeliver, $p_j$, [WRITENACK] ⟩ do
    trigger ⟨ lacDecide, ⊥ ⟩;

upon event ⟨ sp2pDeliver, $p_j$, [WRITEACK] ⟩ do
    wAcks := wAcks +1;

upon (|wAcks | > $N/2$) do
    trigger ⟨ lacDecide, estimate ⟩;
```

Module:

Name: Randomized Consensus (rc).

Events:

Request: ⟨ *rcPropose*, v ⟩: Used to propose a value for consensus.

Indication: ⟨ *rcDecide*, v ⟩: Used to indicate the decided value for consensus.

Properties:

RC1: *Termination:* With probability 1, every correct process decides some value.

RC2: *Validity:* If a process decides v, then v was proposed by some process.

RC3: *Integrity:* No process decides twice.

RC4: *Agreement:* No two correct processes decide differently.

Module 5.6 Interface and properties of randomized consensus.

5.4.2 A randomized Consensus Algorithm

The randomized consensus algorithm described here operates in (asynchronous) rounds where, in each round, the processes try to ensure that the same value is proposed by a majority of processes. If there is no such value, the processes use randomization to select which of the initial values they will propose in the next round. The probability that processes agree in a given round is strictly greater than zero. Therefore, if the algorithm continues to execute rounds, eventually it terminates with probability 1.

Algorithm 5.9 is randomized and requires a majority of correct processes to make progress. Initially, each process uses reliable broadcast to disseminate its own initial value to every other correct processes. Therefore, eventually, all correct processes will have all initial values from every other correct process.

Algorithm 5.9 Randomized algorithm: consensus

```
Implements:
    Randomized Consensus (rc);

Uses:
    ReliableBroadcast (rb). BestEffortBroadcast (beb).

upon event ⟨ Init ⟩ do
    decided := ⊥; estimate := ⊥; round := 0;
    val := ∅;

upon event ⟨ rcPropose, $v$ ⟩ do
    trigger ⟨ bebBroadcast, [IniValue, $v$] ⟩;
    estimate := $v$; round := round +1;
    val:= val ∪ {$v$};
    trigger ⟨ bebBroadcast, [Phase1, round, $v$] ⟩;

upon event ⟨ bebDeliver, $p_i$, [IniVal, $v$] ⟩ do
    val:= val ∪ {$v$};

upon event ⟨ bebDeliver, $p_i$, [Phase1, $r$, $v$] ⟩ do
    phase1[$r$] := phase1[$r$] ⊕ $v$;

upon (decided=⊥ ∧ |phase1[round]| > $N/2$) do
    if exists $v$ such that $\forall x \in$ phase1[round]: $x = v$ then estimate := $v$;
    else estimate := ⊥;
    trigger ⟨ bebBroadcast, [Phase2, round, estimate] ⟩;

upon event ⟨ bebDeliver, $p_i$, [Phase2, $r$, $v$] ⟩ do
    phase2[$r$] := phase2[$r$] ⊕ $v$;

upon (decided=⊥ ∧ |phase2[round]| > $N/2$) do
    if exists $v \neq \perp$ such that $\forall x \in$ phase2[round]: $x = v$ then
        decided := $v$;
        trigger ⟨ rbBroadcast, [Decided, round, decided] ⟩;
    else
        if exists $v \in$ phase2[round] such that $v \neq \perp$ then estimate := $v$;
        else estimate := random(val);
        round := round +1; // start one more round
        trigger ⟨ rbBroadcast, [Phase1, round, estimate] ⟩;

upon event ⟨ rbDeliver, $p_i$, [Phase2, $r$, $v$] ⟩ do
    decided := $v$;
    trigger ⟨ rcDecided, decided ⟩;
```

The algorithm operates in rounds. Each round consists of two phases. In the first phase every correct process proposes a value. If a process observes that a majority of processes have proposed the same value in the first phase, then it proposes that value for the second phase. If a process is unable to observe a majority of proposals for the same value in the first phase, it simply proposes $\perp$ for the second phase. Note that as a result of this procedure, if two processes propose a value (different from $\perp$) for the second phase, they propose exactly the same value. Let this value be called *majph1*.

The purpose of the second phase is to verify if *majph1* was observed by a majority of processes. In this is the case, *majph1* is the decided value. A process that receives *majph1* in the second phase but is unable to collect a majority of *majph1* in that phase, starts a new round with *majph1* as its estimate.

Finally, it is possible that a process does not receive *majph1* in the second phase (either because no such value was found in phase 1 or simply because it has received a majority of $\perp$ in the second phase). In this case, the process has to start a new round, with a new estimate. To ensure that there is some probability of obtaining a majority in the new round, the process selects, at random, one of the initial values it has seen from, and uses this value as its proposal for the first phase of the next round.

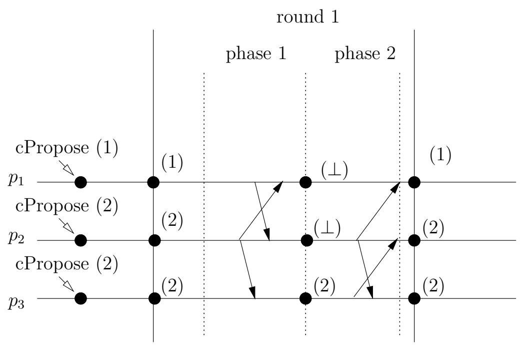

Figure 5.3. Role of randomization.

Figure 5.3 illustrates the idea underlying the algorithm. At first glance, it may seem that a deterministic decision would allow a majority in the first phase to be reached faster. For instance, if a process would receive a majority of $\perp$ in the second phase of a round, it could deterministically select the first non-$\perp$ initial value instead of selecting a value at random. Unfortunately, a deterministic choice allows executions where the algorithm never terminates.

In the example, we have three processes, p_1, p_2 and p_3 with initial values of 1, 2 and 2 respectively. Each process proposes its own value for the first phase of the round. Consider the following execution for the first phase:

- Process p_1 receives the value from p_2. Since both values differ, p_1 proposes $\perp$ for the second phase.
- Process p_2 receives the value from p_1. Since both values differ, p_2 proposes $\perp$ for the second phase.

- Process p_3 receives the value from p_2. Since both values are the same, p_3 proposes 2 for the second phase.

Now consider the following execution for the second phase:

- Process p_1 receives the value from p_2. Since both values are $\perp$, p_1 deterministically selects value 1 for the first phase of the next round.
- Process p_2 receives the value from p_3. Since one of the values is 2, p_2 proposes 2 for the first phase of the next round.
- Process p_3 receives the value from p_2. Since one of the values is 2, p_3 proposes 2 for the first phase of the next round.

This execution is clearly possible. Unfortunately, the result of this execution is that the input values for the next round are exactly the same as for the previous round. The same execution sequence could be repeated indefinitely. Randomization prevents this infinite executions from occurring since, there would be a round where p_1 would also propose 2 as the input value for the next round.

Hands-On

To-be-done. To-be-done.

To-be-done. To-be-done.

To-be-done. To-be-done.

To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done.

Exercices

Exercise 5.1 *Improve our hierarchical regular consensus algorithm to save one communication step. (The algorithm we presented requires N communication steps for all correct processes to decide. By a slight modification, it can run in $N-1$ steps: suggest such a modification.)*

Exercise 5.2 *Explain why none of our (regular) consensus algorithms ensures uniform consensus.*

Exercise 5.3 *Can we optimize our flooding uniform consensus algorithm to save one communication step, i.e., such that all correct processes always decide after $N-1$ communication steps? Consider the case of a system of two processes.*

Exercise 5.4 *What would happen in our flooding uniform consensus algorithm if:*

1. *we did not use set[round] but directly updated proposedSet in* ***upon event*** *bebDeliver?*
2. *we accepted any bebDeliver event, even if $p_i \notin correct$?*

Exercise 5.5 *Consider all our fail-stop consensus algorithms. Explain why none of those algorithms would be correct if the failure detector turns out not to be perfect.*

Exercise 5.6 *Explain why any fail-noisy consensus actually solves uniform consensus.*

Exercise 5.7 *Explain why any fail-noisy consensus (resp. abortable consensus) algorithm needs a majority of correct processes.*

Exercise 5.8 *Give a fail-recovery logged consensus algorithm which uses the Ω abstraction and ensures the following property: if p_1 does not crash and is the only leader from the beginning of the execution, only 3 communication steps, $3(N-1)$ messages, and 1 log at each process of a majority is needed for all correct processes to decide.*

Exercise 5.9 *Give a fail-noisy consensus algorithm that assumes a correct majority of processes and uses an eventually perfect failure detector abstraction in such a way that: (1) in any execution where p_1 is never suspected, p_1 imposes its decision, (2) in any execution where p_1 crashes initially and p_2 is never suspected, p_2 imposes its decision, and so forth.*

Solutions

Solution 5.1 The last process (p_N) does not need to broadcast its message. Indeed, the only process that uses p_N's broadcast value is p_N itself, and p_N decides its proposal just *before* it broadcasts it (not when it delivers it). More generally, note that no process p_i ever uses the value broadcast from any process p_j such that $i \geq j$. □

Solution 5.2 Consider our flooding algorithm first and the scenario of Figure 5.1. Assume that p_1's message has reached only p_2. At the end of the first round, p_2 has not suspected any process and can thus decide 3. However, if p_2 crashes after deciding 3, p_3 and p_4 might decide 5.

Now consider our hierarchical algorithm and the scenario of Figure 5.2. In the case where p_1 decides and crashes and no other process sees p_1's proposal (i.e., 3), then p_1 would decide differently from the other processes. □

Solution 5.3 No. We give a counter example for the particular case of $N = 2$. The interested reader will then easily extend beyond this case to the general case of any N. Consider a system made of two processes p_1 and p_2. We exhibit an execution where processes do not reach uniform agreement after one round, thus they need at least two rounds. More precisely, consider the execution where p_1 and p_2 propose two different values, respectively, v_1 and v_2. Without loss of generality, consider that $v_1 < v_2$. We shall consider the following execution where p_1 is a faulty process.

During round 1, p_1 and p_2 respectively send their message to each other. Process p_1 receives its own value and p_2's message (p_2 is correct), and decides. Assume that p_1 decides its own value v_1, which is different from p_2's value, and then crashes. Now, assume that the message p_1 sent to p_2 in round 1 is arbitrarily delayed (this is possible in an asynchronous system). There is a time after which p_2 permanently suspects p_1 because of the Strong Completeness property of the perfect failure detector. As p_2 does not know that p_1 did send a message, p_2 decides at the end of round 1 on its own value v_2. Hence the violation of uniform agreement.

Note that if we allow processes to decide only after 2 rounds, the above scenario does not happen, because p_1 crashes *before* deciding (i.e. it never decides), and later on, p_2 decides v_2. □

Solution 5.4 For case (1), it would not change anything. Intuitively, the algorithm is correct and preserves uniform agreement because any process executes for N rounds before deciding. Thus, unless all processes crash, there exists a round r during which no process crashes. This is because, at each round, every process broadcasts the values it knows from the previous rounds. After executing round r, all processes that are not crashed know exactly the same information. If we now update *proposedSet* *before* the beginning of the next round (and in particular before the beginning of round r), the processes will still have the information on time. In conclusion, the fact they get the information earlier is not a problem since they must execute N rounds anyway.

In case (2), the algorithm would be wrong. In the following, we exhibit an execution that leads to disagreement. More precisely, consider a system made of three processes p_1, p_2 and p_3. The processes propose 0, 1 and 1, respectively. During the first round, the messages of p_1 are delayed and p_2 and p_3 never receive them. Process p_1 crashes at the end of round 2, but p_2 still receives p_1's round 2 message (that is, the set $\{0, 1\}$) in round 2 (possible because channels are not FIFO). Process p_3 does not receive p_1's message in round 2 though. In round 3, the message from p_2 to p_3 (that is, the set $\{0, 1\}$) is delayed and process p_2 crashes at the end of round 3, so that p_3 never receives p_2's message. Before crashing, p_2 decides on value 0, whereas p_3 decides on 1. Hence the disagreement. □

Solution 5.5 In all our fail-stop algorithms, there is at least one critical point where a process p waits to deliver a message from a process q or to suspect the process q. Should q crash and p never suspect q, p would remain blocked forever and never decide. In short, in any of our algorithm using a perfect failure detector, a violation of *strong completeness* could lead to violate the *termination* property of consensus.

Consider now *strong accuracy*. Consider for instance our flooding consensus algorithm and the scenario of Figure 5.1: if p_2 crashes after deciding 3, and p_1 is falsely suspected to have crashed by p_3 and p_4, then p_3 and p_4 would decide 5. A similar scenario can happen for our hierarchical consensus algorithms. □

Solution 5.6 Consider any fail-noisy consensus algorithm that does not solve uniform consensus. This means that there is an execution where two processes p_i and p_j decide differently and one of them crashes: the algorithm violates uniform agreement. Assume that process p_i crashes. With an eventually perfect failure detector, it might be the case that p_i is not crashed but just falsely suspected by all other processes. Process p_j would decide the same as in the previous execution and the algorithm would violate (non-uniform) agreement. □

Solution 5.7 We explain this for the case of a system of four processes $\{p_1, p_2, p_3, p_4\}$. Assume by contradiction that there is a fail-noisy consensus algorithm that tolerates the crash of two processes. Assume that p_1 and p_2 propose a value v whereas p_3 and p_4 propose a different value v'. Consider an execution E_1 where p_1 and p_2 crash initially: in this execution, p_3 and p_4 decide v' to respect the *validity* property of consensus. Consider also an execution E_2 where p_3 and p_4 crash initially: in this scenario, p_1 and p_2 decide v. With an eventually perfect failure detector, a third execution E_3

is possible: the one where no process crashes, p_1 and p_2 falsely suspect p_3 and p_4 whereas p_3 and p_4 falsely suspect p_1 and p_2. In this execution E_3, p_1 and p_2 decide v, just as in execution E_1 (they execute the same steps as in E_1 and cannot distinguish E_3 from E_1 up to the decision point), whereas p_3 and p_4 decide v', just as in execution E_2 (they execute the same steps as in E_2 and cannot distinguish E_3 from E_2 up to the decision point). *Agreement* would hence be violated.

A similar argument applies to abortable consensus. We would in this case assume that p_1 is leader in execution $E1$ whereas p_4 is leader in execution E_2 and consider an execution E_3 where both are leaders until the decision point. □

Solution 5.8 The algorithm is a variant of our logged consensus algorithm where the underlying logged abortable consensus building block is opened for optimization purposes. In the case where p_1 is initially elected leader, p_1 directly tries to impose its decision, i.e., without consulting the other processes. In a sense, it skips the read phase of the underlying logged abortable consensus. This computation phase is actually only needed to make sure that the leader will propose any value that might have been proposed. For the case where p_1 is initially the leader, p_1 is sure that no decision has been made in a previous round (there cannot be any previous round) and can save one communication phase by directly proposing its own proposal. This also leads to save the first access to stable storage and one communication round-trip.
□

Solution 5.9 The algorithm we give here is round-based and the processes play two roles: the role of a leader, described in Algorithm 5.10, and the role of a witness, described in Algorithm 5.11. Every process goes sequentially from round i to round $i + 1$: no process ever jumps from one round k to another round $k' < k + 1$. Every round has a leader determined a priori: the leader of round i is process $p_{(i-1)\ mod\ (N+1)}$, e.g., p_2 is the leader of rounds 2, $N + 2$, $2N + 2$, etc.

The process that is leader in a round computes a new proposal and tries to impose that proposal to all: every process that gets the proposal from the current leader adopts this proposal and assigns it the current round number as a timestamp. Then it acknowledges that proposal back to the leader. If the leader gets a majority of acknowledgements, it decides and disseminates that decision using a reliable broadcast abstraction.

There is a critical point where processes need the input of their failure detector in every round. When the processes are waiting for a proposal from the leader of that round, the processes should not wait indefinitely if the leader has crashed without having broadcast its proposal. In this case, the processes consult their failure detector module to get a hint whether the leader process has crashed. Given that an eventually perfect detector ensures

Algorithm 5.10 Fail-noisy consensus algorithm: leader role.

```
Uses:
    PerfectPointToPointLinks (pp2p);
    ReliableBroadcast (rb);
    BestEffortBroadcast (beb);
    EventuallyPerfectFailureDetector ($\diamond\mathcal{P}$);

upon event ⟨ Init ⟩ do
    proposal := decided := ⊥;
    round := 1;
    suspected:= estimate-set[] := ack-set[] := ∅;
    estimate[] := ack[] := false;
    for i = 1 to N do ps[i] := $p_i$;

upon event(ps[round mod N + 1]= self.id) ∧ ⟨ pp2pDeliver, $p_i$, [ESTIMATE, round, estimate] ⟩ do
    estimate-set[round] := estimate-set[round] ∪{estimate};

upon (ps[round mod N + 1]= self.id) ∧ (|estimate-set[round]| > N/2)) do
    proposal := highest(estimate-set[round]);
    trigger ⟨ bebBroadcast, [PROPOSE, round, proposal] ⟩;

upon event(ps[round mod N + 1]= self.id) ∧ ⟨ pp2pDeliver, $p_i$, [ACK, round] ⟩ do
    ack-set[round] := ack-set[round] ∪{$p_i$};

upon (ps[round mod N + 1]= self.id) ∧ ⟨ pp2pDeliver, $p_i$, [NACK, round] ⟩ do
    round := round + 1;

upon (ps[round mod N + 1]= self.id) ∧ (|ack-set[round] | > N/2) do
    trigger ⟨ rbBroadcast, [DECIDE, proposal] ⟩;
```

that, eventually, every crashed process is suspected by every correct process, the process that is waiting for a crashed leader will eventually suspect it. In this case, the process sends a specific message *nack* to the leader, then moves to the next round. In fact, a leader that is waiting for acknowledgements might get some *nacks* (if some processes falsely suspected it): in this case, the leader moves to the next round without deciding.

Note also that processes after acknowledging a proposal move to the next round directly: they do not need to wait for a decision. They might deliver the decision through the reliable broadcast dissemination phase. In that case, they will simply stop their algorithm.

Correctness. Validity and *integrity* follow from the algorithm and the properties of the underlying communication abstractions. Consider *termination.* If some correct process decides, it decides through the reliable broadcast abstraction, i.e., by rbDelivering a decision message. By the properties of this broadcast abstraction, every correct process rbDelivers the decision message and decides. Assume by contradiction that there is at least one correct pro-

Algorithm 5.11 Fail-noisy consensus algorithm: witness role.

```
upon event ⟨ suspect, $p_i$ ⟩ do
    suspected := suspected $\cup\{p_i\}$;

upon event ⟨ restore, $p_i$ ⟩ do
    suspected := suspected $\setminus\{p_i\}$;

upon event ⟨ ucPropose, $v$ ⟩ do
    proposal := $[v, 0]$;

upon event (proposal $\neq \bot$) $\wedge$ (estimate[round] = false) do
    estimate[round] := true;
    trigger ⟨ pp2pSend, ps[round mod N], [ESTIMATE, round, proposal] ⟩;

upon event ⟨ bebDeliver, $p_i$, [PROPOSE, round, value] ⟩ $\wedge$ (ack[round] = false) do
    ack[round] := true;
    proposal := $[value, round]$;
    trigger ⟨ pp2pSend, ps[round mod N], [ACK, round] ⟩;
    round := round + 1;

upon event (ps[round mod N] $\in$ suspected) $\wedge$ (ack[round] = false) do
    ack[round] := true;
    trigger ⟨ pp2pSend, ps[round mod N], [NACK, round] ⟩;
    round := round + 1;

upon event ⟨ rbDeliver, $p_i$, [DECIDED, v] ⟩ $\wedge$ (decided = $\bot$) do
    decided := v;
    trigger ⟨ ucDecided, v ⟩;
```

cess and no correct process decides. Consider the time t after which all faulty processes crashed, all faulty processes are suspected by every correct process forever and no correct process is ever suspected. Let p_i be the first correct process that is leader after time t and let r denote the round at which that process is leader. If no process has decided, then all correct processes reach round r and p_i eventually reaches a decision and rbBroadcasts that decision. Consider now *agreement*. Consider by contradition any two rounds i and j, j is the closest integer to i such that $j > i$ and $p_{i \ mod \ N+1}$, and $p_{j \ mod \ N+1}$, proposed two different decision values v and v'. Process $p_{j \ mod \ N+1}$ must have adopted v before reaching round j. This is because $p_{j \ mod \ N+1}$ selects the value with the highest timestamp and $p_{j \ mod \ N+1}$ cannot miss the value of $p_{i \ mod \ N+1}$: any two majorities always intersect. Given that j is the closest integer to i such that some process proposed v' different from v, after v was proposed, we have a contradiction.

Performance. If no process fails or is suspected to have failed, then 4 communication steps and $4(N-1)$ messages are needed for all correct processes to decide.

□

Historical Notes

- The consensus problem was defined in a seminal paper (Lamport, Shostak, and Pease 1982).
- In another seminal paper (Fischer, Lynch, and Paterson 1985), it was proved that, consensus is impossible to solve with a deterministic algorithm in a fail-silent model even if only one process fails.
- Later on, intermediate models between the synchronous and the asynchronous model were introduced to circumvent the consensus impossibility (Dwork, Lynch, and Stockmeyer 1988). The notion of failure detection was defined to encapsulate partial synchrony assumptions in (Chandra and Toueg 1996; Chandra, Hadzilacos, and Toueg 1996).
- The round-based fail-noisy consensus algorithm presented in the exercices was introduced in (Chandra and Toueg 1996) whereas our fail-noisy consensus algorithm based on abortable consensus is a modular variant of (Lamport 1989).
- It was shown in in (Guerraoui 2000) that any fail-noisy algorithm that solves consensus also solves uniform consensus.
- It was shown in (Chandra and Toueg 1996; Guerraoui 2000) that any consensus algorithm using an unreliable failure detector requires a majority of correct processes.
- Our randomized consensus algorithm is from (Ezhilchelvan, Mostefaoui, and Raynal 2001), and is a generalization of the binary randomized consensus algorithm of (Ben-Or 1983).

6. Ordering

So when they continued asking him, he lifted up himself, and said unto them, he that is without sin among you, let him first cast a stone at her.
(John (not Lennon) 8:7)

This chapter considers ordering abstractions. These are broadcast communication abstractions that provide ordering guarantees among the messages exchanged between the processes. We will study here two categories of such abstractions: *causal ordering* as well as *total ordering* abstractions.

We will build causal order broadcast algorithms over reliable broadcast algorithms and total order broadcast algorithms over reliable broadcast and consensus ones. This modular approach will help abstract away the underlying variations of the models, i.e., fail-silent, fail-stop, etc.

6.1 Causal Order Broadcast

So far, we did not consider any ordering guarantee among messages delivered by different processes. In particular, when we consider a reliable broadcast abstraction for instance, messages can be delivered in any order and the reliability guarantees are in a sense orthogonal to such an order.

In this section, we discuss the issue of ensuring message delivery according to *causal ordering.* This is a generalization of FIFO (*first-in-first-out*) ordering where messages from the same process should be delivered in the order according to which they were broadcast.

6.1.1 Overview

Consider the case of a distributed message board that manages two types of information: proposals and comments on previous proposals. To make the interface user-friendly, comments are depicted attached to the proposal they are referring to. Assume that we implement the board application by replicating all the information at all participants. This can be achieved through the use of a reliable broadcast primitive to disseminate both proposals and comments. With a reliable broadcast, the following sequence would be possible: participant p_1 broadcasts a message m_1 containing a new proposal; participant p_2 delivers m_1 and disseminates a comment in message m_2; due to message delays, another participant p_3 delivers m_2 before m_1. In this case, the application at p_3 would be forced to keep m_2 and wait for m_1, to avoid presenting the comment before the proposal being commented. In fact, m_2 is causally after m_1 ($m_1 \rightarrow m_2$), and a causal order primitive would make sure that m_1 would have been delivered before m_2, relieving the application programmer of such a task.

6.1.2 Specifications

As the name indicates, a causal order protocol ensures that messages are delivered respecting cause-effect relations. This is expressed by the *happened-before* relation described earlier in this manuscript. This relation, also called the *causal order* relation, when applied to the messages exchanged among processes, is captured by broadcast and delivery events. In this case, we say that a message m_1 may potentially have caused another message m_2 (or m_1 happened before m_2), denoted as $m_1 \rightarrow m_2$, if the following relation, applies:

- m_1 and m_2 were broadcast by the same process p and m_1 was broadcast before m_2 (Figure 6.1a).
- m_1 was delivered by process p, m_2 was broadcast by process p and m_2 was broadcast after the delivery of m_1 (Figure 6.1b).
- there exists some message m' such that $m_1 \rightarrow m'$ and $m' \rightarrow m_2$ (Figure 6.1c).

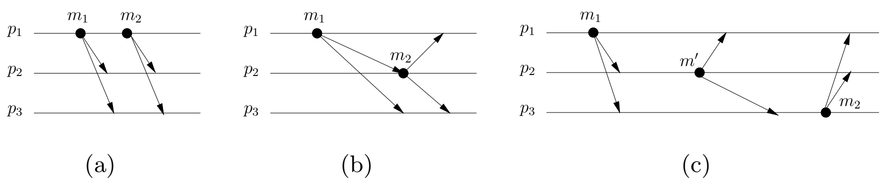

Figure 6.1. Causal order of messages.

Using the causal order relation, a causal order broadcast can be defined by the property CB in Module 6.1. The property states that messages are delivered by the communication abstraction according to the causal order

Module:

Name: CausalOrder (co).

Events:

Request: $\langle$ *coBroadcast*, m $\rangle$: Used to broadcast message m to Π.

Indication: $\langle$ *coDeliver*, src, m $\rangle$: Used to deliver message m broadcast by process src.

Properties:

CB: *Causal delivery:* No process p_i delivers a message m_2 unless p_i has already delivered every message m_1 such that $m_1 \rightarrow m_2$.

Module 6.1 Properties of causal broadcast.

Module:

Name: ReliableCausalOrder (rco).

Events:

$\langle$ *rcoBroadcast*, m $\rangle$ and $\langle$ *rcoDeliver*, src, m $\rangle$: with the same meaning and interface as the causal order interface.

Properties:

RB1-RB3, **RB4**, from reliable broadcast and **CB** from causal order broadcast.

Module 6.2 Properties of reliable causal broadcast.

Module:

Name: UniformReliableCausalOrder (urco).

Events:

$\langle$ *urcoBroadcast*, m $\rangle$ and $\langle$ *urcoDeliver*, src, m $\rangle$: with the same meaning and interface as the causal order interface.

Properties:

URB1-URB4 and **CB**, from uniform reliable broadcast and causal order.

Module 6.3 Properties of uniform reliable causal broadcast.

relation. There must be no “holes” in the causal past, i.e., when a message is delivered, all preceding messages have already been delivered.

Clearly, a broadcast primitive that has only to ensure the *causal delivery* property might not be very useful: the property might be ensured by having no process ever deliver any message. However, the *causal delivery* property can be combined with both reliable broadcast and uniform reliable broadcast semantics. These combinations would have the interface and properties of Module 6.2 and Module 6.3, respectively. To simplify, we call the first *causal order broadcast* and the second *uniform causal order broadcast.*

Algorithm 6.1 No-waiting reliable causal broadcast.

```
Implements:
    ReliableCausalOrder (rco).

Uses:
    ReliableBroadcast (rb).

upon event ⟨ Init ⟩ do
    delivered := ∅;
    past := ∅

upon event ⟨ rcoBroadcast, m ⟩ do
    trigger ⟨ rbBroadcast, [DATA, past, m] ⟩;
    past := past ∪ { [self,m] };

upon event ⟨ rbDeliver, $p_i$, [DATA, $past_m$, m] ⟩ do
    if $m \notin$ delivered then
        forall $[s_n, n] \in past_m$ do //in order
            if $n \notin$ delivered then
                trigger ⟨ rcoDeliver, $s_n, n$ ⟩;
                delivered := delivered ∪ $\{n\}$
                past := past ∪ $\{[s_n, n]\}$;
        trigger ⟨ rcoDeliver, $p_i, m$ ⟩;
        delivered := delivered ∪ $\{m\}$
        past := past ∪ $\{[p_i, m]\}$;
```

6.1.3 Fail-Silent Algorithm: No-Waiting Causal Broadcast

Algorithm 6.1 is a causal broadcast algorithm. The algorithm uses an underlying reliable broadcast communication abstraction defined through rbBroadcast and rbDeliver primitives. The same algorithm could be used to implement a uniform causal broadcast abstraction, simply by replacing the underlying reliable broadcast module by a uniform reliable broadcast module.

The algorithm is said to be *no-waiting* in the following sense: whenever a process rbDeliver a message m, it can rcoDeliver m without waiting for other messages to be rbDelivered. Each message m carries a control field called $past_m$. The $past_m$ field of a message m includes all messages that causally precede m. When a message m is rbDelivered, $past_m$ is first inspected: messages in $past_m$ that have not been rcoDelivered must be rcoDelivered before m itself is also rcoDelivered. In order to record its own causal past, each process p memorizes all the messages it has rcoBroadcast or rcoDelivered in a local variable *past*. Note that *past* (and $past_m$) are ordered sets.

The biggest advantage of Algorithm 6.1 is that the delivery of a message is never delayed in order to enforce causal order. This is illustrated in Figure 6.2. Consider for instance process p_4 and message m_2. Process p_4 rbDelivers m_2. Since m_2 carries m_1 in its past, m_1 and m_2 are delivered in order. Finally, when m_1 is rbDelivered from p_1, it is discarded.

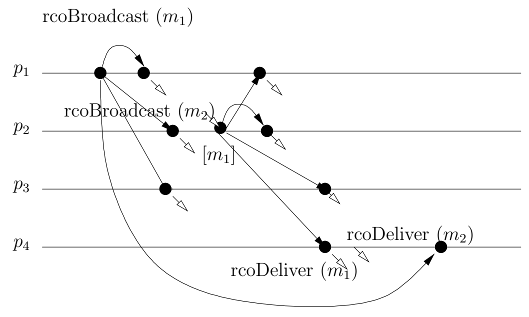

Figure 6.2. Sample execution of causal broadcast with complete past.

Correctness. All properties of reliable broadcast follow from the use of an underlying reliable broadcast primitive and the no-waiting flavor of the algorithm. The causal order property is enforced by having every message carry its causal past and every process making sure that it rcoDelivers the causal past of a message before rcoDelivering the message.

Performance. The algorithm does not add additional communication steps or messages to the underlying uniform reliable broadcast algorithm. However, the size of the messages grows linearly with time, unless some companion garbage collection algorithm to purge *past* is executed.

There is a clear inconvenience however with this algorithm: the $past_m$ field may become extremely large, since it includes the complete causal past of m. In the next subsection we illustrate a simple scheme to reduce the size of *past*. However, even with this optimization, this approach consumes too much bandwidth to be used in practice. Note also that no effort is made to prevent the size of the *delivered* set from growing indefinitely. We will later discuss an algorithm that circumvents these issues at the expense of blocking.

6.1.4 Fail-Stop Extension: Garbage Collecting the Causal Past

We now present a very simple algorithm, Algorithm 6.2, to delete messages from the past set. The algorithm supposes a fail-stop model, i.e., it builds upon a perfect failure detector.

The garbage collection algorithm, is aimed to be used in conjunction with Algorithm 6.1. It works as follows: when a process rbDelivers a message m, it rbBroadcasts an *Ack* message to all other processes; when an *Ack* for message m has been rbDelivered from all correct processes, m is purged from *past*.

6.1.5 Fail Silent Algorithm: Waiting Causal Order Broadcast

Algorithm 6.3 circumvents the main limitation of Algorithm 6.1: the huge size of the messages. Instead of keeping a record of all past messages, we

Algorithm 6.2 Garbage collection of past.

```
Implements:
    GarbageCollectionOfPast.

Extends:
    No-waiting reliable causal broadcast algorithms.
    (Algorithm 6.1).

Uses:
    ReliableBroadcast (rb).
    PerfectFailureDetector ($\mathcal{P}$);

upon event ⟨ Init ⟩ do
    delivered := past := $\emptyset$;
    correct := $\Pi$;
    ack$_m$ := $\emptyset, \forall_m$;

upon event ⟨ crash, $p_i$ ⟩ do
    correct := correct $\setminus\{p_i\}$;

upon $\exists_m \in$ delivered: self $\notin$ ack$_m$ do
    ack$_m$ := ack$_m$ $\cup$ { self };
    trigger ⟨ rbBroadcast, [Ack, $m$] ⟩;

upon event ⟨ rbDeliver, $p_i$, [Ack, $m$] ⟩ do
    ack$_m$ := ack$_m$ $\cup$ $\{p_i\}$;
    if correct $\subseteq$ ack$_m$do
        past := past $\setminus\{[s_m, m]\}$;
```

keep just the sequence number of the last message that was rcoBroadcast. In this way, $past_p$ is reduced to an array of integers. Temporal information stored in this way is called a *vector clock*. Algorithm 6.3 uses an underlying reliable broadcast communication abstraction defined through rbBroadcast and rbDeliver primitives.

With Algorithm 6.3, messages do not carry the complete past, only a summary of the past in the form of the vector clock. It is possible that a message may be prevented from being rcoDelivered immediately when it is rbDelivered, because some of the preceding messages have not been rbDelivered yet. It is also possible that the rbDelivery of a single message triggers the rcoDelivery of several messages that were waiting to be rcoDelivered. For instance, in Figure 6.3 message m_2 is rbDelivered at p_4 before message m_1, but its rcoDelivery is delayed until m_1 is rbDelivered and rcoDelivered.

As with the no-waiting variant, Algorithm 6.3 could also be used to implement uniform reliable causal broadcast, simply by replacing the underlying reliable broadcast module by a uniform reliable broadcast module.

Performance. The algorithm does not add any additional communication steps or messages to the underlying reliable broadcast algorithm. The size of

Algorithm 6.3 Waiting causal broadcast.

```
Implements:
    ReliableCausalOrder (rco).

Uses:
    ReliableBroadcast (rb).

upon event ⟨ init ⟩ do
    $\forall_{p_i \in \Pi} : VC[p_i] := 0;$

upon event ⟨ rcoBroadcast, m ⟩ do
    trigger ⟨ rbBroadcast, [DATA, VC, m] ⟩;
    $VC[self] := VC[self]+1;$

upon event ⟨ rbDeliver, $p_i$, [DATA, $VC_m$, m] ⟩ do
    wait until $(\forall_{p_j} : VC[p_j] \geq VC_m[p_j])$
    trigger ⟨ rcoDeliver, $s_m, m$ ⟩;
    if $p_i \neq$ self then
        VC[$p_i$] := VC[$p_i$]+1;
```

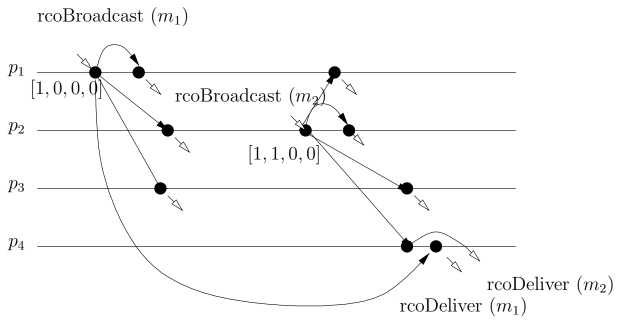

Figure 6.3. Sample execution of causal broadcast with vector clocks.

the message header is linear with regard to the number of processes in the system.

6.2 Total Order Broadcast

6.2.1 Overview

Causal order broadcast enforces a global ordering for all messages that causally depend on each other: such messages need to be delivered in the same order and this order must be the causal order. Messages that are not related by causal order are said to be *concurrent* messages. Such messages can be delivered in any order.

In particular, if in parallel two processes each broadcasts a message, say p_1 broadcasts m_1 and p_2 broadcasts m_2, then the messages might be delivered in different orders by the processes. For instance, p_1 might deliver m_1 and then m_2, whereas p_2 might deliver m_2 and then m_1.

A total order broadcast abstraction is a reliable broadcast communication abstraction which ensures that all processes deliver the same set of messages exactly in the same order. In short, whereas reliable broadcast ensures that processes agree on the same set of messages they deliver, total order broadcast ensures that they agree on the same sequence of messages, i.e., the set is now ordered. This abstraction is sometimes also called *atomic broadcast* because the message delivery occurs as if the broadcast was an indivisible primitive (i.e., atomic): the message is delivered to all or to none of the processes and, if it is delivered, every other messages is ordered either before or after this message.

This sort of ordering eases the maintenance of a global consistent state among a set of processes. In particular, if each process is programmed as a state machine, i.e., its state at a given point depends exclusively on the initial state and on the sequence of messages received, the use of total order broadcast ensures consistent replicated behavior. The replicated state machine is one of the fundamental techniques to achieve fault-tolerance. One application of this approach is to implement shared objects of arbitrary types in a distributed system, i.e., beyond read-write (register) objects studied earlier in the manuscript. Each process would host a copy of the object and invocations to the distributed object would be broadcast to all copies using the total order broadcast primitive. This will ensure that all copies keep the same state and ensure that the responses are consistent. In short, the use of total order broadcast ensures that the object is highly available, yet it appears as if it was a single logical entity accessed in a sequential and failure-free manner.

6.2.2 Specification

Two variants of the abstraction can be defined: a regular variant only ensures total order among the correct processes; and a uniform variant that ensures total order with regard to the faulty processes as well. Total order is captured by the properties TO1 and RB1-4 depicted in Module 6.4 and UTO1 and URB1-4 in Module 6.5.

Note that the total order property (uniform or not) can be combined with the properties of a uniform reliable broadcast. For conciseness, we omit the interface of the resulting modules.

Note also that total order is orthogonal to the causal order discussed in Section 6.1. It is possible to have a total-order abstraction that does not respect causal order. (As we pointed out, a causal order abstraction does not enforce total order and the processes may deliver concurrent messages in different order to different processes.) On the other hand, it is possible to

Module:

Name: TotalOrder (to).

Events:

Request: ⟨ *toBroadcast*, m ⟩: Used to broadcast message m to Π.

Indication: ⟨ *toDeliver*, src, m ⟩: Used to deliver message m sent by process *src*.

Properties:

TO1: *Total order:* Let m_1 and m_2 be any two messages. Let p_i and p_j be any two correct processes that deliver m_2. If p_i delivers m_1 before m_2, then p_j delivers m_1 before m_2.

RB1-RB4: from reliable broadcast.

Module 6.4 Interface and properties of total order broadcast.

Module:

Name: UniformTotalOrder (uto).

Events:

⟨ *utoBroadcast*, m ⟩, ⟨ *utoDeliver*, src, m ⟩: with the same meaning and interface of the consensus interface.

Properties:

UTO1: *Uniform total order:* Let m_1 and m_2 be any two messages. Let p_i and p_j be any two processes that deliver m_2. If p_i delivers m_1 before m_2, then p_j delivers m_1 before m_2.

URB1-URB4: from uniform reliable broadcast.

Module 6.5 Interface and properties of uniform total order broadcast.

build a total order abstraction on top of a causal order primitive. We also omit the interface of the resuling module in this case.

6.2.3 Algorithm: Uniform Total Order Broadcast

In the following, we give a uniform total order broadcast algorithm. More precisely, the algorithm ensures the properties of uniform reliable broadcast plus the uniform total order property. The algorithm uses a uniform reliable broadcast and a uniform consensus abstractions as underlying building blocks. The algorithm might perform in a fail-silent model assuming a correct majority of processes as well as in a fail-stop model with no assumption on the number of correct processes. This depends on how the uniform reliable broadcast and consensus abstractions are themselves implemented.

In the total order broadcast algorithm, messages are first disseminated using a uniform (but unordered) reliable broadcast primitive. Messages delivered this way are stored in a bag of unordered messages at every process.

The processes then use the consensus abstraction to order the messages in this bag.

More precisely, the algorithm works in consecutive rounds. Processes go sequentially from 1 to round 2, .., to round i to round $i+1$ and so forth: as long as new messages are broadcast, the processes keep on moving from one round to the other. There is one consensus instance per round. The consensus instance of a given round is used to make the processes agree on a set of messages to assign to the sequence number corresponding to that round: these messages will be delivered in that round. For instance, the first round decides which messages are assigned sequence number 1, i.e., which messages are delivered in round 1. The second round decides which messages are assigned sequence number 2, etc. All messages that are assigned round number 2 are delivered after the messages assigned round number 1. Messages with the same sequence number are delivered according to some deterministic order agreed upon by the processes in advance (e.g., based on message identifiers). That is, once the processes have agreed on a set of messages for a given round, they simply apply a deterministic function to sort the messages of the same set.

In each instance of the consensus, every process proposes a (potentially different) set of messages to be ordered. The process simply proposes the set of messages it has seen (i.e., urbDelivered) and not yet delivered according to the total order semantics (i.e., utoDelivered). The properties of consensus ensure that all processes decide the same set of messages for that sequence number. The full description is given in Algorithm 6.4. The *wait* flag is used to ensure that a new round is not started before the previous round has terminated.

An execution of the algorithm is illustrated in Figure 6.4. The figure is unfolded into two parallel flows: That of the reliable broadcasts, used to disseminate the messages, and that of the consensus instances, used to order the messages. As messages are received from the reliable module they are proposed to the next instance of consensus. For instance, process p_4 proposes message m_2 to the first instance of consensus. Since the first instance of consensus decides message m_1, process p_4 re-submits m_2 (along with m_3 that was received meanwhile) to the second instance of consensus.

Correctness. The no-creation property follows from (1) the no-creation property of the reliable broadcast abstraction and (2) the validity property of consensus. The no-duplication property follows from (1) the no-duplication property of the reliable broadcast abstraction, and (2) the integrity property of consensus (more precisely, the use of the variable delivery). Consider the agreement property. Assume that some correct process p_i utoDelivers some message m. By the algorithm, p_i must have decided a batch of messages with m inside that batch. Every correct process will reach that point because of the algorithm and the termination property of consensus, will decide that batch, and will utoDeliver m. Consider the validity property of total order

Algorithm 6.4 Uniform total order broadcast algorithm.

```
Implements:
    UniformTotalOrderBroadcast (uto);

Uses:
    UniformReliableBroadcast (urb).
    UniformConsensus (uc);

upon event ⟨ Init ⟩ do
    unordered := delivered := $\emptyset$;
    wait := false;
    sn := 1;

upon event ⟨ utoBroadcast, $m$ ⟩ do
    trigger ⟨ urbBroadcast, $m$ ⟩;

upon event ⟨ urbDeliver, $s_m$, $m$ ⟩ do
    if $m \notin$ delivered then
        unordered := unordered $\cup$ $\{(s_m, m)\}$

upon (unordered $\neq \emptyset$) $\wedge$ ($\neg$ wait) do
    wait := true;
    trigger ⟨ ucPropose, sn,unordered ⟩;

upon event ⟨ ucDecided, sn, decided ⟩ do
    delivered := delivered $\cup$ decided;
    unordered := unordered $\setminus$ decided;
    decided := sort (decided); // some deterministic order;
    $\forall_{(s_m,m)} \in$ decided: trigger ⟨ utoDeliver, $s_m, m$ ⟩; // following the deterministic order
    sn := sn +1;
    wait := false;
```

broadcast, and let p_i be some correct process that utoBroadcasts a message m. Assume by contradiction that p_i never utoDelivers m. This means that m is never included in a batch of messages that some correct process decides. By the validity property of reliable broadcast, every correct process will eventually urbDeliver and propose m in a batch of messages to consensus. By the validity property of consensus, p_i will decide a batch of messages including m and will utoDeliver m. Consider now the total order property. Let p_i and p_j be any two processes that utoDeliver some message m_2. Assume that p_i utoDelivers some message m_1 before m_2. If p_i utoDelivers m_1 and m_2 in the same batch (i.e., the same round number), then by the agreement property of consensus, p_j must have also decided the same batch. Thus, p_j must utoDeliver m_1 before m_2 since we assume a deterministic function to order the messages for the same batch before their utoDelivery. Assume that m_1 is from a previous batch at p_i. By the agreement property of consensus, p_j must have decided the batch of m_1 as well. Given that processes proceed

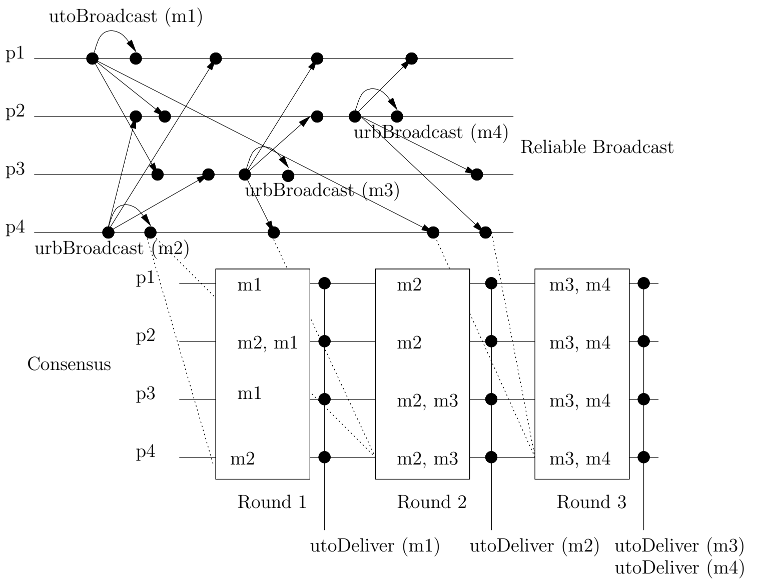

Figure 6.4. Sample execution of the uniform total order broadcast algorithm.

sequentially from one round to the other, then p_j must have utoDelivered m_1 before m_2.

Performance. The algorithm requires at least one communication step to execute the reliable broadcast and at least two communication steps to execute the consensus. Therefore, even if no failures occur, at least three communication steps are required.

Variations. It is easy to see that a regular total order broadcast algorithm is automatically obtained by replacing the uniform consensus abstraction by a regular one. Similarly, one could obtain a total order broadcast that satisfies uniform agreement if we used a uniform reliable broadcast abstraction instead of regular reliable broadcast abstraction. Finally, the algorithm can trivially be made to ensure in addition causal ordering, for instance if we append past information with every message (see our no-waiting causal order broadcast algorithm).

6.3 Logged Total Order Broadcast

To derive a total order abstraction for the fail-recovery model we apply the same sort of approach we have used to derive a reliable broadcast or a consensus abstractions for the same model. We depart from an abstraction designed from the crash-stop model and adapt the following aspects: interface with adjacent modules, logging of relevant state, and definition of recovery

Module:

Name: TotalOrder in the Crash-Recovery model (cr-uto).

Events:

Request: ⟨ *cr-utoBroadcast*, m ⟩: Used to broadcast message m.

Indication: ⟨ *cr-utoDeliver*, *delivered* ⟩: Used to deliver the log of all ordered messages up to the moment the indication is generated.

Properties:

CR-UTO1: *Total order:* Let $delivered_i$ be the sequence of messages delivered to process p_i. For any pair (i, j), either $delivered_i$ is a prefix of $delivered_j$ or $delivered_j$ is a prefix of $delivered_i$.

CR-RB1 to CR-RB3: from reliable broadcast in the crash recovery model.

CR-UTO4: *Uniform Agreement:* If there exists p_i such that $m \in delivered_i$ then eventually $m \in delivered_j$ at every process p_j that eventually remains permanently up.

Module 6.6 Interface and properties of total order broadcast.

procedures. Besides, we make use of underlying abstractions implemented in the crash-recovery model, e.g., consensus and reliable broadcast.

6.3.1 Specification

We consider here just the uniform definition, which is presented in Module 6.6. Note that the module exports to the upper layers the sequence of delivered (and ordered) messages.

6.3.2 Fail-Recovery Algorithm: Total Order Broadcast Algorithm with Logs

Our algorithm (Algorithm 6.5) closely follows the algorithm for the crash no-recovery model presented in Section 6.2. The algorithm works as follows. Messages sent by the upper layer are disseminated using the reliable broadcast algorithm for the fail-recovery model. The total order algorithm keeps two sets of messages: the set of *unordered* messages (these messages are the messages received from the reliable broadcast module) and the set of *ordered* messages (obtained by concatenating the result of the several executions of consensus). A new consensus is started when one notices that there are unordered messages that have not yet been ordered by previous consensus executions. The *wait* flag is also used to ensure that consensus are invoked in serial order. Upon a crash and recovery, the total order module may reinvoke the same consensus execution more than once. Before invoking the ith instance of consensus, the total order algorithm stores the values to be

Algorithm 6.5 Uniform total order broadcast algorithm for the crash-recovery model.

```
Implements:
    CrashRecoveryUniformTotalOrderBroadcast (cr-uto);

Uses:
    CrashRecoveryReliableBroadcast (cr-rb).
    CrashRecoveryUniformConsensus (cr-uc);

upon event ⟨ Init ⟩ do
    unordered := $\emptyset$; delivered := $\emptyset$;
    sn := 0; wait := false;
    $\forall_k$ : propose[$k$] := $\perp$;

upon event ⟨ Recovery ⟩ do
    sn := 0; wait := false;
    while propose[$k$] $\neq \perp$ do
        trigger ⟨ cr-ucPropose, sn, propose[$ns$] ⟩;
        wait ⟨ cr-ucDecided, sn, decided ⟩;
        decided := sort (decided); // some deterministic order;
        delivered := delivered $\oplus$ decided;
        sn := sn +1;
    trigger ⟨ cr-utoDeliver, delivered ⟩;

upon event ⟨ cr-utoBroadcast, $m$ ⟩ do
    trigger ⟨ cr-rbBroadcast, $m$ ⟩;

upon event ⟨ cr-rbDeliver, msgs ⟩ do
    unordered := unordered $\cup$ msgs;

upon (unordered\decided $\neq \emptyset$) $\wedge$ ($\neg$ wait) do
    wait := true;
    propose[$ns$] := unordered\delivered; store (propose[$ns$]);
    trigger ⟨ cr-ucPropose, sn, propose[$ns$] ⟩;

upon event ⟨ cr-ucDecided, sn, decided ⟩ do
    decided := sort (decided); // some deterministic order;
    delivered := delivered $\cup$ decided;
    trigger ⟨ cr-utoDeliver, delivered ⟩;
    sn := sn +1; wait := false;
```

proposed in stable storage. This ensures that a given instance of consensus is always invoked with exactly the same parameters. This may not be strictly needed (depending on the implementation of consensus) but is consistent with the intuitive notion that each processes proposes a value by storing it in stable storage.

The algorithm has the interesting feature of never storing the unordered and delivered sets. These sets are simply reconstructed upon recovery from the stable storage kept internally by the reliable broadcast and consensus implementations. Since the initial values proposed for each consensus execution are logged, the process may re-invoke all past instance of consensus to obtain all messages ordered in previous rounds.

Correctness. The correctness argument is identical to the argument presented for the crash no-recovery model.

Performance. The algorithm requires at least one communication step to execute the reliable broadcast and at least two communication steps to execute the consensus. Therefore, even if no failures occur, at least three communication steps are required. No stable storage access is needed besides those needed by the underlying consensus module.

Hands-On

In this section, we describe the implementation in *Appia* of the following algorithms introduced in this chapter: no-waiting reliable causal order broadcast, no-waiting reliable causal order broadcast with garbage collection, waiting reliable causal order broadcast, uniform total order broadcast.

No-Waiting Reliable Causal Order Broadcast

The communication stack used to implement the protocol is the following:

SampleApplLayer
CONoWaitingLayer
DelayLayer
RBLayer
PerfectFailureDetectorLayer
BEBLayer
NewTcpLayer

The role of each of the layers is exlained below.

SampleAppllayer This layer performs the user interface, and allows to experiment the implementations. The user indicates to the application which protocol he pretends to test. Then, the application reads the input from the user, builds a corresponding message, SampleSendableEvent, and sends it down to the stack. The application is also responsible for reading the file indicated in the command line and creating the event, ProcessInitEvent, that contains the information about the processes in the group and for sending it down to the stack.

CONoWaitingLayer This layer implements the causal order protocol. Each message, in the protocol, is uniquely identified by its source and a sequence number, as each process in the group has its own sequence number. The events that walk through the stack aren't serializable so we have chosen the relevant information of those events to send, as the past list. A message coming from the protocol looks like Figure 6.5.

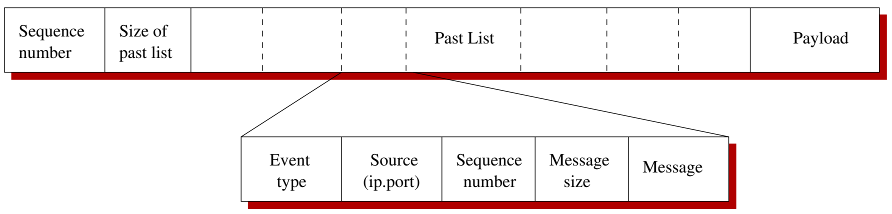

Figure 6.5. Format of messages exchanged by CONoWaiting protocol.

DelayLayer Test protocol used to delay the messages of one process-source when delivering them to one process-destination. This is used to check that messages are really delivered in the right order, even when delays are present. In short, this layer simulates network delays. Note that this layer doesn't belong to the protocol stack, it was just developed for testing.

RBLayer Protocol that implements the reliable broadcast algorithm. The remaining layers are required by this protocol (see Chapter 3).

The protocol implementation is depicted in Listing 6.1.

Listing 6.1. No-waiting reliable causal order broadcast implementation.

```
public class CONoWaitingSession extends Session {
    Channel channel;
    InetWithPort myID;
    int seqNumber=0;
    LinkedList delivered; /* Set of delivered messages */
    LinkedList myPast; /* Set of messages in past */

    public CONoWaitingSession(Layer l) {
        super(l);
    }

    public void handle(Event e) {
        if (e instanceof ChannelInit)
            handleChannelInit((ChannelInit)e);
        else if (e instanceof RegisterSocketEvent)
            handleRegisterSocketEvent((RegisterSocketEvent)e);
        else if (e instanceof SendableEvent){
            if (e.getDirection(). direction==Direction.DOWN)
                handleSendableEventDOWN((SendableEvent)e);
            else
                handleSendableEventUP((SendableEvent)e);
        }
    }

    public void handleChannelInit (ChannelInit e){
        delivered=new LinkedList();
        myPast=new LinkedList();
        this.channel = e.getChannel();
        e.go();
    }

    public void handleRegisterSocketEvent (RegisterSocketEvent e){
        myID=new InetWithPort(InetAddress.getLocalHost(),e.port);
        e.go();
    }

    public void handleSendableEventDOWN (SendableEvent e){
        //cloning the event to be sent in oder to keep it in the mypast list...
        SendableEvent e_aux= (SendableEvent)e.cloneEvent();
        ObjectsMessage om=(ObjectsMessage)e.getMessage();


        //inserting myPast list in the msg:
        for(int k=myPast.size();k>0;k--){
            ObjectsMessage om_k= (ObjectsMessage)((ListElement)myPast.get(k-1)).getSE().getMessage();

            om.push(om_k.toByteArray());
            om.pushInt(om_k.toByteArray().length);
            om.pushInt(((ListElement)myPast.get(k-1)).getSeq());
            InetWithPort.push((InetWithPort)((ListElement)myPast.get(k-1)).getSE().source,om);
            om.pushString(((ListElement)myPast.get(k−1)).getSE().getClass().getName());
        }
        om.pushInt(myPast.size());
        om.pushInt(seqNumber);
        e.setMessage(om);

        e.go();

        //add this message to the myPast list:
        e_aux.source=myID;
        ListElement le=new ListElement(e_aux,seqNumber);
        myPast.add(le);

        //increments the global seq number
        seqNumber++;
    }

    public void handleSendableEventUP (SendableEvent e){
        ObjectsMessage om=(ObjectsMessage)e.getMessage();
        int seq=om.popInt();

        //checks to see if this msg has been already delivered...
        if (! isDelivered ((InetWithPort)e.source,seq)){

            //size of the past list of this msg
            int pastSize=om.popInt();
            for(int k=0;k<pastSize;k++){
                String className=om.popString();
                InetWithPort msgSource=InetWithPort.pop(om);
                int msgSeq=om.popInt();
                int msgSize=om.popInt();
                byte[] msg=(byte[])om.pop();

                //if this msg hasn't been already delivered, we must deliver it prior to the one that just arrived!
                if (! isDelivered (msgSource,msgSeq)){
                    //composing and sending the msg!
                    SendableEvent se=null;
                    se = (SendableEvent) Class.forName(className).newInstance();
                    se.setChannel(channel);
                    se.setDirection(new Direction(Direction.UP));
                    se.setSource(this);

                    ObjectsMessage aux_om = new ObjectsMessage();
                    aux_om.setByteArray(msg,0,msgSize);
                    se.setMessage(aux_om);
                    se.source=msgSource;

                    se. init ();
                    se.go();
                }

                //this msg has been delivered!
                ListElement le=new ListElement(se,msgSeq);
                delivered.add(le);
                myPast.add(le);
            }
        }

        //cloning the event just received to keep it in the mypast list
        SendableEvent e_aux=null;
        e_aux=(SendableEvent)e.cloneEvent();

        // deliver the event
        e.setMessage(om);
        e.go();
        ListElement le=new ListElement(e_aux,seq);
        delivered.add(le);

        if(!e_aux.source.equals(myID))
            myPast.add(le);
    }

    boolean isDelivered(InetWithPort source,int seq){
        for(int k=0;k<delivered.size();k++){
            InetWithPort iwp_aux=(InetWithPort) ((ListElement)delivered.get(k)).getSE().source;
            int seq_aux=((ListElement)delivered.get(k)).getSeq();
            if( iwp_aux.equals(source) && seq_aux==seq)
                return true;
        }
        return false;
    }
}

class ListElement{
    SendableEvent se;
    int seq;

    public ListElement(SendableEvent se, int seq){
        this.se=se;
        this.seq=seq;
    }

    SendableEvent getSE(){
        return se;
    }

    int getSeq(){
        return seq;
    }
}
```

Try It Assuming that the package **order** is loaded into directory **DIR**, follow the next steps:

1. Setup
   a) Open 3 shells/command prompts.
   b) In each shell go to directory **DIR/order**.
   c) In shell 0 execute *java appl/SampleAppl appl/procs 0*.
   d) In shell 1 execute *java appl/SampleAppl appl/procs 1*.
   e) In shell 2 execute *java appl/SampleAppl appl/procs 2*.
   f) A list of the developed protocols should appear in the 3 shells. If this list doesn't appear, check if the error NoClassDefError has appeared instead. If it has, confirm that both packages **order** and **rb** are included in the classpath.
   g) Now, that everything is ok, choose protocol 1 in every shell.
2. Run: Now that processes are launched and running, let's try 2 different executions:
   a) Execution I
      i. In shell 0, send a message **M1** (just write **M1** and press enter).
         - Note that all processes received message **M1**.
      ii. In shell 1, send a message **M2**.

- Note that all processes received message **M2**.

iii. Confirm that all processes have received **M1** and then **M2** and note the continuous growth of the size of the messages sent.

b) Execution II: For this execution it is necessary to first modify file *appl/SampleAppl.java* in package **order**. Line 121 should be uncommented in order to insert a test layer that allows to inject delays in messages sent between process 0 and process 2. After modifying the file, it is necessary to compile it.

i. In shell 0, send a message **M1**.
   - Note that process 2 didn't receive the message **M1**.

ii. In shell 1, send a message **M2**.
   - Note that all processes received **M2**.

iii. Confirm that all processes received **M1** and then **M2**. Process 2 received message **M1** because it was appended to message **M2**.

No-Waiting Reliable Causal Order Broadcast with Garbage Collection

The next protocol to present is an optimization of the previous one. It intends to circunvents its main disadvantage by deleting messages from the past list. When the protocol delivers a message it broadcasts an acknowledgement to all other processes; when an acknowledgement for the same message has been received from all correct processes, this message is removed from the past list.

The communication stack used to implement the protocol is the following:

SampleApplLayer
CONoWaitingLayerGC
DelayLayer
RBLayer
PerfectFailureDetectorLayer
BEBLayer
NewTcpLayer

The protocol implementation is depicted in Listing 6.2.

Listing 6.2. No-waiting reliable causal order broadcast with garbage collection implementation.

```
public class CONoWaitingSessionGC extends Session {
    Channel channel;
    int seqNumber;
    LinkedList delivered; /* delivered messages */
    LinkedList myPast; /* Past set */
    /*
     * Set of the correct processes.
     */
    private ProcessSet correct;
    /*
     * Set of the msgs not yet acked by all correct processes.
     * unit: seq number+source+list of processes acked
    *         (AckElement)
    */
   LinkedList acks;

   public void handle(Event e) {
  if (e instanceof ChannelInit)
      handleChannelInit((ChannelInit)e);
  else if (e instanceof ProcessInitEvent)
      handleProcessInit((ProcessInitEvent)e);
  else if (e instanceof AckEvent)
      handleAck((AckEvent)e);
  else if (e instanceof SendableEvent){
      if (e.getDirection().direction==Direction.DOWN)
   handleSendableEventDOWN((SendableEvent)e);
      else
   handleSendableEventUP((SendableEvent)e);
  }else if (e instanceof Crash)
      handleCrash((Crash)e);
  else{
      try {
   e.go();
      } catch (AppiaEventException ex) {
   System.out.println("[CONoWaitingGCSession:handle]"
          + ex.getMessage());
      }
  }
   }
}
```

Try It Assuming that the package **order** is loaded into directory **DIR**, follow the next steps:

1. Setup
   a) Open 3 shells/command prompts.
   b) In each shell go to directory **DIR/order**.
   c) In shell 0 execute *java appl/SampleAppl appl/procs 0.*
   d) In shell 1 execute *java appl/SampleAppl appl/procs 1.*
   e) In shell 2 execute *java appl/SampleAppl appl/procs 2.*
   f) A list of the developed protocols should appear in the 3 shells. If this list doesn't appear, check if the error NoClassDefError has appeared instead. If it has, confirm that both packages **order** and **rb** are included in the classpath.
   g) Now, that everything is ok, choose protocol 2 in every shell.
2. Run: Now that processes are launched and running, let's try 3 different executions:
   a) Since this protocol is very similar with the previous one, the 2 executions presented in the previous section can be applied to this protocol. Note that the line of code inq *appl/SampleAppl.java* that has to be altered is now line 175.
   b) Execution III:(this execution is to be done with the delay layer in the protocol stack.)
      i. In shell 0, send a message **M1** (just write **M1** and press enter).
         - Note that process 2 didn't receive the message **M1**.
      ii. In shell 1, send a message **M2**.

- Note the size of the message that was sent and note also that all processes received message **M2**.

iii. In shell 2, send a message **M3**.
- Note the smaller size of the message that was sent and note also that all processes received message **M3**.

iv. Confirm that all processes received **M1**, **M2** and **M3** by the right order.

Waiting Reliable Causal Order Broadcast

The communication stack used to implement the protocol is the following:

SampleApplLayer
COWaitingLayer
DelayLayer
RBLayer
PerfectFailureDetectorLayer
BEBLayer
NewTcpLayer

The protocol implementation is depicted in Listing 6.3.

Listing 6.3. Waiting reliable causal order broadcast implementation.

```
public class COWaitingSession extends Session{
    Channel channel;
    /*
    * Set of the correct processes.
    */
    private ProcessSet correct;
    /*
    * The list of the msg that are waiting to be delivered
    * unit: SendableEvent
    */
    private LinkedList pendingMsg;

    int [] vectorClock;

    public void handle(Event e) {
  if (e instanceof ChannelInit)
      handleChannelInit((ChannelInit)e);
  else if (e instanceof ProcessInitEvent)
      handleProcessInit((ProcessInitEvent)e);
  else if (e instanceof SendableEvent){
      if (e.getDirection().direction==Direction.DOWN)
    handleSendableEventDOWN((SendableEvent)e);
      else
    handleSendableEventUP((SendableEvent)e);
  }else{
      try {
    e.go();
      } catch (AppiaEventException ex) {
    System.out.println("[COWaitingSession:handle]"
          + ex.getMessage());
      }
  }
    }
}
```

Try It Assuming that the package **order** is loaded into directory **DIR**, follow the next steps:

1. Setup
   a) Open 3 shells/command prompts.
   b) In each shell go to directory **DIR/order**.
   c) In shell 0 execute *java appl/SampleAppl appl/procs 0*.
   d) In shell 1 execute *java appl/SampleAppl appl/procs 1*.
   e) In shell 2 execute *java appl/SampleAppl appl/procs 2*.
   f) A list of the developed protocols should appear in the 3 shells. If this list doesn't appear, check if the error NoClassDefError has appeared instead. If it has, confirm that both packages **order** and **rb** are included in the classpath.
   g) Now, that everything is ok, choose protocol 3 in every shell.
2. Run: Now that processes are launched and running, let's try 2 different executions:
   a) Execution I
      i. In shell 0, send a message **M1** (just write **M1** and press enter).
         - Note that all processes received message **M1**.
      ii. In shell 1, send a message **M2**.
         - Note that all processes received message **M2**.
      iii. Confirm that all processes received **M1** and then **M2**.
   b) Execution II: For this execution it is necessary to first modify file *appl/SampleAppl.java* in package **order**. Line 228 should be uncommented in order to insert a test layer that allows to inject delays in messages sent between process 0 and process 2. After modifying the file, it is necessary to compile it.
      i. In shell 0, send a message **M1**.
         - Note that process 2 didn't receive the message **M1**.
      ii. In shell 1, send a message **M2**.
         - Note that process 2 also didn't receive **M2**.
      iii. Wait for process 2 to receive the messages.
         - Note that process 2 didn't receive **M1**, immediately, due to the presence of the Delay layer. And it didn't receive **M2**, also immediately, because it had to wait for **M1** to be delivered, as **M1** preceded **M2**.

Uniform Total Order Broadcast

The communication stack used to implement the protocol is the following:

SampleApplLayer
TOUniformLayer
DelayLayer
FloodingUniformLayer
URBLayer
PerfectFailureDetectorLayer
BEBLayer
NewTcpLayer

The protocol implementation is depicted in Listing 6.4.

Listing 6.4. Uniform total order broadcast implementation.

```
public class TOUniformSession extends Session{
    /*my id*/
    InetWithPort iwp;

    Channel channel;

    /*global sequence number of the message sent by
    this process*/

    int seqNumber;

    /*Sequence number of the set of messages to
    deliver in the same round!*/
    int sn;

    /*Sets the beginning and the end of the rounds*/
    boolean wait;

    /*
    * Set of delivered messages.
    * unit: sendable event+seq number (ListElement)
    */
    LinkedList delivered;

    /*
    * Set of unordered messages.
    * unit: sendable event+seq number (ListElement)
    */
    LinkedList unordered;

    public void handle(Event e) {
  if (e instanceof ChannelInit)
      handleChannelInit((ChannelInit)e);
  else if(e instanceof ProcessInitEvent)
      handleProcessInitEvent((ProcessInitEvent)e);
  else if (e instanceof SendableEvent){
      if (e.getDirection(). direction==Direction.DOWN)
    handleSendableEventDOWN((SendableEvent)e);
      else
    handleSendableEventUP((SendableEvent)e);
  }else if (e instanceof ConsensusDecide)
      handleConsensusDecide((ConsensusDecide)e);
  else{
      try {
    e.go();
      } catch (AppiaEventException ex) {
    System.out.println("[TOUniformSession:handle]"
            + ex.getMessage());
      }
  }
    }
}
```

Try It Assuming that the package **order** is loaded into directory **DIR**, follow the next steps:

1. Setup
   a) Open 3 shells/command prompts.
   b) In each shell go to directory **DIR/order**.
   c) In shell 0 execute *java appl/SampleAppl appl/procs 0*.
   d) In shell 1 execute *java appl/SampleAppl appl/procs 1*.
   e) In shell 2 execute *java appl/SampleAppl appl/procs 2*.
   f) A list of the developed protocols should appear in the 3 shells. If this list doesn't appear, check if the error NoClassDefError has appeared instead. If it has, confirm that both packages **order** and **rb** are included in the classpath.
   g) Now, that everything is ok, choose protocol 4 in every shell.
2. Run: Now that processes are launched and running, let's try the following execution:
   a) Execution I: Note that the protocol stack already has the delay layer in order to perform the following execution.
      i. In shell 0, send a message **M1** (just write **M1** and press enter).
         - Note that no process received the message **M1**.
      ii. In shell 1, send a message **M2**.
         - Note that all processes received message **M1**. The consensus decided to deliver **M1**.
      iii. In shell 1, send a message **M3**.
         - Note that all processes received message **M2** and **M3**. Now, the consensus decided to deliver these both messages.
      iv. Confirm that all processes received **M1**, **M2** and **M3** in the same order.
      v. You can keep doing these steps, in order to introduce some delays, and check that all processes receive all messages by the same order.

Exercises

Exercise 6.1 *Compare our causal broadcast property with the following property:* "If a process delivers messages m_1 and m_2, and $m_1 \rightarrow m_2$, then the process must deliver m_1 before m_2".

Exercise 6.2 *Can we devise a best-effort broadcast algroithm that satisfies the* causal delivery *property without being a causal broadcast algorithm, i.e., without satisfying the* agreement *property of a reliable broadcast?*

Exercise 6.3 *Can we devise a broadcast algorithm that does not ensure the* causal delivery *property but only its non-uniform variant: No correct process p_i delivers a message m_2 unless p_i has already delivered every message m_1 such that $m_1 \rightarrow m_2$.*

Exercise 6.4 *Suggest a modification of the garbage collection scheme to collect messages sooner than in Algorithm 6.2.*

Exercise 6.5 *What happens in our total order broadcast algorithm if the set of messages decided on are not sorted deterministically after the decision but prior to the proposal? What happens in our total order broadcast algorithm if the set of messages decided on is not sorted deterministically, neither a priori nor a posteriori?*

Solutions

Solution 6.1 We need to compare the two following properties:

1. If a process delivers a message m_2, then it must have delivered every message m_1 such that $m_1 \rightarrow m_2$.
2. If a process delivers messages m_1 and m_2, and $m_1 \rightarrow m_2$, then the process must deliver m_1 before m_2.

Property 1 says that *any* message m_1 that causally precedes m_2 must only be delivered before m_2 if m_2 is delivered. Property 2 says that *any delivered* message m_1 that causally precedes m_2 must only be delivered before m_2 if m_2 is delivered.

Both properties are safety properties. In the first case, a process that delivers a message m without having delivered a message that causally precedes m violates the property and this is irremediable. In the second case, a process that delivers both messages without respecting the causal precedence might violate the property and this is also irremediable. The first property is however strictly stronger than the second. If the first is satisfied then the second is. However, it can be the case with the second property is satisfied whereas the first is not: a process delivers a message m_2 without delivering a message m_1 that causally precedes m_1. □

Solution 6.2 The answer is no. Assume by contradiction that some broadcast algorithm ensures causal order deliver and is not reliable but best-effort. We define the abstraction implemented by such an algorithm with primitives: coBroadcast and coDeliver. The only possibility for a broadcast to ensure the best-effort properties and not be reliable is to violate the *agreement* property: there must be some execution of the algorithm implementing the abstraction where some correct process p coDelivers a message m that some other process q does never coDeliver. Because the algorithm is best-effort, this can only happen if the original source of the message, say r is faulty.

Assume now that after coDelivering m, process p coBroadcasts a message m'. Given that p is correct and the broadcast is best-effort, all correct processes, including q, coDeliver m'. Given that m precedes m', q must have coDelivered m: a contradiction. Hence, any best-effort broadcast that satisfies the *causal delivery* property is also reliable. □

Solution 6.3 Assume by contradiction that some algorithm does not ensure the *causal delivery* property but ensures its non-uniform variant. This means that the algorithm has some execution where some process p delivers some message m without delivering a message m' that causally precedes m. Given that we assume a model where processes do not commit suicide, p might very well be correct, in which case it violates even the non-uniform variant. □

Solution 6.4 When removing a message m from the past, we can also remove all the messages that causally depend on this message—and then recursively those that causally precede these. This means that a message stored in the past must be stored with its own, distinct past. □

Solution 6.5 If the deterministic sorting is done prior to the proposal, and not a posteriori upon a decision, the processes would not agree on a set but on a sequence, i.e., an ordered set. If they then toDeliver the messages according to this order, we would still ensure the total order property.

If the messages that we agree on through consensus are not sorted deterministically within every batch (neither a priori nor a posteriori), then the total order property is not ensured. Even if the processes decide on the same batch of messages, they might toDeliver the messages within this batch in a different order. In fact, the total order property would only be ensured with respect to the *batches* of messages, and not to the messages themselves. We thus get a coarser granularity in the total order.

We could avoid using the deterministic sort function at the cost of proposing a single message at a time in the consensus abstraction. This means that we would need exactly as many consensus instances as there are messages exchanged between the processes. If messages are generated very slowly by processes, the algorithm ends up using one consensus instance per message anyway. If the messages are generated rapidly, then it is beneficial to use several messages per instance: within one instance of consensus, several messages would be gathered, i.e., every message of the consensus algorithm would concern several messages to toDeliver. Agreeing on several messages at the same time reduces the number of times we use the consensus protocol. □

Historical Notes

- The causal broadcast abstraction was defined by Birman and Joseph in (Birman and Joseph 1987a) following the notion of causality initially introduced by Lamport (Lamport 1978).
- In this chapter, we presented algorithms that implement causal broadcast assuming that all messages are broadcast to all processes in the system. It is also possible to ensure causal delivery in the cases where individual messages may be sent to an arbitrary subset of group members, but the algorithms require a significantly larger amount of control information (Raynal, Schiper, and Toueg 1991).
- Similarly, we considered that messages need to be totally ordered were broadcast to all processes in the system, and hence it was fine to have all the processes participate in the ordering activity. It is also possible to consider a total order multicast abstraction where the sender can select the subset of processes to which the message needs to be sent, and require that no other process besides the sender and the multicast set participates in the ordering. The algorithms in this case are rather tricky (Rodrigues, Guerraoui, and Schiper 1998; Guerraoui and Schiper 2001).
- Our no-waiting causal broadcast algorithm was inspired by one of the earliest implementations of causal ordering, included in the ISIS toolkit (Birman and Joseph 1987b).
- Our waiting causal broadcast algorithms was based on the notion of vector clocks introduced in (Fidge 1988; Ladin, Liskov, Shrira, and Ghemawat 1990; Schwarz and Mattern 1992).
- The total order broadcast abstraction was specified by Schneider (Schneider 1990), following the work on state machine replication by Lamport. Our total order broadcast algorithm is inspired by (Chandra and Toueg 1996).
- Our total order broadcast specification and algorithms in the crash-recovery model are inspired by (Boichat, Dutta, Frolund, and Guerraoui 2003; Rodrigues and Raynal 2003).

.

7. Coordination

I know what you're thinking punk, you're thinking did he fire six shots or only five? Well, to tell you the truth I kinda forgot myself in all this excitement. But being this here's the 44 Magnum, the most powerful handgun in the world and would blow your head clean off, you've got to ask yourself one question, Do I feel lucky? Well do ya punk?.
(Dirty Harry.)

This chapter considers agreement abstractions where, like in consensus, the processes need to decide on a common value. Unlike in consensus however, the value that the processes need to decide on cannot be any value proposed. It should obey some specific coordination requirements.

Examples of abstractions we will study here include *terminating reliable broadcast*, *(non-blocking) atomic commitment*, *leader election*, *group membership*, and *view-synchronous communication.* We will mainly give here fail-stop, consensus-based, algorithms that implement these abstractions. In fact, a common characteristic of these abstractions is that they cannot be implemented in a fail-noisy model, and hence neither in a fail-silent nor in a fail-recovery model. (We will discuss this impossibility through the exercices.) We will also discuss randomized implementations of these algorithms.

7.1 Terminating Reliable Broadcast

7.1.1 Overview

As its name indicates, terminating reliable broadcast is a form of reliable broadcast with a termination property.

To explain the underlying intuition, consider the case where a given process p_i is known to have the obligation of broadcasting some message to all processes in the system. In other words, p_i is a source of information in the system and all processes must perform some specific processing according to the message m to be delivered from p_i. All the remaining processes are thus waiting for p_i's message. If p_i uses a best-effort broadcast and does not crash, then its message m will be seen by all correct processes. Consider now the case where p_i crashed and some process p_j detects that p_i has crashed without having seen m. Does this means that m was not broadcast? Not really. It is possible that p_i crashed while broadcasting m: some processes may have received m whereas others have not. Process p_j needs to know whether it should keep on waiting for m, or if it can know at some point that m will never be delivered by any process.

At this point, one may think that the problem could be avoided if p_i had used a uniform reliable broadcast primitive to broadcast m. Unfortunately, this is not the case. Consider process p_j in the example above. The use of a uniform reliable broadcast primitive would ensure that, if some other process p_k delivered m, then p_j would eventually deliver m also. However, p_j cannot decide if it should wait for m or not.

The *terminating reliable broadcast (TRB)* abstraction ensures precisely that every process p_j either delivers the message m or some indication F that m will not be delivered. This indication is given in the form of a specific message to the processes: it is however assumed that the indication is not like any other message, i.e., it does not belong to the set of possible messages. The TRB abstraction is an agreement one because the processes do all deliver the same message, i.e., either message m or message F.

7.1.2 Specifications

The properties of terminating reliable broadcast abstraction are depicted in Module 7.1. It is important to notice that the abstraction is defined for a specific originator process, denoted by *src* in Module 7.1. A process declares itself as the originator by broadcasting a message m and indicating itself as the source. A process indicates that it participates in the terminating reliable broadcast by broadcasting an empty message. We consider the uniform variant of the problem where agreement is uniformly required among any pair of processes, be them correct or faulty.

7.1.3 Fail-Stop Algorithm: Consensus-Based TRB

Algotithm 7.1 implements TRB using three underlying abstractions: a perfect failure detector, a uniform consensus and a best-effort broadcast.

The algorithm works by having the source of the message m disseminate m to all correct processes using a best-effort broadcast. Every process waits until it either gets the message broadcast by the sender process or it detects the crash of the originator process. The assumption of a perfect failure detector and the validity property of the broadcast ensures that the process cannot

Module:

Name: TerminatingReliableBroadcast (trb).

Events:

Request: ⟨ *trbBroadcast*, src, m ⟩: Used to initiate a terminating reliable broadcast for process *src*.

Indication: ⟨ *trbDeliver*, src, m ⟩: Used to deliver message m broadcast by process *src* (or F in the case *src* crashes).

Properties:

TRB1: *Termination:* Every correct process eventually delivers exactly one message.

TRB2: *Validity:* If the sender *src* is correct and broadcasts a message m, then *src* eventually delivers m.

TRB3: *Integrity:* If a correct process delivers a message m then either $m = F$ or m was previously broadcast by *src*.

TRB5: *Uniform Agreement:* If any process delivers a message m, then every correct process eventually delivers m.

Module 7.1 Interface and properties of terminating reliable broadcast.

wait forever. Then all processes run a consensus to agree on whether to deliver m or a to deliver the failure notification F. The processes that got m propose it to consensus and those who detected the crash of the sender, *src*, propose F. The result of the consensus is the value delivered by the TRB algorithm.

An execution of the algorithm is illustrated in Figure 7.1. Process p_1 crashes while broadcasting m. Therefore p_2 and p_3 get m but p_4 does not. The remaining processes use the consensus module to decide which value must be delivered. In the example of the figure the processes decide to deliver m but F would be also a possible outcome (since p_1 has crashed).

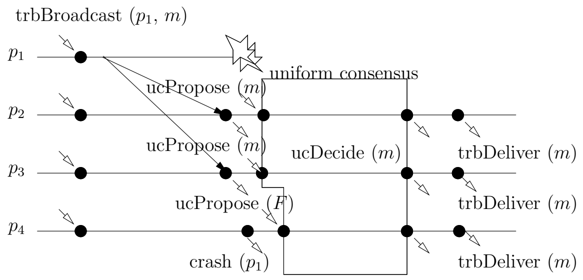

Figure 7.1. Sample execution of terminating reliable broadcast.

Correctness. The *integrity* property of best-effort broadcast and the *validity* property of consensus ensure that if a process trbDelivers a message m, then

Algorithm 7.1 Consensus-based terminating reliable broadcast algorithm.

```
Implements:
    TerminatingReliableBroadcast (trb).

Uses:
    BestEffortBroadcast (beb).
    UniformConsensus (uc);
    PerfectFailureDetector ($\mathcal{P}$);

upon event ⟨ Init ⟩ do
    proposal := src := $\perp$;
    correct := $\Pi$;

upon event ⟨ crash, $p_i$ ⟩ do
    correct := correct $\setminus\{p_i\}$;

upon event ⟨ trbBroadcast, $p_i$, m ⟩ do
    src := $p_i$;
    if (src = self) then trigger ⟨ bebBroadcast, $m$ ⟩;

upon event ⟨ bebDeliver, src, $m$ ⟩ do
    proposal := m;

upon (src $\notin$ correct) do
    proposal := $F_{src}$;

upon (proposal $\neq \perp$) do
    trigger ⟨ ucPropose, proposal ⟩;

upon event ⟨ ucDecide, decided ⟩ do
    trigger ⟨ trbDeliver, src, decided ⟩
```

either m is F or m was trbBroadcast by *src*. The *no-duplication* property of best-effort broadcast and the *integrity* property of consensus ensure that no process trbDelivers more than one message. The *completeness* property of the failure detector, the *validity* property of best-effort broadcast and the *termination* property of consensus ensure that every correct process eventually trbDelivers a message. The *agreement* property of consensus ensures that of terminating reliable broadcast. Consider now the *validity* property of terminating reliable broadcast. Consider that *src* does not crash and trbBroadcasts a message $m \neq F$. By the *accuracy* property of the failure detector, no process detects the crash of *src*. By the *validity* property of best-effort broadcast, every correct process bebDelivers m and proposes m to consensus. By the *termination* property of consensus, all correct processes, including *src*, eventually decide and trbDeliver a message m.

Performance. The algorithm requires the execution of the consensus abstraction. In addition to the cost of consensus, the algorithm exchanges $N-1$ messages and requires one additional communication step (for the initial best-effort broadcast).

Variation. Our TRB specification has a uniform agreement property. As for reliable broadcast, we could specify a regular variant of TRB with a regular agreement property. By using a regular consensus abstraction instead of uniform consensus, we can automatically obtain a regular terminating reliable broadcast abstraction.

7.2 Non-blocking Atomic Commit

7.2.1 Overview

The unit of data processing in a distributed information systems is the *transaction.* This can be viewed as a portion of a program delimited by two primitives: *begin* and *end.* The transaction is typically expected to be *atomic* in two senses. (1. Concurrency atomicity) Transactions appear to execute one after the other and this *serialisability* is usually guaranteed using some form of *distributed locking* or some form of *optimistic concurrency control.* (2. Failure atomicity) Every transaction appears to execute either completely (it said to commit) or not at all (it is said to abort). Ensuring this form of atomicity in a distributed environment is not trivial because the transaction might have accessed information on different processes (i.e., different data managers) and these need to make sure that they all discard this information or they all make it visible. In other words, they all need to agree on the same outcome for the transaction. An additional difficulty lies here in the facts that some processes might not be able to commit the transaction (as we discuss below).

The *non-blocking atomic commit* (NBAC) abstraction is precisely used to solve this problem in a reliable way. The processes, each representing a data manager, agree on the outcome of a transaction. The outcome is either to *commit* the transaction, say to decide 1, or to *abort* the transaction, say to decide 0. The outcome depends of the initial proposals of the processes. Every process proposes an initial vote for the transaction: 0 or 1. Voting 1 for a process means that the process is willing and able to commit the transaction.

- Typically, by voting 1, a process witnesses the absence of any problem during the execution of the transaction. Furthermore, the process promises to make the update of the transaction permanent. This in particular means that the process has stored the temporary update of the transaction in stable storage: should it crash and recover, it can install a consistent state including all updates of the committed transaction.
- By voting 0, a data manager process vetos the commitment of the transaction. Typically, this can occur if the process cannot commit the transaction

Module:

Name: Non-Blocking Atomic Commit (nbac).

Events:

Request: ⟨ *nbacPropose*, v ⟩: Used to propose a value for the commit (0 or 1).

Indication: ⟨ *nbacDecide*, v ⟩: Used to indicate the decided value for nbac.

Properties:

NBAC1: Agreement No two processes decide different values.

NBAC2: Integrity No process decides twice.

NBAC3: Abort-Validity 0 can only be decided if some process proposes 0 or crashes.

NBAC4: Commit-Validity 1 can only be decided if no process proposes 0.

NBAC5: Termination Every correct process eventually decides.

Module 7.2 Interfaces and properties of NBAC.

for an application-related reason, e.g., not enough money for a bank transfer in a specific node, for a concurrency control reason, e.g., there is a risk of violating serialisability in a database system, or a storage reason, e.g., the disk is full and there is no way to guarantee the persistence of the transaction's updates.

At first glance, the problem looks like consensus: the processes propose 0 or 1 and need to decide on a common final value 0 or 1. There is however a fundamental difference: in consensus, any value decided is valid as long as it was proposed. In the atomic commitment problem, the decision 1 cannot be taken if any of the processes had proposed 0. It is indeed a veto right that is expressed with a 0 vote.

7.2.2 Specifications

NBAC is characterized by the properties listed in Module 7.2. Without the termination property, the abstraction is simply called *atomic commit* (or *atomic commitment*).

7.2.3 Fail-Stop Algorithm: Consensus-Based NBAC

Algorithm 7.2 implements NBAC using three underlying abstractions: a perfect failure detector, a consensus and a best-effort broadcast.

The algorithm works as follows. Every process p_i broadcasts its proposal (0 or 1) to all, and waits, for every process p_j, either to get the proposal of p_j or to detect the crash of p_j. If p_i detects the crash of any other process or gets a

Algorithm 7.2 Consensus-based non-blocking atomic commit algorithm.

```
Implements:
    NonBlockingAtomicCommit (nbac).

Uses:
    BestEffortBroadcast (beb).
    Consensus (uc);
    PerfectFailureDetector ($\mathcal{P}$);

upon event ⟨ Init ⟩ do
    delivered := $\emptyset$;
    correct := $\Pi$;
    proposal := 1;

upon event ⟨ crash, $p_i$ ⟩ do
    correct := correct $\setminus\{p_i\}$;

upon event ⟨ nbacPropose, $v$ ⟩ do
    trigger ⟨ bebBroadcast, $v$ ⟩;

upon event ⟨ bebDeliver, $p_i$, $v$ ⟩ do
    delivered := delivered $\cup\{p_i\}$ ;
    proposal := proposal * $v$;

upon (correct \ delivered = $\emptyset$) do
    if correct $\neq \Pi$ then
        proposal := 0;
    trigger ⟨ ucPropose, proposal ⟩;

upon event ⟨ ucDecide, decided ⟩ do
    trigger ⟨ nbacDecide, decided ⟩
```

proposal 0 from any process, then p_i invokes consensus with 0 as its proposal. If p_i gets the proposal 1 from all processes, then p_i invokes consensus with 1 as a proposal. Then the processes decide for NBAC following the outcome of consensus.

Correctness. The *agreement* property of NBAC directly follows from that of consensus. The *no-duplication* property of best-effort broadcast and the *integrity* property of consensus ensure that no process nbacDecides twice. The *termination* property of NBAC follows from the *validity* property of best-effort broadcast, the *termination* property of consensus, and the *completeness* property of the failure detector. Consider now the *validity* properties of NBAC. The *commit-validity* property requires that 1 is decided only if all processes propose 1. Assume by contradiction that some process p_i nbacProposes 0 whereas some process p_j nbacDecides 1. By the algorithm, for p_j to nbacDecide 1, it must have decided 1, i.e., through the consensus abstraction. By the *validity* property of consensus, some process p_k must have proposed 1 to the consensus abstraction. By the *validity* property of best-effort broadcast, there are two cases to consider: (1) either p_i crashes before p_k bebDelivers p_i's proposal or (2) p_k bebDelivers p_i's proposal. In both cases, by the algorithm, p_k proposes 0 to consensus: a contradiction. Consider now the *abort-validity* property of NBAC. This property requires that 0 is decided only if some process nbacProposes 0 or crashes. Assume by contradiction that all processes nbacPropose 1 and no process crashes, whereas some process p_i nbacDecides 0. For p_i to nbacDecide 0, by the *validity* property of consensus, some process p_k must propose 0. By the algorithm and the *accuracy* property of the failure detector, p_k would only propose 0 if some process nbacProposes 0 or crashes: a contradiction.

Performance. The algorithm requires the execution of the consensus abstraction. In addition to the cost of consensus, the algorithm exchanges N^2 messages and requires one additional communication step (for the initial best-effort broadcast).

Variation. One could define a non-uniform variant of NBAC, i.e., by requiring only *agreement* and not *uniform agreement.* However, this abstraction would not be useful in a practical setting to control the termination of a transaction in a distributed database system. Indeed, the very fact that some process has decided to commit a transaction is important, say the process has delivered some cash through an ATM. Even if that process has crashed, its decision is important and other processes should reach the same outcome.

7.3 Leader Election

7.3.1 Overview

The leader election abstraction consists in choosing one process to be selected as a unique representative of the group of processes in the system. For this abstraction to be useful in a distributed setting, a new leader should be elected if the current leader crashes. Such abstraction is in case indeed useful in a primary-backup replication scheme for instance. Following this scheme, a set of replica processes coordinate their activities to provide the illusion of a unique fault-tolerant (highly-available) service. Among the set of replica processes, one is chosen as the leader. This leader process, sometimes called the primary, is supposed to treat the requests submitted by the client processes, on behalf of the other replicas, called backups. Before a leader returns a reply to a given client, it updates its backups to keep them up to date. If the leader crashes, one of the backups is elected as the new leader, i.e., the new primary.

7.3.2 Specification

We define the leader election abstraction more precisely through a specific primtive, denoted by *leLeader* which, when invoked on a process at some

Module:

Name: LeaderElection (le).

Events:

Indication: ⟨ *leLeader* ⟩p_i: Used to indicate that process p_i is now the leader.

Properties:

LE1: Either there is no correct process, or some correct process is eventually the leader.

LE2: If a process p_i is leader, then all other leaders have crashed.

Module 7.3 Interface and properties of leader election.

given time, means that the process is elected leader from that time on. The properties of the abstraction are given in Module7.3.

The first property ensures the eventul presence of a correct leader. Clearly, it might be the case that, at some point in time, no process is leader. It might also be the case that no leader is up. The property ensures however that, unless there is no correct process, some correct process is eventually elected leader. The second property ensures the stability of the leader. In other words, it ensures that the only reason to change the leader is if it crashes. Indirectly, this property precludes the possibility for two processes to be leader at the same time. In this sense, the leader election abstraction we consider here is strictly stronger than the eventually accurate leader election notion we introduced earlier in the manuscript.

7.3.3 Fail-Stop Algorithm: Monarchical Leader Election

Algorithm 7.3 implements the leader election abstraction assuming a perfect failure detector. The algorithm assumes the existence of a total order among processes agreed on a priori. This is encapsulated by some function O. (This could also be known to the user of the leader election abstraction, e.g., the clients of a primary-backup replication scheme, for optimization purposes).

This function O associates to every process, those that precede it in the ranking. A process can only become leader if those that precede it have crashed. Think of the function as representing the royal ordering in a monarchical system. The prince becomes leader if and only if the queen dies. If the prince dies, may be his little sister is the next on the list, etc. Typically, we would assume that O(p_1) = $\emptyset$, O(p_2) = $\{p_1\}$, O(p_3) = $\{p_1, p_2\}$, and so forth. The order in this case is $p_1; p_2; p_3; ...$

Correctness. Property *LE1* follows from the *completeness* property of the failure detector whereas property *LE2* follows from the *accuracy* property of the failure detector.

Algorithm 7.3 Monarchical leader election algorithm.

```
Implements:
    LeaderElection (le);

Uses:
    PerfectFailureDetector (P);

upon event ⟨ Init ⟩ do
    suspected := $\emptyset$;

upon event ⟨ crash, $p_i$ ⟩ do
    suspected := suspected $\cup\{p_i\}$;

upon event O(self) $\subset$ suspected do
    trigger ⟨ leLeader, self ⟩;
```

Performance. The process of becoming a leader is a local operation. The time to react to a failure and become the new leader depends on the latency of the failure detector.

7.4 Group Membership

7.4.1 Overview

In the previous sections, our algorithms were required to make decisions based on the information about which processes were operational or crashed. This information is provided by the failure detector module available at each process. However, the output of failure detector modules at different processes is not coordinated. This means that different processes may get notifications of failures of other processes in different orders and, in this way, obtain a different perspective of the system evolution. One of the roles of a membership service is to provide consistent information about which processes have crashed and which processes have not.

Another role of a membership service is to allow new processes to leave and join the set of processes that are participating in the computation, or let old processes voluntarily leave this set. As with failure information, the result of leave and join operations should be provided to correct processes in a consistent way.

To simplify the presentation, we will consider here just the case of process crashes, i.e., the initial membership of the group is the complete set of processes and subsequent membership changes are solely caused by crashes. Hence, we do not consider explicit join and leave operations.

Module:

Name: Membership (memb).

Events:

Indication: $\langle$ *membView*, g, V^i $\rangle$ Used to deliver update membership information in the form of a *view*. The variable g denotes the group id. A view V^i is a tuple (i, M), where i is a unique view identifier and M is the set of processes that belong to the view.

Properties:

Memb1: *Self inclusion:* If a process p installs view $V^i = (i, M_i)$, then $p \in M_i$.

Memb2: *Local Monotonicity:* If a process p installs view $V^j = (j, M_j)$ after installing $V^i = (i, M_i)$, then $j > i$.

Memb3: *Initial view:* Every correct process installs $V^0 = (0, \Pi)$.

Memb4: *Agreement:* If a correct process installs V^i, then every correct process also installs V^i.

Memb5: *Completeness:* If a process p crashes, then eventually every correct process installs $V^i = (i, M_i)$ with $p \notin M_i$.

Memb6: *Accuracy:* If some process installs a view $V^i = (i, M_i)$ and $q \notin M_i$, then q has crashed.

Module 7.4 Interface and properties of group membership.

7.4.2 Specification

We name the set of processes that participate in the computation a *group*. The current membership of the group is called a *group view*. Each view $V^i = (i, M_i)$ is a tuple that contains a unique view identifier i and a set of member processes M. We consider here a *linear group membership* service, where all correct processes see the same sequence of views: $V^0 = (0, M_0), V^1 = (1, M_1), \ldots$. As we have noted before, the initial view of all processes V^0 includes the complete set of processes Π in the system. A process that delivers a view V^i is said to *install* view V^i. The membership abstraction is characterized by the properties listed in Module 7.4.

7.4.3 Fail-Stop Algorithm: Consensus-Based Group Membership

Algorithm 7.4 implements the group membership abstraction assuming a consensus and a perfect failure detector. At initialization, each process installs the initial view with all the processes in the system. From that point on, the algorithm remains idle until a process is detected to have crashed. Since different processes may detect crashes in different orders, a new view is not generated immediately. Instead, a consensus is executed to decide which processes are to be included in the next view. The *wait* flag is used to prevent

Algorithm 7.4 Consensus-based group membership algorithm.

```
Uses:
    UniformConsensus (uc);
    PerfectFailureDetector ($\mathcal{P}$);

upon event ⟨ Init ⟩ do
    current-id := 0;
    current-membership := $\Pi$;
    next-membership := $\Pi$;
    current-view := (current-id, current-membership);
    wait := false;
    trigger ⟨ memView, $g$, current-view ⟩;

upon event ⟨ crash, $p_i$ ⟩ do
    next-membership := next-membership $\setminus\{p_i\}$;

upon (current-membership $\neq$ next-membership) $\wedge$ ($\neg$ wait) do
    wait := true;
    trigger ⟨ ucPropose, current-id+1, next-membership ⟩;

upon event ⟨ ucDecided, id, memb ⟩ do
    current-id := id;
    current-membership := memb;
    next-membership := current-membership $\cap$ next-membership;
    current-view := (current-id, current-membership);
    wait := false;
    trigger ⟨ membView, $g$, current-view ⟩
```

a process to start a new consensus before the previous consensus terminates. When consensus decides, a new view is delivered and the *current-membership* and *next-membership* are updated. Note that a process p_i may install a view containing a process that p_i already knows to be crashed. In this case, p_i will, immediately after installing that view, initiate a new consensus to trigger the installation of another view.

An execution of the membership algorithm is illustrated in Figure 7.2. In the execution, both p_1 and p_2 crash. Process p_3 detects the crash of p_2 and initiates the consensus to define a new view. Process p_4 detects the crash of p_1 and proposes a different view to consensus. As a result of the first consensus, p_1 is excluded from the view. Since p_3 has already detected the crash of p_2, p_3 starts a new consensus to exclude p_2. Eventually, p_4 also detects the crash of p_2 and also participates in the consensus for the third view. This view only includes the correct processes.

Correctness. self inclusion, *local monotonicity*, and *initial view* follow from the algorithm. The *agreement* property follows from consensus. The *completeness* property follows from the *completeness* property of the failure detector and the *accuracy* property follows from the *accuracy* property of the failure detector.

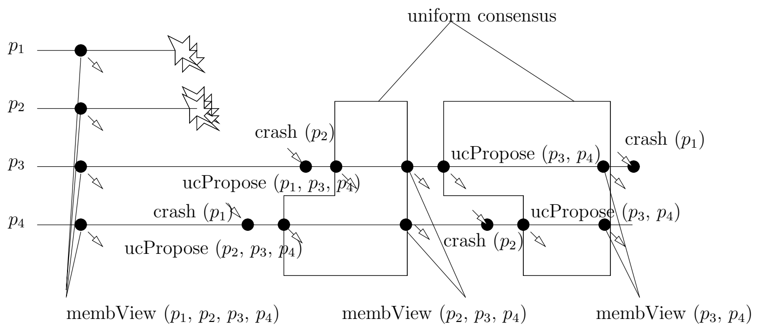

Figure 7.2. Sample execution of the membership algorithm.

Performance. The algorithm requires at most one consensus execution for each process that crashes.

7.5 View-Synchronous Communication

7.5.1 Overview

Early in the book we have introduced the problem of reliable broadcast (see Section 3.3). We recall here that the goal of reliable broadcast if to ensure that if a message is delivered to a process, then it is delivered to all correct processes (uniform definition). In the previous section, we have introduced a membership service; this service is responsible for providing to the application information about which processes are correct and which have failed. We now discuss the subtle issues that arise when you try to combine these two abstractions!

Let us assume that processes of a group g are exchanging messages and that one of these processes, say f, fails. Assume also that this failure is detected and that the membership service intalls a new view $V^i = (i, M_i)$ such $f \notin M_i$. Assume now that, after V^i has been installed, at some process p arrives a message m originally sent by f. Note that such a run is possible as, in order to ensure reliably delivered, messages originally sent by f can be relayed by other processes. Clearly, it is counter-intuitive for the application programmer to receive a message from a process f, after f has been declared to be failed and expelled from the group view. Therefore, it would be desirable to discard m. Unfortunatly, it may also happen that some other process q has already delivered m *before* delivering view V^i. So, in this scenario, one may be faced with two conflicting goals: To provide reliable broadcast the message must be delivered at q but, at the same time, to be consistent with the view information, the message should be discarded at q!

To solve this contradiction one needs a new abstraction. In fact, what is needed is to ensure that the instalation of views is ordered with regard to the message flow. If a message m is delivered to a correct process before the instalation of V^i, it should be delivered before the view change to all processes that install V^i. If, by any chance, m would be delivered after the instalation of V^i, then m is simply discarded at all processes. The abstraction that preserves this ordering constraint is called view-synchrony (Birman and Joseph 1987b), as it gives the illusion that failures are synchronous, i.e., they occur at the same point in time with regard to the message flow. Note that the combination of properties VS1 and VS2 eliminates the contradiction raised earlier.

7.5.2 Specification

View synchronous communication is characterized by the properties depicted in Mod.7.5 (in addition to the properties already state for reliable broadcast). In these properties we state that a process *delivers (sends) a message m in view V^i* if it delivers (sends) the message m after installing view V^i and before installing V^{i+1}.

In the specification we introduce an alternative definition of view synchrony, captured by property VS2' (Sending view delivery). This alternative definition, called *Strong View Synchrony* (Friedman and van Renesse 1995), offers stronger guarantees as it ensures that messages are always delivered in the view they were sent. Note that the view inclusion property allows for messages to be sent in a view and to be delivered in a later view, as long as the sender remains correct.

To implement strong view synchrony, the interface with the upper layers need to be augmented to prevent the transmission of new messages whenever a new view needs to be installed. Note that since messages have to be delivered in the view they are sent, if new messages are continuously sent then the installation of the new view may be indefinitely postponed. Therefore, the interface of a view synchronous service includes three new events: *block*, *block-ok* and *block-release*. The *block* event is used by the view synchronous service to request the upper layers to stop sending messages in the current view. The *block-ok* event is used by the client of the view synchronous service to acknowledge the *block* request. The *block-release* event is used by the view synchronous service to notify the upper layers that messages canbe sent again.

Note that with regard to the properties enforced on the messages exchanged, the properties of view synchrony can be combined with the properties of regular and uniform reliable broadcast and, optionally, with the properties of causal order, resulting in different possible flavors of virtually synchronous communication. For instance, Mod. 7.6 depicts the combination of view synchrony with regular reliable causal order.

Module:

Name: ViewSynchrony (vs).

Events:

Request: $\langle$ *vsBroadcast*, g, m $\rangle$: Used to broadcast message m to a group g.

Indication: $\langle$ *vsDeliver*, g, src, m $\rangle$: Used to deliver message m sent by process src in group g.

Indication: $\langle$ *vsView*, g, V^i $\rangle$ Used to deliver update membership information in the form of a *view*. A view V^i is a tuple (i, M), where i is a unique view identifier and M is the set of processes that belong to the view.

Indication: $\langle$ *vsBlock*, g $\rangle$ Used by the view synchronous layer to block the application.

Request: $\langle$ *vsBlockOk*, g $\rangle$: Used by the application to confirm the transmission of new messages will be temporarily blocked.

Indication: $\langle$ *vsRelease*, g $\rangle$ Used by the view synchronous layer to release a communication in the group.

Properties:

VS1: *Same view delivery:* If two processes p and q both receive message m, they receive m in the same view.

VS2: *View inclusion:* If a process p receives in view V^i a message m from q , then $q \in V^i$.

VS2': *Sending view delivery:* If a process p receives a message m from q in view V^i, then m was sent by q in view V^i.

Module 7.5 Interface and properties of view synchronous communication.

7.5.3 Fail-Stop Algorithm: TRB-Based View-Synchrony

An algorithm to implement regular causal strong view synchronous communication is depicted in Alg. 7.5. The algorithm works by performing a *flush* procedure before installing each new view. The flush procedure requires coordination among processes in a view and it is possible that new views arrive before the flush is complete. Therefore, new views are kept in a list of *pending-views*.

The flush is initiated by requesting the application to stop sending messages in the view. When this requests is granted, the process stops delivering new messages from the underlying reliable causal order module. Then, using a terminating reliable broadcast, it sends to the remaining members of the group the set of messages that have been delivered up to that point. All the remaining processes to do same. Eventually, when all TRBs are concluded, each process has the set of messages delivered by every other correct process.

Module:

Name: (regular) CausalStrongViewSynchrony (csvs).

Events:

⟨ *csvsBroadcast*, g, m ⟩, ⟨ *csvsDeliver*, g, src, m ⟩, ⟨ *vsView*, g, V^i ⟩, ⟨ *vs-Block*, g ⟩, ⟨ *vsBlockOk*, g ⟩, ⟨ *vsRelease*, g ⟩: with the same meaning and interface of view synchronous communication.

Properties:

VS1, VS2': from view synchronous communication.

RB1-RB4: from reliable broadcast.

CB1-CB2: from causal order.

Module 7.6 Interface and properties of (regular) causal strong VS.

A union of all these sets is taken as the set of messages to be delivered before a new view is installed.

An example of the execution of the algorithm is presented in Fig.7.3. Process p_1 sends messages m_1 and m_2 before crashing. Message m_1 is delivered by p_3 and p_4 but not by p_2. On the other hand, m_2 is delivered by p_2 but not by p_3 and p_4. There is also a third message that is delivered by all correct processes before the flush procedure is initiated. When the underlying membership module delivers a new view, excluding p_1 from the group, a TRB is initiated for each process in the previous view. Each TRB includes the set of messages that have been locally delivered. For instance, the TRB from p_2 includes m_1 and m_3 since m_2 has not yet been delivered. The union of these sets, $\{m_1, m_2, m_3\}$ is the set of messages that have to be delivered before installing the new view. Note that m_1 is eventually delivered to p_2 by the underlying reliable broadcast module, but it will be discarded (the same will happen to m_2 with regard to p_3 and p_4).

The reader must be aware that this algorithm is not very efficient, as it requires the parallel execution of several instances of consensus (one for each process in the view). It is possble to optimize the algorithm by running a single instance of consensus to agree both on the new membership and on the set of messages to be delivered before the group change.

Correctness. Upon transmission, a message is immediately delivered to its sender. On the other hand, a view is not installed until a correct process stops sending messages in the previous view. This ensures the sending view delivery property. If a message m is delivered to a process p and p remain correct, p will add m to the set of messages to e delivered before the next view. According to the TRB properties, all correct processes will receive this set, and merge it with the corresponding sets received from other correct processes. Since the union of these sets is delivered before the installation of the new view, both same view delivery and the reliability guarantees are ensured.

Algorithm 7.5 Regular causal strong view synchronous communication.

```
Implements:
    (regular) CausalStrongViewSynchrony (csvs);

Uses:
    TerminatingReliableBroadcast (trb); Membership (memb); ReliableCausalOrder (rco);

upon event ⟨ Init ⟩ do
    pending-views := $\emptyset$; delivered := $\emptyset$; trb-done := $\emptyset$;
    current-view := $\perp$; next-view := $\perp$; flushing := false; blocked := true;

upon event ⟨ memView, $V^i = (i, M)$ ⟩ do
    if $i = 0$ do //first view
        current-view := $V^0$; blocked := false; trigger ⟨ vsView, $g$, $V^0$ ⟩; trigger ⟨ vsRelease, $g$ ⟩
    else
        addToTail ( pending-views, $V^i$) ;

upon event ⟨ csvsBroadcast, $g, m$ ⟩ $\wedge\neg$ blocked do
    delivered := delivered $\cup$ {( self, $m$ )};
    trigger ⟨ csvsDeliver, $g$, self, $m$ ⟩;
    trigger ⟨ rcoBroadcast, [DATA, current-view.id, m] ⟩;

upon event ⟨ rcoDeliver, src$_m$,[DATA, vid, m] ⟩ do
    if (current-view.id = vid) $\wedge$ ((src$_m$, $m$) $\notin$ delivered) do
        delivered := delivered $\cup$ {( src$_m$, $m$ )};
        trigger ⟨ csvsDeliver, $g$, src$_m$, $m$ ⟩;

upon (pending-views $\neq \emptyset$) $\wedge$ (flushing = false) do
    next-view := removeFromHead (pending-views); flushing := true;
    trigger ⟨ vsBlock, $g$ ⟩;

upon event ⟨ vsBlockOk, $g$ ⟩ do
    blocked := true; trb-done := $\emptyset$;
    forall $p_i \in$ current-view.memb do
        if $p_i$ = self do trigger ⟨ trbBroadcast, current-view.id, $p_i$, delivered ⟩;
        else trigger ⟨ trbBroadcast, current-view.id, $p_i$, $\perp$ ⟩;

upon event ⟨ trbDeliver, vid, $p_i$, del ⟩ do
    trb-done := trb-done $\cup\{p_i\}$;
    forall (src$_m$, $m$) $\in$ del: (src$_m$, $m$) $\notin$ delivered do
        delivered := delivered $\cup$ { (src$_m$, $m$) };
        trigger ⟨ csvsDeliver, $g$, src$_m$, $m$ ⟩;

upon ( trb-done = current-view.memb) $\wedge$ blocked do
    current-view := next-view; flushing := false; blocked := false;
    trigger ⟨ vsView, $g$, current-view ⟩; trigger ⟨ vsRelease, $g$ ⟩
```

Performance. The algorithm requires the parallel execution of a TRB for each process in the old view in order to install the new view.

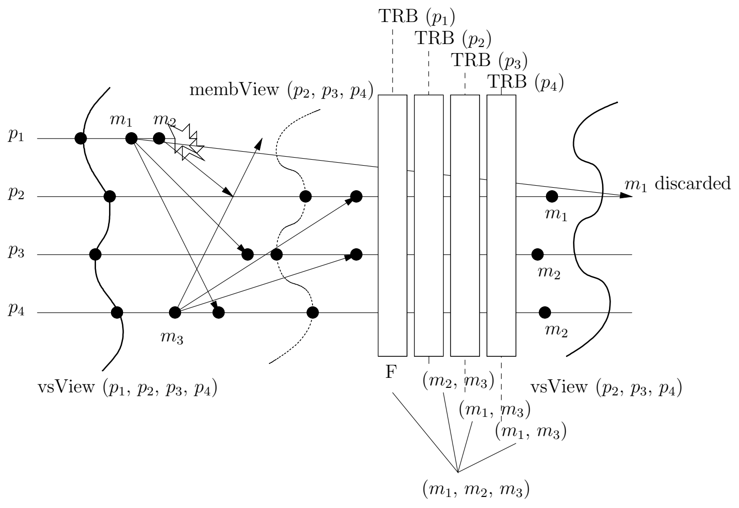

Figure 7.3. Sample excution of view change algorithm.

7.6 Probabilistic Partial Membership

Both probabilistic broadcast algorithms presented in Chapter 3 so far require processes to randomly select, among the complete set of processes in the system, a subset of k processes to gossip messages to. Therefore, the algorithms implicitly assume that each process has a global knowledge of the complete membership of the processes in the system. This assumption has two main limitations:

- In large scale systems, it is not realistic to assume that the membership does not change. Assuming a dynamic membership would mean here that every process should keep track of all changes in the global system.
- One of the strongest advantages of probabilistic approaches is their scalability. However, when considering very large systems, it may not be practical, or even feasible, to store the complete membership of the system at each process.

Fortunately, probabilistic approaches can also be used to manage the membership information. One possible solution consists of having each process store just a *partial* view of the complete membership. For instance, every process would store a fixed number of processes in the system, i.e., every process would have a set of acquaintances. This is also called the view of the process.

7.6.1 Specification

Naturally, to ensure connectivity, views of different processes must overlap at least at one process, and a larger overlap is desirable for fault-tolerance.

Furthermore, the union of all views should include all processes in the system. If processes in the system are uniformly distributed among the views, it can be shown that a probabilistic broadcast algorithm preserves the same properties as an algorithm that relies on full membership information.

To be completed

7.6.2 Randomized Algorithm: Probabilistic Broadcast with Partial Membership

The problem is then to derive an algorithm that allows new processes to join the system and that promotes an uniform distribution of processes among the view. The basic idea consists of having processes gossip information about the contents of their local views. The nice thing is that this (partial) membership information may be piggybacked in data messages. Using this information, processes may "mix" their view with the views of the processes they receive messages from, by randomly discarding old members and inserting new members.

Algorithm 7.6 illustrates the idea. It works by managing three variables: *view*, that maintains the partial membership; *subs* that maintains a set of processes that are joining the membership; and, *unsubs*, a set of processes that want to leave the membership. Each of these sets has a maximum size, *viewsz*, *subssz*, and *unsubssz* respectively. If during the execution of the algorithm, these sets become larger than the maximum size, elements are removed at random until the maximum size is reached. Processes periodically gossip (and merge) their *subs* and *unsubs* sets. The partial view is updated according to the information propagated in these sets. Note that, as a result of new subscriptions, new members are added and some members are randomly removed from the partial view. Members removed from the partial view, say due to the overflow of the table where each process stores the identities of its acquaintances, are added to the *subs* set, allowing them to be later inserted in the partial view of other members. It is of course assumed that each process is initialized with a set of known group members.

It is important to notice that the probabilistic partial membership algorithm can be viewed as an auxiliary service of the probabilistic broadcast service presented above. When the two algorithms are used in combination, the variable *view* of Algorithm 7.6 replaces the set Π in Algorithm 3.9. Additionally, membership information can simply be piggybacked as control information in the packets exchanged and part of the data gossiping activity.

Algorithm 7.6 A randomized partial membership algorithm.

```
Implements:
      Probabilistic Partial Membership (ppm).

Uses:
      unreliablePointToPointLinks (up2p).

upon event ⟨ Init ⟩ do
      view := set of known group members;
      subs := ∅; unsubs := ∅;

every T units of time do
      for 1 to fanout do
            target := random (view);
            trigger ⟨ upp2pSend, target, [GOSSIP, subs, unsubs] ⟩;

upon ⟨ ppmJoin ⟩ do
      subs := subs ∪ { self };

upon ⟨ ppmLeave ⟩ do
      unsubs := unsubs ∪ { self };

upon event ⟨ up2pDeliver, $p_i$, [GOSSIP, s, u] ⟩ do
      view := view \ u;
      view := view ∪ s \ { self };
      unsubs := unsubs ∪ u;
      subs := subs ∪ s \ { self };
      //trim variables
      while | view | > viewsz do
            target := random (view);
            view := view \ { target };
            subs := subs ∪ { target };
      while | unsubs | > unsubssz do unsubs := unsubs \ { random(unsubs) };
      while | subs | > subssz do subs := subs \ { random(subs) };
```

Hands-On

To-be-done. To-be-done.

To-be-done. To-be-done.

To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done.

To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done.

To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done. To-be-done.

Exercices

Exercise 7.1 *Can we implement TRB with the eventually perfect failure detector $\diamond\mathcal{P}$ if we assume that at least one process can crash?*

Exercise 7.2 *Do we need the perfect failure detector $\mathcal{P}$ to implement TRB (assuming that any number of processes can crash and every process can trbBroadcast messages)?*

Exercise 7.3 *Devise two algorithms that, without consensus, implement weaker specifications of NBAC where we replace the* termination *property with the following ones:*

- (1) weak termination*: let p_i be some process: if p_i does not crash then all correct processes eventually decide;*
- (2) very weak termination*: if no process crashes, then all processes decide.*

Exercise 7.4 *Can we implement NBAC with the eventually perfect failure detector $\diamond\mathcal{P}$ if we assume that at least one process can crash? What if we consider a weaker specification of NBAC where the* agreement *was not required?*

Exercise 7.5 *Do we need the perfect failure detector $\mathcal{P}$ to implement NBAC if we consider a system where at least two processes can crash but a majority is correct?*

Exercise 7.6 *Do we need the perfect failure detector $\mathcal{P}$ to implement NBAC if we assume that at most one process can crash?*

Exercise 7.7 *Consider a specification of leader election where we require that (1)* there cannot be two leaders at the same time *and (2)* either there is no correct process, or some correct process is eventually leader*. Is this specification sound? e.g., would it be useful for a primary-backup replication scheme?*

Exercise 7.8 *What is the difference between the specification of leader election given in the core of the chapter and a specification with the two properties of the previous exercice and the following property: (3)* (stability) a leader remains leader until it crashes.

Exercise 7.9 *Do we need the perfect failure detector $\mathcal{P}$ to implement a general leader election abstraction where we could choose in advance the order according to which the processes should be elected leaders?*

Solutions

Solution 7.1 No. Consider TRB$_i$, i.e., the sender is process p_i. We discuss below why it is impossible to implement TRB$_i$ with $\diamond\mathcal{P}$ if one process can crash. Consider an execution E_1 where process p_i crashes initially and consider some correct process p_j. By the *termination* property of TRB$_i$, there must be a time T at which p_j trbDelivers F_i. Consider an execution E_2 that is similar to E_1 up to time T, except that p_i is correct: p_i's messages are delayed until after time T and the failure detector behaves as in E_1 until after time T. This is possible because the failure detector is only eventually perfect. Up to time T, p_j cannot distinguish E_1 from E_2 and trbDelibevers F_i. By the *agreement* property of TRB$_i$, p_i must trbDeliver F_i as well. By the *termination* property, p_i cannot trbDeliver two messages and will contadict the *validity* property of TRB$_i$. □

Solution 7.2 We explain below that if we have TRB$_i$ abstractions, for every process p_i, and if we consider a model where failures cannot be predicted, then we can *emulate* a perfect failure detector. This means that the perfect failure detector is not only sufficient to solve TRB, but also necessary. The *emulation* idea is simple. Every process trbBroadcasts a series of messages to all processes. Every process p_j that trbDelivers F_i, suspects process p_i. The *strong completeness* property would trivially be satisfied. Consider the *strong accuracy* property (i.e., no process is suspected before it crashes). If p_j trbDelivers F_i, then p_i is faulty. Given that we consider a model where failures cannot be predicted, p_i must have crashed. □

Solution 7.3 The idea of the first algorithm is the following. It uses a perfect failure detector. All processes bebBroadcast their proposal to process p_i. This process would collect the proposals from all that it does not suspect and compute the decision: 1 if all processes propose 1 and 0 otherwise, i.e., if some process proposes 0 or is suspected to have crashed. Then p_i bebBroadcasts the decision to all and decide. Any process that bebDelivers the message decides accordingly. If p_i crashes, then all processes are blocked. Of course, the processes can figure out the decision by themselves if p_i crashes after some correct process has decided, or if some correct process decides 0. However, if all correct processes propose 1 and p_i crashes before any correct process, then no correct process can decide.

This algorithm is also called the *Two-Phase Commit (2PC)* algorithm. It implements a variant of atomic commitment that is *blocking.*

The second algorithm is simpler. All processes bebBroadcast their proposals to all. Every process waits from proposals from all. If a process bebDelivers 1 from all it decides 1, otherwise, it decides 0. (This algorithm does not make use of any failure detector.) □

Solution 7.4 No. The reason is similar to that of exercice 7.4. Consider an execution E_1 where all processes are correct and propose 1, except some process p_i which proposes 0 and crashes initially. By the *abort-validity* property, all correct processes decide 0. Let T be the time at which one of these processes, say p_j, decides 0. Consider an execution E_2 that is similar to E_1 except that p_i proposes 1. Process p_j cannot distinguish the two executions (because p_i did not send any message) and decides 0 at time T. Consider now an execution E_3 that is similar to E_2, except that p_i is correct but its messages are all delayed until after time T. The failure detector behaves in E_3 as in E_2: this is possible because it is only eventually perfect. In E_3, p_j decides 0 and violates *commit-validity*: all processes are correct and propose 1.

In this argumentation, the *agreement* property of NBAC was not explicitly needed. This shows that even a specification of NBAC where *agreement* was not needed could not be implemented with an eventually perfect failure detector if some process crashes. □

Solution 7.5 If we assume that a minority of processes can crash, then the perfect failure detector is not needed. To show that, we exhibit a failure detector that, in a precise sense, is strictly weaker than the perfect failure detector and that helps solving NBAC.

The failure detector in question is denoted by $?P$, and called the *anonymously perfect* perfect failure detector. This failure detector ensures the *strong completess* and *eventual strong accuracy* of an eventually perfect failure detector, plus the following *anonymous detection* property: every correct process suspects outputs a specific value F iff some process has crashed.

Given that we assume a majority of correct processes, then failure detector $?P$ implements uniform consensus and we can build a consensus module. Now we give the idea of an algorithm that uses $?P$ and a consensus module to implement NBAC.

The idea of the algorithm is the following. All processes bebBroadcast their proposal to all. Every process p_i waits either (1) to bebDeliver 1 from all processes, (2) to bebDeliver 0 from some process, or (3) to output F. In case (1), p_i invokes consensus with 1 as a proposed value. In cases (2) and (3), p_i invokes consensus with 0. Then p_i decides the value output by the consensus module.

Now we discuss in which sense ?P is strictly weaker than P. Assume a system where at least two processes can crash. Consider an execution E_1 where two processes p_i and p_j crash initially and E_2 is an execution where only p_i initially crashes. Let p_k be any correct process. Using ?P, at any time T, process p_k can confuse executions E_1 and E_2 if the messages of p_j are delayed. Indeed, p_k will output F and know that some process has indeed crashed but will not know which one.

Hence, in a system where two processes can crash but a majority is correct, then P is not needed to solve NBAC. There is a failure detector that is strictly weaker and this failure detector solves NBAC. □

Solution 7.6 We show below that in a system where at most one process can crash, we can emulate a perfect failure detector if we can solve NBAC. Indeed, the processes go through sequential rounds. In each round, the processes bebBrodcast a message *I-Am-Alive* to all and trigger an instance of NBAC (two instances are distinguished by the round number at which they were triggered). In a given round r, every process waits to decide the outcome of NBAC: if this outcome is 1, then p_i moves to the next round. If the outcome is 0, then p_i waits to bebDeliver $N-1$ messages and suspects the missing message. Clearly, this algorithm emulates the behavior of a perfect failure detector P in a system where at most one process crashes. □

Solution 7.7 The specification looks simple but is actually bogus. Indeed, nothing prevents an algorithm from changing leaders all the time: this would comply with the specification. Such a leader election abstraction would be useless, say for a primary-backup replication scheme, because even if a process is leader, it would not know for how long and that would prevent it from treating any request from the client. This is because we do not explicitly handle any notion of time. In this context, to be useful, a leader must be *stable*: once it is elected, it should remain leader until it crashes. □

Solution 7.8 A specification with properties (1), (2) and (3) makes more sense but still has an issue: we leave it up to the algorithm that implements the leader election abstraction to choose the leader. In practice, we typically expect the clients of a replicated service to know which process is the first leader, which is the second to be elected if the first has crashed, etc. This is important for instance in failure-free executions where the clients of a replicated service would consider sending their requests directly to the actual leader instead of broadcasting the requests to all, i.e., for optimization issues. Our specification, given in the core of the chapter, is based on the knowledge of an ordering function that the processes should follow in the leader election process. This function is not decided by the algorithm and can be made available to the client of the leader election abstraction. □

Solution 7.9 Yes. More precisely, we discuss below that if we have a leader election abstraction, then we can emulate a perfect failure detector. This means that the perfect failure detector is not only sufficient to solve leader election, but also necessary. The *emulation* idea is simple. Every process p_i triggers $N-1$ instances of leader election, each one for a process p_j different from p_i. In instance j, $O(p_j) = \emptyset$ and $O(p_i) = \{p_j\}$, for every $p_j \neq p_i$. Whenever p_i is elected leader in some instance j, p_i accurately detects the crash of

p_j. $\square$

Historical Notes

- The atomic commit problem was introduced by Gray (Gray 1978), together with the two-phase commit algorithm, which we studied in the exercice section.
- The non-blocking atomic commit (NBAC) problem was introduced by Skeen (Skeen 1981). The NBAC algorithm presented in the chapter is a modular variant of his decentralized three-phase. It is more modular in the sense that we encapsulate many tricky issues of NBAC within consensus.
- The terminating broadcast problem, discussed in (Hadzilacos and Toueg 1994), is a variant of the Byzantine Generals problem (Lamport, Shostak, and Pease 1982). Whereas the original Byzantine Generals problem consider processes that might behave in an arbitrary manner and be, in particular, malicious, the terminating broadcast problem assumes that processes may only fail by crashing.
- The group membership problem was initially discussed by Birman and Joseph in the context of the Isis system (Birman and Joseph 1987a).

References

Alpern, B. and F. Schneider (1985). Defining lineness. Technical Report TR85-650, Cornell University.

Amir, Y., D. Dolev, S. Kramer, and D. Malki (1992, July). Ytransis: A communication sub-system for high availability. In *22nd Annual International Symposium on Fault-Tolerant Computing (FTCS), Digest of Papers*, pp. 76–84. IEEE.

Attiya, H., A. Bar-Noy, and D. Dolev (1995, June). Sharing memory robustly in message passing systems. *Journal of the ACM 1*(42).

Ben-Or, M. (1983). Another advantage of free choice: Completely asynchonous agreement protocols. In *Proceedings of 2nd ACM Symposium on Principles of Distributed Computing (PODC'83)*, Montreal, Canada, pp. 27–30.

Birman, K., M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky (1999, May). Bimodal multicast. *ACM Transactions on Computer Systems 17*(2).

Birman, K. and T. Joseph (1987a, February). Reliable communication in the presence of failures. *ACM Transactions on Computer Systems 1*(5).

Birman, K. and T. Joseph (1987b, February). Reliable Communication in the Presence of Failures. *ACM, Transactions on Computer Systems 5*(1).

Boichat, R., P. Dutta, S. Frolund, and R. Guerraoui (2001, January). Deconstructing paxos. Technical Report 49, Swiss Federal Institute of Technology in Lausanne, CH 1015, Lausanne.

Boichat, R., P. Dutta, S. Frolund, and R. Guerraoui (2003, March). Deconstructing paxos. In *ACM SIGACT News Distributed Computing Colomn*, Number 34 (1).

Chandra, T., V. Hadzilacos, and S. Toueg (1996). The weakest failure detector for consensus. *Journal of the ACM*.

Chandra, T. and S. Toueg (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM 43*(2), 225–267.

Cherriton, D. and W. Zwaenepoel (1985, May). Distributed process groups in the v kernel. *ACM Transactions on Computer Systems 3*(2).

Delporte-Gallet, C., H. Fauconnier, and R. Guerraoui (2002, October). Failure detection lower bounds on consensus and registers. In *Proc. of the International Conference on Distributed Computing Systems (DISC'02).*

Dutta, D. and R. Guerraoui (2002, July). The inherent price of indulgence. In *Proc. of the ACM Symposium on Principles of Distributed Computing (PODC'02).*

Dwork, C., N. Lynch, and L. Stockmeyer (1988, April). Consensus in the presence of partial synchrony. *Journal of the ACM 35*(2), 288–323.

Eugster, P., R. Guerraoui, and P. Kouznetsov (2004, March). Delta reliabile broadcast: A probabilistic measure of broadcast reliability. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS 2004)*, Tokyo, Japan.

Eugster, P., S. Handurukande, R. Guerraoui, A.-M. Kermarrec, and P. Kouznetsov (2001, July). Lightweight probabilistic broadcast. In *Proceedings of The Interna-*

tional Conference on Dependable Systems and Networks (DSN 2001), Goteborg, Sweden.

Ezhilchelvan, P., A. Mostefaoui, and M. Raynal (2001, May). Randomized multivalued consensus. In *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, Magdeburg, Germany.

Felber, P. and R. Guerraoui (2000, March). Programming with object groups in corba. *IEEE Concurrency 8*(1), 48–58.

Fidge, C. (1988). Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference*.

Fischer, M., N. Lynch, and M. Paterson (1985, April). Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinery 32*(2), 374–382.

Friedman, R. and R. van Renesse (1995, March). Strong and weak virtual synchrony in horus. Technical Report 95-1537, Department of Computer Science, Cornell University.

Golding, R. and D. Long (1992, October). Design choices for weak-consistency group communication. Technical Report UCSC–CRL–92–45, University of California Santa Cruz.

Gray, C. and D. Cheriton (1989, December). Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, Litchfield Park, Arizona, pp. 202–210.

Gray, J. (1978). Notes on database operating systems. *Lecture Notes in Computer Science*.

Guerraoui, R. (2000, July). Indulgent algorithms. In *Proc. of the ACM Symposium on Principles of Distributed Computing (PODC'00)*.

Guerraoui, R. and R. Levy (2004, March). Robust emulations of a shared memory in a crash-recovery model. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS 2004)*, Tokyo, Japan.

Guerraoui, R. and A. Schiper (2001). Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science 254*, 297–316.

Gupta, I., A.-M. Kermarrec, and A. Ganesh (2002, October). Adaptive and efficient epidemic-style protocols for reliable and scalable multicast. In *Proceedings of Symposium on Reliable and Distributed Systems (SRDS 2002)*, Osaka, Japan.

Hadzilacos, V. (1984). Issues of fault tolerance in concurrent computations. Technical Report 11-84, Harvard University, Ph.D thesis.

Hadzilacos, V. and S. Toueg (1994, May). A modular approach to fault-tolerant broadcast and related problems. Technical Report 94-1425, Cornell University, Dept of Computer Science, Ithaca, NY.

Herlihy, M. and J. Wing (1990, July). Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 3*(12).

Israeli, A. and m. M. Li (1993). Bounded timestamps. *Distributed Computing 4*(6), 205–209.

Kaashoek, F., A. Tanenbaum, S. Hummel, and H. Bal (1989, October). An efficient reliable broadcast protocol. *Operating Systems Review 4*(23).

Kermarrec, A.-M., L. Massoulie, and A. Ganesh (2000, October). Reliable probabilistic communication in large-scale information dissemination systems. Technical Report MMSR-TR-2000-105, Microsoft Reserach, Cambridge, UK.

Kouznetsov, P., R. Guerraoui, S. Handurukande, and A.-M.Kermarrec (2001, October). Reducing noise in gossip-based reliable broadcast. In *Proceedings of the 20th Symposium on Reliable Distributed Systems (SRDS 2001)*, NewOrleans,USA.

Ladin, R., B. Liskov, L. Shrira, and S. Ghemawat (1990). Lazy replication: Exploiting the semantics of distributed services. In *Proceedings of the Ninth Annual ACM Symposium of Principles of Distributed Computing*, pp. 43–57.

Lamport, L. (1977). Concurrent reading and writing. *Communications of the ACM 11*(20), 806–811.

Lamport, L. (1978, July). Time, clocks and the ordering of events in a distributed system. *Communications of the ACM 21*(7), 558–565.

Lamport, L. (1986a). On interprocess communication, part i: Basic formalism. *Distributed Computing 2*(1), 75–85.

Lamport, L. (1986b). On interprocess communication, part ii: Algorithms. *Distributed Computing 2*(1), 86–101.

Lamport, L. (1989, May). The part-time parliament. Technical Report 49, Digital, Systems Research Center, Palo Alto, California.

Lamport, L., R. Shostak, and M. Pease (1982, July). The byzantine generals problem. *ACM Transactions on Prog. Lang. and Systems 4*(3).

Lin, M.-J. and K. Marzullo (1999, September). Directional gossip: Gossip in a wide area network. In *Proceedings of 3rd European Dependable Computing Conference*, pp. 364–379.

Lynch, N. and A. Shvartsman (1997). Robust emulation of shared memory using dynamic quorum acknowledged broadcasts. In *Proc. of the International Symposium on Fault-Tolerant Computing Systems (FTCS'97)*.

Lynch, N. and A. Shvartsman (2002, October). Rambo: A reconfigurable atomic memory service for dynamic networks. In *Proc. of the International Conference on Distributed Computing Systems (DISC'02)*.

Miranda, H., A. Pinto, and L. Rodrigues (2001, April). Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, Phoenix, Arizona, pp. 707–710. IEEE.

Moser, L., P. Melliar-Smith, A. Agarwal, R. Budhia, C. Lingley-Ppadopoulos, and T. Archambault (1995, June). The totem system. In *Digest of Papers of the 25th International Symposium on Fault-Tolerant Computing Systems*, pp. 61–66. IEEE.

Neiger, G. and S. Toueg (1993, April). Simulating synchronized clocks and common knowledge in distributed systems. *Journal of the ACM 2*(40).

Peterson, G. (1983). Concurrent reading while writing. *ACM Transactions on Prog. Lang. and Systems 1*(5), 56–65.

Peterson, L., N. Bucholz, and R. Schlichting (1989). Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems 7*(3), 217–246.

Powell, D., P. Barret, G. Bonn, M. Chereque, D. Seaton, and P. Verissimo (1994). The delta-4 distributed fault-tolerant architecture. *Readings in Distributed Systems, IEEE, Casavant and Singhal (eds)*.

Raynal, M., A. Schiper, and S. Toueg (1991, September). The causal ordering abstraction and a simple way to implement it. *Information processing letters 39*(6), 343–350.

Rodrigues, L., R. Guerraoui, and A. Schiper (1998). Scalable atomic multicast. In *IEEE Proc. of IC3N'98*.

Rodrigues, L., S. Handurukande, J. Pereira, R. Guerraoui, and A.-M. Kermarrec (2003, June). Adaptive gossip-based broadcast. In *Proceedings of the IEEE International Symposium on Dependable Systems and Networks*.

Rodrigues, L. and M. Raynal (2003). Atomic broadcast in asynchronous crash-recovery distributed systems and its use in quorum-based replication. *IEEE Transactions on Knowledge and Data Engineering 15*(4).

Rodrigues, L. and P. Veríssimo (1992, October). xAMp: a Multi-primitive Group Communications Service. In *Proceedings of the 11th Symposium on Reliable Distributed Systems (SRDS'11)*, Houston, Texas, pp. 112–121. IEEE.

Schneider, F. (1987). Decomposing properties into safety and lineness. Technical Report TR87-874, Cornell University.

Schneider, F. (1990). Implementing fault-tolerant services with the state machine approach. *ACM Computing Surveys* (22 (4)), 300–319.

Schneider, F., D. Gries, and R. Schlichting (1984). Fault-tolerant broadcasts. *Science of Computer Programming* (4), 1–15.

Schwarz, R. and F. Mattern (1992, February). Detecting causal relationships in distributed computations: In search of the holy grail. Technical report, Univ. Kaiserslautern, Kaiserslautern, Germany.

Shao, C., E. Pierce, and J. Welch (2003, October). Multi-writer consistency conditions for shared memory objects. In *Proceedings of the 17th Symposium on Distributed Computing (DISC 2003)*, Sorrento,Italy.

Skeen, D. (1981, July). A decentralized termination protocol. In *Proceedings of the 1st Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, USA. IEEE.

van Renesse, T. Birman, K. and S. Maffeis (1996, April). Horus: A flexible group communication system. *Communications of the ACM 4*(39).

Vidyasankar, K. (1988, August). Converting lamport's regular register to atomic register. *Information Processing Letters* (28).

Vidyasankar, K. (1990, June). Concurrent reading while writing revisited. *Distributed Computing 2*(4).

Vitanyi, P. and B. Awerbuch (1986). Atomic shared register by asynchronous hardware. In *Proc. of the IEEE Symposium on Foundations of Computer Science (FOCS'86)*, pp. 233–243.

Wensley, J. e. a. (1978, October). The design and analysis of a fault-tolerant computer for air craft control. *IEEE 10*(66).

Xiao, Z., K. Birman, and R. van Renesse (2002, June). Optimizing buffer management for reliable multicast. In *Proceedings of The International Conference on Dependable Systems and Networks (DSN 2002)*, Washington, USA.