Rachid Guerraoui, Luís Rodrigues

Abstractions for Distributed Programming

(Preliminary Draft)

October 12, 2003

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 under different distributed environments. The reader is provided with an insight on the fundamental 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. A 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.

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.

The 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.

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. Many exercises are rather easy and can be discussed within an undergraduate teaching classroom. Some exercises are more difficult and need more time.

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, can be abstracted away when reasoning about the distributed algorithms. Elementary notions of algorithmic, first order logics, programming languages 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 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 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 is indeed on abstraction specifications and algorithms, not on calculability and complexity. There is indeed no theorem in this manuscript. Correctness arguments are given with the aim of better understanding the algorithms: they are not formal correctness proofs per see. 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. 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* about the underlying distributed environment. Basically, we present the basic abstractions on which more sophisticated ones are built. This chapter should 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 *storage* abstractions which encapsulate simple forms of distributed memory objects with read-write semantics. 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 and describe algorithms that have a set of processes decide on a common value, based on some initial values, despite the crash of some of the processes.

- In Chapter 6 we consider *ordering* abstractions. In particular, we discuss how consensus can be used to ensure totally ordered delivery of messages broadcast to a group of processes. We also discuss how such an abstraction makes it easy to implement sophisticated forms of shared distributed objects, beyond read-write storage objects.
- In Chapter 7 we gather what we call *coordination* abstractions, namely, leader election, terminating reliable broadcast, non-blocking atomic commit and group membership.

References

We have been exploring the world of distributed computing 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 *school*, including Ken Birman, Tushar Chandra, Vassos Hadzilacos, Prasad Jayanti, Robert van Renessee, Fred Schneider, and Sam Toueg, for their seminal work on various forms of distributed agreement.

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 trace the history of the concepts presented in the chapter, 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 the topics, and mention major technical conferences in the area for the latest research results.

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 provided useful feedback.

Partha Dutta, Corine Hari, Ron Levy, Petr Kouznetsov and Bastian Pochon, PhD 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, PhD students at the Distributed Systems Laboratory of the University of Lisboa (UL), suggested many improvements to the algorithms presented in the manuscript.

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, 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

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 to 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 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 sig-

nificant 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 action 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, distribution 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 may be 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 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 fail and be 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* ab-

straction 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 or 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.

On the other hand, 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.

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

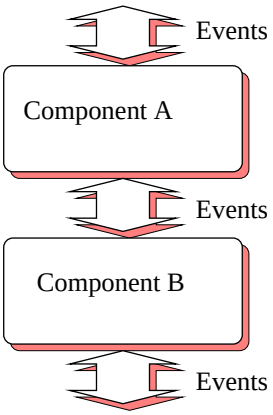


Figure 1.1. Composition model

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 $\langle$ *Event1*, att_1^1, att_1^2, ... $\rangle$ **do**
    something
    // send some event
    trigger $\langle$ *Event2*, att_2^1,att_2^2, ... $\rangle$;

upon event $\langle$ *Event3*, att_3^1, att_3^2, ... $\rangle$ **do**
    something else
    // send some other event
    trigger $\langle$ *Event4*, att_4^1, att_4^2, ... $\rangle$;

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.

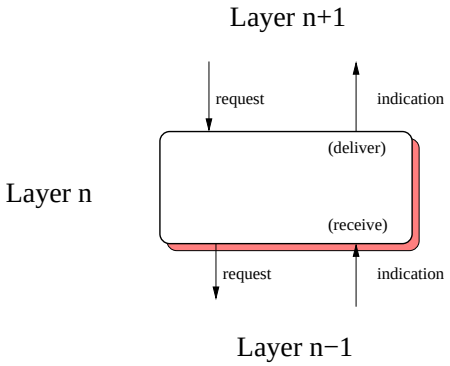


Figure 1.2. Layering

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.

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 ⟩;

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.

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

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 ⟩;

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 four different classes of algorithmic solutions to implement our distributed programming abstractions, namely: *fail-stop* algorithms, where processes can fail by crashing but the crashes can be reliably detected by all the other processes; *fail-silent* algorithms where process crashes cannot always be reliably detected; *crash-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 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-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. This is for instance the case for randomized solutions to the shared memory abstractions we consider in Chapter 4.

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.

2. Basic Abstractions

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 example, 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 distributed automata, one per process. The automata 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 is one that 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 automata, i.e., the algorithm. Local events that are generated are typically those exchanged between modules of the same process at different layers.

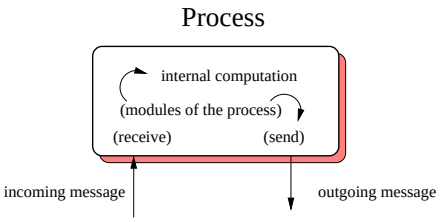


Figure 2.1. Step of a process

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 destinator process, then the destinator 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 received.
  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).

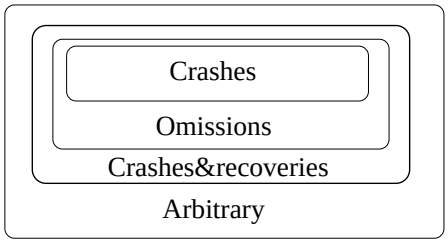


Figure 2.2. Failure modes of a process

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 fault 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. It is common to talk here about a *crash failure* (Figure 2.2), and a *crash stop* process abstraction. 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 are 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 *fail-recovery* one; we also talk about a *fail-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.

According to the fail-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 looses 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 $\langle$ *Recovery* $\rangle$ event is supposed to be automatically generated by the run-time environment in a similar manner to the $\langle$ *Init* $\rangle$ event, executed each time a process starts executing some algorithm. The processing of the $\langle$ *Recovery* $\rangle$ 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 $\langle$ *Init* $\rangle$ event is considered atomic with respect to recovery. More precisely, if a process crashes in the middle of its initialization procedure and recovers without having processed the $\langle$ *Init* $\rangle$ event properly, the process should redo again the $\langle$ *Init* $\rangle$ procedure. On the other hand, if the $\langle$ *Init* $\rangle$ event was processed entirely, then the process must handle the $\langle$ *Recovery* $\rangle$ event instead.

In some sense, a fail-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 fail-recovery abstraction is to minimize the access to stable storage.

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

- One way to alleviate the need for accessing any form of stable storage is to assume that some of the processes do never crash (during the lifetime of an algorithm execution). This might look contradictory with the actual motivation for introducing the fail-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 fail-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 fail-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.

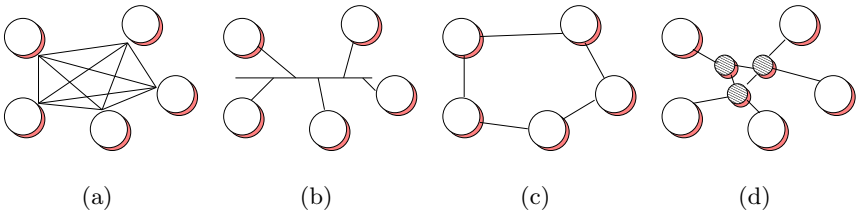


Figure 2.3. The link abstraction and different instances.

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

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 or p_j crash, then m is delivered infinitely often 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.

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. 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

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 eventually 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.

Algorithm 2.1 Stubborn links using fair-loss links.

Implements:
StubbornPointToPointLink (sp2p).

Uses:
FairLossPointToPointLinks (flp2p).

upon event ⟨ *sp2pSend*, dest, m ⟩ **do**
    while (true) **do**
        trigger ⟨ *flp2pSend*, dest, m ⟩;

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

the processes themselves. Finally, the *no creation* property ensures that no message is created or corrupted by the network.

2.3.3 Stubborn Links

We define the abstraction of *stubborn* channels in Module 2.2. 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 destinator processes.

Algorithm 2.1 describes a very simple implementation of stubborn links over fair-loss ones. 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 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" 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 describes a very simple implementation of perfect links over stubborn ones. 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

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.

Algorithm 2.2 Perfect links using stubborn links.

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**
trigger ⟨ *pp2pDeliver*, src, m ⟩;
else delivered := delivered ∪{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, and which clearly apply to the perfect link implementation of Algorithm 2.2, 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 fail-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 fail-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 however not 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 fail-recovery abstraction, a process is said to be correct if, eventually, it is up and does not crash anymore.

Interestingly, Algorithm 2.1 implements stubborn links over fair loss ones also with the fail-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, i.e., this algorithm is not correct with the fail-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.

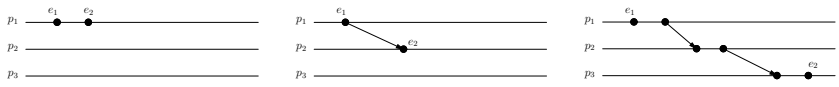
2.4.1 Asynchronous System

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 there are no 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.



(a) (b) (c)

Figure 2.4. The *happened-before* relation.

- 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)).
- 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 processing.* 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 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 measure 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. This is not a significant problem when δ can be made small enough such that only concurrent events (i.e., events that are not causally related) can have the same timestamp.

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, (2) nor does it mean that 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 Failure Detection

2.5.1 Abstracting Time

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 clearly 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 t 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

Module:

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

Events:

Indication: $\langle$ *crash*, p_i $\rangle$: Used to notify that process p_i has crashed.

Properties:

PFD1: *Eventual 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.

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 procedure. 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. The act of detecting a crash coincides with the triggering of the event *crash* (Module 2.4): 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 implements a perfect failure detector assuming a synchronous system. Communication links do not lose messages sent by a

Algorithm 2.3 Perfect failure detector with perfect links and timeouts.

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

Uses:
PerfectPointToPointLinks (pp2p).

upon event $\langle$ *Init* $\rangle$ **do**
alive := Π;

upon event $\langle$ *TimeDelay* $\rangle$ **do**
$\forall_{p_i \in \Pi}$:
if $p_i \notin$ *alive* **then**
trigger $\langle$ *crash*, p_i $\rangle$;
alive := $\emptyset$;
$\forall_{p_i \in \Pi}$: **trigger** $\langle$ *pp2pSend*, p_i, [DATA, heartbeat] $\rangle$;

upon event $\langle$ *pp2pDeliver*, src, [DATA, heartbeat] $\rangle$ **do**
alive := alive $\cup$ $\{src\}$;

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 *TimeDelay* event is triggered.

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 after 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

assumotions 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 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. A process p is said to be *suspected* by process q whenever q triggers the event *suspect*(p_i) and does not trigger the event *restore*(p_i).

Algorithm. Algorithm 2.4 implements an eventually perfect failure detector assuming a partially synchronous system. As for Algorithm 2.3, 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 of Algorithm 2.3. 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 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

Module:

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

Events:

Indication: $\langle$ *suspect*, p_i $\rangle$: Used to notify that process p_i is suspected to have crashed.

Indication: $\langle$ *restore*, p_i $\rangle$: Used to notify that process p_i is not suspected anymore.

Properties:

EPFD1: *Eventual 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.

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 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 fail-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 fail-recovery abstraction, that at least one process

Algorithm 2.4 Eventually perfect failure detector with perfect links and timeouts.

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

Uses:
PerfectPointToPointLinks (pp2p).

upon event $\langle$ *Init* $\rangle$ **do**
alive := Π;
suspected := $\emptyset$;

upon event $\langle$ *TimeDelay* $\rangle$ **do**
$\forall_{p_i \in \Pi}$:
if $p_i \notin$ *alive* **then**
suspected := suspected $\cup \{p_i\}$;
trigger $\langle$ *crash*, p_i $\rangle$;
else
if $p_i \in$ *suspected* **then**
suspected := suspected $\setminus \{p_i\}$;
TimeDelay := TimeDelay + Δ;
trigger $\langle$ *restore*, p_i $\rangle$;
alive := $\emptyset$;
while (true) **do**
$\forall_{p_i \in \Pi}$: **trigger** $\langle$ *pp2pSend*, p_i, [DATA, heartbeat] $\rangle$;

upon event $\langle$ *pp2pDeliver*, src, [DATA, heartbeat] $\rangle$ **do**
alive := alive $\cup \{src\}$;

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.

does never crash, or eventually recovers and never crashes again (in every execution of the algorithm). It is pretty obvious that, without such assumption, no algorithm can implement Ω with the process fail-recovery abstraction.

In Algorithm 2.5, every process p_i keeps track of how many times it crashed and recovered, within an *epoch* integer variable. This variable, rep-

resenting the *epoch number* of p_i, is retrieved, incremented, and then stored in stable storage whenever p_i recovers from a crash. 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 *possible*. Initially, at every process p_i, *possible* contains all processes. Then any process that does not communicate in a timely manner with p_i is excluded from *possible*. A process p_j that communicates in a timely manner with p_i, after having recovered or being slow in communicating with p_i, is simply added again to *possible*, 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 (*alive*), 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 fail-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 (*possible*) 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

Algorithm 2.5 Eventually leader election with fail-recovery processes, fair loss links and timeouts .

Implements:
EventualLeaderDetector (Ω).

Uses:
FairLossPointToPointLinks (flp2p).

upon event $\langle$ *Init* $\rangle$ **do**
leader := p_1;
possible := Π;
epoch := 0;

upon event $\langle$ *Recovery* $\rangle$ **do**
retrieve(epoch);
epoch := epoch + 1;
store(epoch);

upon event $\langle$ *TimeDelay* $\rangle$ **do**
if *leader* $\neq$ select(possible) **then**
TimeDelay := TimeDelay + Δ;
leader := select(possible);
trigger $\langle$ *trust*, leader $\rangle$;
possible := $\emptyset$;
while (true) **do**
$\forall_{p_i \in \Pi}$: **trigger** $\langle$ *flp2pSend*, p_i, [DATA, heartbeat,epoch] $\rangle$;

upon event $\langle$ *flp2pDeliver*, src, [DATA, heartbeat,epc] $\rangle$ **do**
possible := possible $\cup$ $\{(src, epc)\}$;

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 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 does, 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 four 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 four specific combinations to define four 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 (Module 2.4). As the reader will have the opportunity to observe, when comparing algorithms in this model with algorithms in the three 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 a perfect failure detector. Instead, we might rely on the eventually perfect failure detector ($\diamond\mathcal{P}$) of Module 2.5 or on the eventual leader detector (Ω) of Module 2.6.
- **fail-recovery**. We consider here the fail-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 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 fail-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 progressively 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 loads cannot be enforced, and where infrastructures such as real-time are not strictly required.

Exercises

Exercise 2.1 *Explain when (a) a fail-recovery model, and (b) an asynchronous model 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 implemenz a perfect failure detector if we cannot bound the number of omission 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. *eventually, 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.*

Corrections

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. □

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 proved 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 1987, 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 (**?**; Schneider 1987).
- 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 1996 (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).

3. Reliable Broadcast

This chapter covers the specifications of a family of agreement abstractions: *broadcast communication* abstractions. These are used to disseminate information among a set of processes. Roughly speaking, these abstractions capture a weak form of coordination coordination among processes, as processes must agree on the set of messages they deliver. We study here different variants of such abstractions. These differ according to the level of reliability they guarantee. 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 agreement even if the senders crash while broadcasting their messages.

We will consider six related abstractions: Best-Effort Broadcast, Regular Reliable Broadcast, Uniform Reliable Broadcast, Logged Best-Effort Broadcast, Logged Uniform Broadcast and Probabilistic Broadcast. For each of these abstractions, we will provide one or more algorithms implementing it, in order to cover the different models addressed in this book (fail-stop, fail-silent, fail-recovery and randomized).

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 would 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 not 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.

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*.

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:* 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.

3.2.1 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 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-Stop/ Fail-Silent Algorithm: Basic Multicast

We first provide an algorithm that implements best effort multicast using perfect links. This algorithm works both for fail-stop and fail-silent assumptions. 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 and illustrated by Figure 3.1. As long as the sender of the message does not crash, the properties of perfect links ensure that all correct processes will deliver the message.

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.

Algorithm 3.1 Basic Multicast.

Implements:
BestEffortBroadcast (beb).

Uses:
PerfectPointToPointLinks (pp2p).

upon event ⟨ *bebBroadcast*, m ⟩ **do**
forall $p_i \in \Pi$ **do** // Π is the set of all system processes
trigger ⟨ *pp2pSend*, p_i, m ⟩;

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

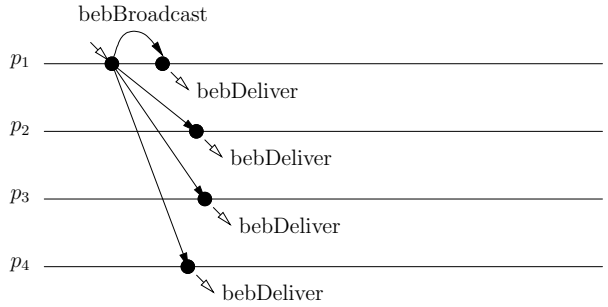


Figure 3.1. Sample execution of Basic Multicast algorithm.

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 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*.

Module:

Name: (regular)ReliableBroadcast (rb).

Events:

Request: $\langle$ *rbBroadcast*, m $\rangle$: Used to broadcast message m.

Indication: $\langle$ *rbDeliver*, src, m $\rangle$: 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 reliable broadcast.

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. This is depicted in Algorithm 3.2.

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") 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 *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

Algorithm 3.2 Lazy reliable broadcast.

Implements:
ReliableBroadcast (rb).

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

upon event $\langle$ *Init* $\rangle$ **do**
delivered := $\emptyset$;
correct := Π;
$\forall_{p_i \in \Pi}$: from$[p_i]$:= $\emptyset$;

upon event $\langle$ *rbBroadcast*, m $\rangle$ **do**
trigger $\langle$ *bebBroadcast*, [DATA, self, m] $\rangle$;

upon event $\langle$ *bebDeliver*, p_i, [DATA, s_m, m] $\rangle$ **do**
if $m \notin$ delivered **then**
delivered := delivered $\cup$ $\{m\}$
trigger $\langle$ *rbDeliver*, s_m, m $\rangle$;
from$[p_i]$:= from$[p_i] \cup \{[s_m, m]\}$
if $p_i \notin$ *correct* **then**
trigger $\langle$ *bebBroadcast*, [DATA, s_m, m] $\rangle$;

upon event $\langle$ *crash*, p_i $\rangle$ **do**
correct := correct $\setminus$ $\{p_i\}$
forall $[s_m, m] \in from[p_i]$: **do**
trigger $\langle$ *bebBroadcast*, [DATA, s_m, m] $\rangle$;

from the *validity* property of the underlying best effort broadcast 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, at 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.

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] ⟩;

In fact, we can circumvent the need for a failure detector (*completeness*) property as well by adopting a *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.

Algorithm 3.3 is in this sense eager but asynchronous: it makes use only of the best-effort primitive described in Section 3.2. In Figure 3.2a we illustrate how the algorithm ensures *agreement* event 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.

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.

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 could have communicated with the external world after having delivered the message. The processes that remain alive in the system should also be aware of that message having been delivered.

Figure 3.2b shows why our reliable broadcast algorithm does not ensure uniformity. Both process 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 it 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

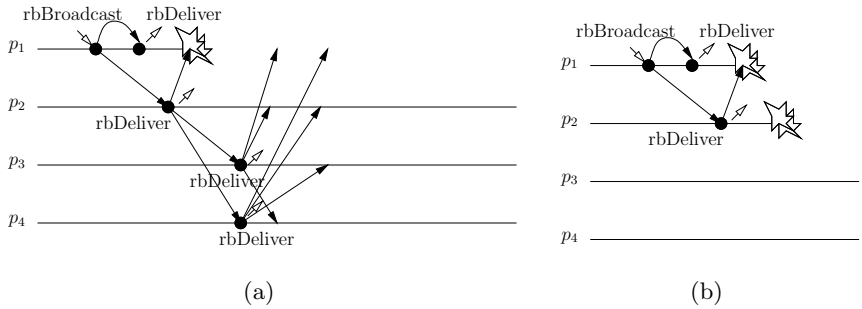


Figure 3.2. Sample executions of eager reliable broadcast.

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 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* 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.

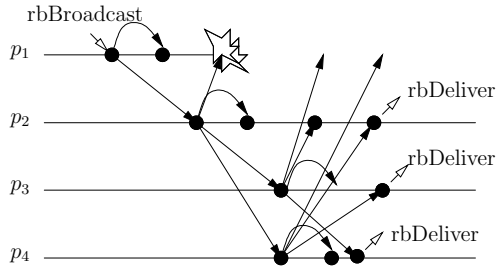


Figure 3.3. Sample execution of uniform reliable broadcast.

Correctness. As before, except for uniform agreement, all properties are trivially derived from the properties of the best-effort broadcast. *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 use of a perfect failure detector.

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 $\subset$ ack$_m$) $\wedge$ ($m \notin$ delivered);

upon event $\langle$ *Init* $\rangle$ **do**
delivered := forward := $\emptyset$;
correct := Π;
ack$_m$:= $\emptyset, \forall_m$;

upon event $\langle$ *urbBroadcast*, m $\rangle$ **do**
forward := forward $\cup$ $\{m\}$
trigger $\langle$ *bebBroadcast*, [DATA, self, m] $\rangle$;

upon event $\langle$ *bebDeliver*, p_i, [DATA, s_m, m] $\rangle$ **do**
ack$_m$:= ack$_m$ $\cup$ $\{p_i\}$
if $m \notin$ forward **do**
forward := forward $\cup$ $\{m\}$;
trigger $\langle$ *bebBroadcast*, [DATA, s_m, m] $\rangle$;

upon event $\langle$ *crash*, p_i $\rangle$ **do**
correct := correct $\backslash\{p_i\}$;

upon (canDeliver(m)) **do**
delivered := delivered $\cup$ $\{m\}$;
trigger $\langle$ *urbDeliver*, s_m, m $\rangle$;

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 broadcast requires one more step to deliver the messages 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 pro-

Algorithm 3.5 Majority ack uniform reliable broadcast.

Implements:
UniformReliableBroadcast (urb).

Uses:
BestEffortBroadcast (beb).

function canDeliver(m) **returns** boolean **is**
return ($|\text{ack}_m| > \text{N}/2) \wedge (m \notin$ delivered);
// Except for the function above, same as Algorithm 3.4.

cesses. In the example above of Figure 3.2, this 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 *agreement* and *validity* properties, we first argue that: if a correct process bebDelivers any message m, then p_i urbDelivers m. Indeed, if p_i is correct, and given that it 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 process p_i urbBroadcasts a message m, then p_i bebBroadcasts and hence bebDelivers m: by the argument above, it eventually urbDelivers m. Consider now *agreement* and let p_j be some process that urbDelivers m. To do so, p_j must have bebDelivered m from a majority of correct 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 Logged Best-Effort Broadcast

We now present an abstraction that captures the notion of reliable broadcast in the settings where processes can crash and recover. We present the specification and an algorithm to implement it.

Specification. We have called this abstraction Logged Best-Effort Multicast, to emphasize that fact that it relies on the fact that "delivery" of messages is performed by logging messages in a local log. The specification is

Module:

Name: Logged Best Effort Broadcast (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 logged by p_j.

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

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

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

given in Module 3.4. The key difference to the Best-Effort abstraction defined for the crash no-recovery setting is in the interface between modules. Instead of simply triggering an event to "deliver" a message, this abstraction relies of storing the message on a local log, which can later be read by the layer above (that layer is notified about changes in the log by delivery events). Note that the validity, no duplication and no creation properties are redefined in term of log operations.

Fail-Recovery Algorithm: Basic Multicast with Log. We now present an algorithm that implements logged best-effort broadcast. Algorithm 3.7 is called basic multicast with log and has many similarities, in its structure, with Algorithm 3.1. The main differences are as follows. 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 and the upper layer is notified that the log has changed. If the process crashes and later recovers, the upper layer is notified (as it may have missed some notification triggered just before the crash).

Correctness. The properties are derived from the properties of stubborn links. *No creation* is 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. *No duplication* is derived from the fact that a process logs all messages that it delivers and that this log is checked before accepting a new message.

Performance. The algorithm requires a single communication step and exchanges at least N messages (note that stubborn channels may retransmit the same message several times). Additionally, the algorithms requires a log operation for each delivered message.

Algorithm 3.6 Basic Multicast with Log.

Implements:
Logged Best Effort Broadcast (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** // Π is the set of all system processes
trigger ⟨ *sp2pSend*, p_i, m ⟩;

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

3.6 Logged Uniform Broadcast

In a similar manner to the crash no-recovery case, it is possible to define both regular and uniform variants of reliable broadcast for the fail-recovery setting. We now describe the uniform variant.

3.6.1 Specification

We define in Module 3.5 a *logged* variant of the uniform reliable broadcast for the fail-recovery model. In this variant, if a process logs a message (either correct or not), all correct processes eventually log that message. Note that, naturally, the interface is similar to that of logged reliable broadcast.

3.6.2 Fail-Recovery Algorithm: Uniform Multicast with Log

The Algorithm 3.7 implements logged uniform broadcast. The algorithm uses three variables: *todeliver*, *delivered*, and ack_m . The *todeliver* set is used to collect all messages that have been broadcast. The *delivered* set collects all messages that have been delivered. These two sets are maintained in stable storage and the last one is exposed to the upper layer: in fact, in this case to "deliver" a message consists in logging a message in the *delivered* set.

Module:

Name: Logged Uniform Reliable Broadcast (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_i and p_j are correct, then every message broadcast by p_i is eventually logged by p_j.

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

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

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

Module 3.5 Interface and properties of logged uniform reliable broadcast.

Finally ack_m sets collect acknowledgements fro message m (logically, there is a separate set for each message): a process only acknowledges the reception of a message after logging the message in stable storage. This ensures that the message will be preserved across crashes and recoveries. The *ack* set is not logged, it can be reconstructed upon recovery.

The algorithm exchanges two types of messages: data messages and acknowledgements. The logged best-effort broadcast of Section 3.5 is used to disseminate both types of messages. When a message is received from the first time it is logged in the *todeliver* set. Messages in this set are forwarded to all other processes to ensure delivery in the case the sender fails (the task of forwarding the messages is re-initiated upon recovery). Message are only appended to the *delivered* log when they have been acknowledged by a majority of correct processes.

3.7 Probabilistic Broadcast

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

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.

Algorithm 3.7 Uniform Multicast with Log.

Implements:
Logged Uniform Reliable Broadcast (log-urb).

Uses:
StubbornPointToPointLink (sp2p).
Logged Best-Effort Broadcast (log-beb).

upon event ⟨ *Init* ⟩ **do**
ack$_m$:= ∅, ∀m;
todeliver := ∅; delivered := ∅;
store (todeliver, delivered);
upon event ⟨ *Recovery* ⟩ **do**
ack$_m$:= ∅, ∀m;
retrieve (todeliver, delivered);
trigger ⟨ *log-urbDeliver*, delivered ⟩;
forall $m \in$ todeliver **do**
trigger ⟨ *log-bebBroadcast*, [DATA, m] ⟩;

upon event ⟨ *log-urbBroadcast*, m ⟩ **do**
todeliver := todeliver ∪ {m};
trigger ⟨ *log-bebBroadcast*, [DATA, m] ⟩;

upon event ⟨ *log-bebDeliver*, delset ⟩ **do**
forall packet ∈ delset **do**
// packet = [DATA, m] ∨ packet = [ACK, j, m]
if $m \notin$ todeliver **then**
todeliver := todeliver ∪ {m};
trigger ⟨ *log-bebBroadcast*, [DATA, m] ⟩;
if [ACK, self, m] ∉ ack$_m$ **do**
ack$_m$:= ack$_m$ ∪ { [ACK, self, m] };
trigger ⟨ *log-bebBroadcast*, [ACK, self, m] ⟩;
if packet = [ACK, j, m] ∧ packet ∉ ack$_m$ **do**
ack$_m$:= ack$_m$ ∪ { [ACK, j, m] };
if | ack$_m$ | > $N/2$ **then**
delivered := delivered ∪ {m};
store (todeliver, delivered);
trigger ⟨ *log-urbDeliver*, delivered ⟩;

3.7.1 Limitation 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 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

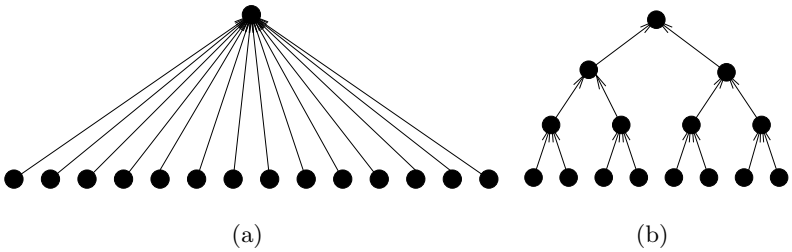


Figure 3.4. Ack implosion and ack tree.

acknowledgments becomes overwhelmed by that task. This phenomena is known as the *ack implosion* problem (see Fig 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 Fig 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.

3.7.2 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.

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: instead, it corresponds to a probabilistic variant.

3.7.3 Specification

Probabilistic broadcast is characterized by the properties PB1-3 depicted in Module 3.6.

Module:

Name: Probabilistic Broadcast (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 3.6 Interface and properties of probabilistic broadcast.

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.

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.

3.7.4 Algorithm: Eager Probabilistic Broadcast

Algorithm 3.8 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.

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

Algorithm 3.8 Eager Probabilistic Broadcast.

Implements:
ProbabilisticBroadcast (pb).

Uses:
unreliablePointToPointLinks (up2p).

upon event ⟨ *Init* ⟩ **do**
delivered := ∅;

function pick-targets (fanout) **returns** set of processes **do**
targets := ∅;
while | targets | < fanout **do**
candidate := random (Π);
if candidate ∉ targets ∧ candidate ≠ self **then**
targets := targets ∪ { candidate };
return targets;

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

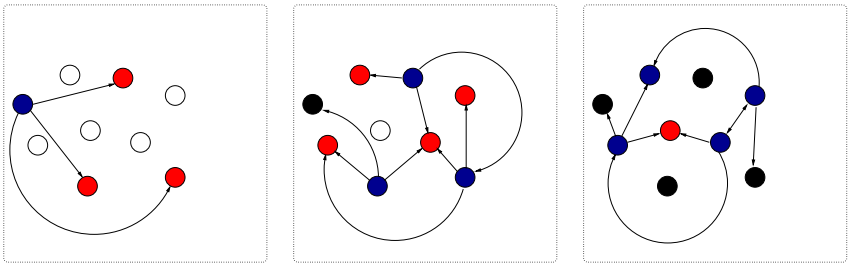
upon event ⟨ *pbBroadcast*, m ⟩ **do**
gossip ([GOSSIP, s_m, m, maxrounds−1]);

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

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 configuarion 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 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, the fanout is in the order of $\log N$, where N is the number of nodes in the system. Naturally, the exact value of the fanout and maximum number of rounds to achieve a given probability of success depends



(a) (b) (c)

Figure 3.5. Gossip Dissemination.

not only on the system size but also on the probability of link and process failures.

3.7.5 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 multicast primitive to disseminate the messages first, and then use a probabilistic approach just as a backup to recover from message omissions.

A simplified version of an algorithm based on this idea is given in Algorithm 3.9. 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 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 the sequence numbers of messages. A process p detects that it has missed a message from a process q when p receives a message from q if 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:

Algorithm 3.9 Lazy Probabilistic Broadcast.

Implements:
ProbabilisticBroadcast (pb).

Uses:
BestEffortBroadcast (beb). unreliablePointToPointLinks (up2p).

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

procedure deliver-pending (s) **do**
while $\exists_x$: [DATA, s, x, sn_x] *in* pending $\wedge$ sn_x = delivered[s]+1 **do**
delivered[s] := delivered[s]+1; pending := pending \ { [DATA, s, x,sn_x]};
trigger ⟨ *pbDeliver*, s, x ⟩;

// Procedure gossip same as in Algorithm 3.8

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

upon event ⟨ *bebDeliver*, 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 ∈ [$s_m - 1$, 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 ⟨ *upp2pSend*, 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] };

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 pure earlier probabilistic broadcast algorithm described above. On the other hand, it does require the availability

of some unreliable broadcast primitive and this primitive 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) to 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.

Exercises

Exercise 3.1 (*) *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 3.2 (**) *Modify the lazy reliable broadcast algorithm (Algorithm 3.2) to reduce the number of messages sent in case of failures.*

Exercise 3.3 (**) *All reliable broadcast (deterministic and fail-stop) algorithms we presented 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. $forward$ *in the uniform reliable broadcast algorithm*

Exercise 3.4 (*) *What do we gain if we replace bebBroadcast with rbBroadcast in our uniform reliable broadcast algorithm?*

Exercise 3.5 (*) *Consider our reliable broadcast and uniform broadcast algorithms that use a perfect failure detector. What happens if each of the following properties of the failure detector are violated:*

1. *accuracy*
2. *completeness*

Exercise 3.6 (**) *Our uniform reliable broadcast algorithm using a perfect failure detector 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 3.7 (**) *Can we devise a uniform reliable broadcast with an eventually perfect failure detector but without the assumption of a correct majority of processes?*

Exercise 3.8 (**) *The specification of reliable broadcast in a fail-recovery model given in Module* **??** *does only restrict the behavior of processes that do never crash, as far as* validity *is concerned.*

How can we implement a reliable broadcast abstraction ensuring the following stronger validity *property?*

- *If a correct process broadcasts a message m, then it eventually delivers m.*

Algorithm 3.10 Simple optimization of lazy reliable broadcast.

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

Exercise 3.9 (**) *Consider Algorithm* **??** *implementing a reliable broadcast in a fail-recovery model. Can we still rely only on an underlying stubborn-broadcast abstraction, yet optimize the algorithm, if the aim is to ensure the following weaker agreement property than the one of Module* **??**.

- *If a process p_i delivers a message m and p_i does not crash, then any process p_j that does not crash delivers m.*

Exercise 3.10 (**) *Our probabilistic broadcast algorithm considers that the connectivity is the same among every pair of processes. In practice, it may happen that some nodes 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.*

Exercise 3.11 (*) *Could the notion of "epoch" be removed from our flow-control algorithm (Algorithm* **??***)?*

Corrections

Solution 3.1 The answer is yes. The process anyway rbDelivers the messages as soon as it bebDelivers it. This does not add any guarantee with respect to rbDelivering the message before bebBroadcasting it. The event that we would need to change to Algorithm 3.2 would simply be the following.
□

Solution 3.2 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 are 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 coordinates the broadcast to every other process. If the leader is correct, everyone eventually bebDelivers and rbDelivers every message. Otherwise, we revert to the previous algorithm, and every process is responsible for bebBroadcasting the messages that it bebDelivers. □

Solution 3.3 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 safety property.

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

Solution 3.4 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 3.5 If the accuracy, i.e. the safety property, of the failure detector is violated, the safety property(ies) of the problem considered might be violated. In the case of (uniform) reliable broadcast, the agreement property can be violated. 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.

If the completeness, i.e. the liveness property of the failure detector, is violated, the liveness property(ies) of the problem considered might be violated. In the case of (uniform) reliable broadcast, 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 would be violated. □

Solution 3.6 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. An lazy scheme would be of no benefit here. □

Solution 3.7 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 have no way of knowing about that message m and eventually urbDelivering it: agreement is violated. □

Solution 3.8 Clearly, this property can only be achieved if the act of broadcasting a message is defined as the storing of that message into some stable storage variable. The property can then be achieved by a slight modification to Algorithm **??**: upon recovery, any process delivers the messages it has broadcast but not delivered yet. □

Solution 3.9 No. To ensure the following property:

- If a process p_i delivers a message m and p_i does not crash, then any process p_j that does not crash delivers m.

Without a perfect failure detector, a process has no choice then to forward any message it delivers. Given than the forwarding can only be achieved with the stubborn-broadcast primitive, the algorithm cannot be optimized any further.

□

Solution 3.10 One approach consists in assigning weights to link between processes. Links 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. □

Solution 3.11 No. Without the notion of epoch, *minb* will always decrease, even if more resources would be available in the system. The introduction of the epoch ensures that new up-to-date values of *minb* are not mixed will outdated values being gossiped in the system. □

Historical Notes

- The requirements for a reliable broadcast communication abstraction seem to have originated from the domain of aircraft control and the Sift system (Wensley 1978). Algorithms ensuring causal delivery of messages came out of the seminal paper of Lamport (Lamport 1978).
- Later on, several distributed computing libraries offered communication primitives with reliable or causal order 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).
- Algorithms for reliable broadcast message delivery were presented in a very comprehensive way in (Hadzilacos and Toueg 1994). The problem of the uniformity of a broadcast was discussed in (Hadzilacos 1984) and then (Neiger and Toueg 1993).
- The idea of applying epidemic dissemination to implement probabilistically reliable broadcast algorithms was explored in (Golding and Long 1992; Birman, Hayden, Ozkasap, Xiao, Budiu, and Minsky 1999; Kermarrec, Massoulie, and Ganesh 2000; Eugster, Handurukande, Guerraoui, Kermarrec, and Kouznetsov 2001; Kouznetsov, Guerraoui, Handurukande, and Kermarrec 2001; Xiao, Birman, and van Renesse 2002).
- The exploitation of topology features in probabilistic algorithms was proposed in (Lin and Marzullo 1999) through an algorithm that assigns weights to link between processes. A similar idea, but using a hierarchy instead of weight was proposed in (Gupta, Kermarrec, and Ganesh 2002) to reduce the traffic across domain boundaries.
- The first probabilistic broadcast algorithm that did not depend of any global membership was given in (Eugster, Handurukande, Guerraoui, Kermarrec, and Kouznetsov 2001) and the notion of message ages was introduced in (Kouznetsov, Guerraoui, Handurukande, and Kermarrec 2001) for purging messages and ensuring the scalability of process buffers.
- The idea of flow control in probabilistic broadcast was developed in (Rodrigues, Handurukande, Pereira, Guerraoui, and Kermarrec 2002). The same paper also introduced a decentralized techniques to control message epochs.