Rachid Guerraoui, Luís Rodrigues

The Unbearable Lightness of Agreement

Distributed Computing for the Undergraduates
(Preliminary Draft)

June 9, 2002

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

To whom it might concern.

Preface

God does not often clap his hands. When He does, everybody should dance.
(African proverb)

This manuscript aims at offering a comprehensive and systematic description of agreement abstractions in distributed computing. The manuscript can be a valuable tool for those that want to have a better knowledge of some fundamentals in distributed computing as well as to those that need to apply this knowledge when building large and complex distributed applications. Although the manuscript is written in a self-contained manner, it is assumed that the reader has previously been exposed to elementary notions of programming and operating systems.

Agreement

The problem of distributed computing consists for a set of processes, each representing a computer, to reach some form of sophisticated agreement, based on some more primitive form of initial agreement. Had no notion of agreement been required, one could simply build a distributed program by superposing independent centralized programs. The heart of distributed computing is indeed agreement.

The processes might for instance have initially agreed on who they are (e.g., IP addresses) and some format for representing messages (e.g., IIOP), and might need to agree on some reliable way of exchanging messages (e.g., TCP). They might then go a step further and try to agree on some common final value based on some initial values (e.g., consensus or atomic commitment). They might similarly need to agree on some total order to deliver messages, or some global notion of logical time to compensate the absence of physical time.

The distributed computing problem is complicated because some of the processes might fail while others might keep operating. It can however be simplified if the programmer is provided with abstractions that factor out the difficulty of various forms of agreement that can be reached, despite the failure of some of these processes. By encapsulating many tricky issues, agreement 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.

Organization

The manuscript follows an incremental approach and was primarily built as a textbook for teaching. It introduces the fundamental agreement problems in an intuitive manner and builds sophisticated agreement abstractions on top of more primitive agreement ones, assuming a simple model of a distributed system first, later revisiting the same basic abstractions in more challenging models.

In Chapter 1 we motivate the need for agreement abstractions and we present the programming model used in the book to build agreement algorithms. In Chapter 2 we present different forms of assumptions about the underlying distributed environment. Basically, we present the minimal form of initial agreement that the processes should be sharing. In Chapter 3 we introduce specific forms of agreement abstractions: those related to the reliable delivery of messages broadcast to a group of processes. In Chapter 4 we introduce the consensus abstraction. Chapter 5 covers the totally ordered delivery of messages broadcast to a group of processes. Chapter 6 considers failure-sensitive forms of agreement, namely, leader election, terminating reliable broadcast and non-blocking atomic commit.

Acknowledgments

We have been exploring the world of agreement for over a decade now. During that period, we were influenced by many researchers in the field of distributed computing. Their work is referenced throughout the manuscript. A special mention to Leslie Lamport and Nancy Lynch for having defined the fascinating field of distributed computing, and to the Cornell band, including Ken Birman, Robert van Renessee, Fred Schneider, and Sam Toueg, for their seminal work on various forms of broadcast abstractions.

We would like to express our gratitude to our undergraduate students from the Swiss Federal Institute of Technology in Lausanne (EPFL) and the University of Lisboa, for serving as cobayes of preliminary drafts of this manuscript. Partha Dutta, Corine Hari, Petr Kouznetsov and Bastian Pochon, PhD students at the Distributed Programming Laboratory (EPFL), suggested many improvements to the algorithms presented in the manuscript. Corine and Bastian were also instrumental in setting up the exercice sessions.

Rachid Guerraoui and Luís Rodrigues

Contents

1. Introduction

I could have been some one. So could any one. (The Pogues)

This chapter first motivates the need for agreement abstractions and gives a hint on the characteristics of these abstractions. It then advocates a modular strategy for the development of distributed programs by encapsulating agreement abstractions within Application Programming Interfaces (APIs). A concrete example API is given to illustrate the typical abstractions offered by an agreement toolkit. This should also be viewed as a simple example of the event-based model used throughout the manuscript to describe agreement algorithms.

1.1 Applications

An obvious target of agreement abstractions are applications that require coordination and dissemination of information among several participants. Examples of such applications are multi-user cooperative applications such as virtual-environments, distributed shared spaces, cooperative editors, replicated systems, and distributed databases. These applications indeed need agreement abstractions, but with a different range of requirements. Often, heterogeneous requirements can only be satisfied by the simultaneous use of different agreement abstractions. For instance, the dissemination of an audio stream can be performed using efficient best-effort broadcast primitives (a weak form of agreement), the acquisition of a lock for a shared object requires reliable and ordered communication (stronger forms of agreement), whereas the termination of a transaction requires an atomic commitment protocol (an even stronger form of agreement).

Event Dissemination Systems. These are a typical class of applications for which agreement might be useful. In such applications, 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. Typically, and for fairness issues, the subscribers need to agree here on the set of messages they deliver. In several publish-subscribe applications, producers and consumers interact indirectly, with the support of a group of intermediate cooperative brokers. In such cases, agreement is not only useful to support the dissemination of information but also for the coordination of the brokers.

Replicated Services. The need to support groups of coordinated processes does not always derive directly from the functional requirements of the application. In fact, it often appears as an artifact of the solution to satisfy some specific non-functional requirements such as *fault-tolerance* or *load-balancing*. One of the techniques to achieve fault-tolerance in distributed environments is the use of replication. Briefly, replication consists of executing several copies of the same component to ensure continuity of service in case a subset of these components crashes. No specific hardware is needed: fault-tolerance through replication is software-based. For replication to be effective, the different copies must be maintained in a consistent state. If replicas are deterministic, one of simplest manners to attain this goal is to ensure that all replicas receive the same set of requests in the same order. Typically, such guarantees are enforced by a reliable group communication primitive called total order broadcast: the processes need to agree here on the sequence of messages they deliver. Of course, algorithms that implement such a primitive are rather tricky, and providing the programmer with an abstraction that encapsulates these algorithms makes the programming of replicated components easier. After a failure, it is desirable to replace the failed replica by a new component. Again, this calls for systems with dynamic group membership and for additional auxiliary abstractions, such as a *state-transfer* mechanism that simplifies the task of bringing the new replica up-to-date.

Databases. Distributed and replicated databases are particular cases of applications where agreement abstractions may play an important role. Replication is typically used here to improve the read-access performance to data by placing it close to the processes where it is supposed to be queried. Agreement abstractions can be used here 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.

1.2 The End-to-end Argument

Agreement abstractions may be useful but are sometimes difficult to implement. In some cases, no simple algorithm is able to provide the desired abstraction and the algorithm that solves the problem can have a high complexity in terms of the number of steps and messages. Therefore, depending on the system model, the network characteristics, and the required quality of service, the overhead of an agreement primitive can range from the negligible to the almost impairing.

Faced with performance constraints, the application designer may be driven to mix the relevant agreement logic 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 agreement abstractions are 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 (Saltzer, Reed, and Clark 1984), that states that most complexity should be implemented at the higher levels of the communication stack. This is indeed true for any abstraction.

However, in the case of agreement, even if some performance gains can be achieved by collapsing the application and the underlying agreement layers, such approach an has many disadvantages. First, it is very error prone. Some of the algorithms that will be presented in this manuscript have a reasonable amount of complexity and exhibit subtle dependencies among their internal components. An apparently obvious "optimization" may break the algorithm correctness. Even if the designer achieves the amount of expertise required to master the difficult task of integrating these algorithms with 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 agreement problem. Instead, different solutions have been proposed and these solutions are not strictly superior to each other: each has 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.

Ultimately, one might indeed consider optimizing the performance of the final release of a distributed application and bypassing any intermediate abstraction layer. The understanding of the candidate algorithms to implement

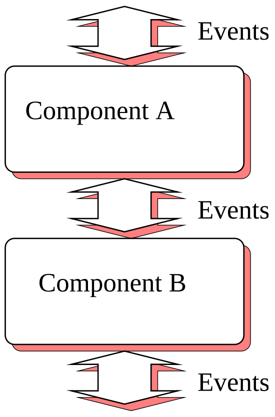

Figure 1.1. Composition model

these abstractions, and some of the associated impossibility results, is fundamental before they can be safely merged with the code of a distributed application. Furthermore, we strongly believe that, in many distributed applications, especially those that require many-to-many interaction, building preliminary prototypes of the distributed application using agreement abstractions can be very helpful.

1.3 Programming Abstractions

Basic agreement abstractions are typically encapsulated through application programming interfaces (API). We will informally discuss here a simple example API.

To describe our APIs and the algorithms implementing these APIs, we shall consider, throughout the manuscript, an *asynchronous event-based composition* model. Every process hosts a set of software modules, called *components*. Each component is identified by a name, characterized by a set of properties, and provides an interface in the form of the events that it accepts and produces.

The properties of a component can be separated in two different classes: *safety* and *liveness*. Safety properties state that the system cannot do something wrong, e.g., processes should not decide on different values or deliver messages in different orders. Of course, safety properties are not enough, since a good way of preventing bad things from happening consists in simply doing nothing (in some countries, some public services seem to understand this rule quite well). Therefore it is necessary to add some liveness properties to ensure that eventually something good happens, e.g., processes need to decide some value or deliver some messages. 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. 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 more safety properties.

Each component is typically organized as a software stack, within every process: every 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 agreement abstractions are in the middle. Components within the same stack communicate through the exchange of *events*, as illustrated in Fig. 1.1. A given agreement abstraction is typically materialized by a set of components, each running at a given 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 ⟨ *Event*, att1, att2, ... ⟩. Typically, each event is treated by a dedicated handler. The processing of an event may result in new events being created and triggered on the same or on other components. Events from the same component are triggered on a component in the same order they were created. Note that this FIFO (*first-in-first-out*) order is local and different from the distributed one that might need to be preserved among processes (i.e., components on different processes). The code of each component looks like this:

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

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

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

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 the broadcast in the example above 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 agreement, e.g., the processes need to agree on whether or not to deliver a message broadcast by some process. The execution is initiated by the reception of a *request* event from the layer above. To satisfy a request, the layer will send one or more messages to its remote peers using the services of the layer below. Messages sent by its peers are also *received* using the services of the underlying layer. When a message is received, it may have to be stored temporarily until the agreement property is satisfied, before being *delivered* to the layer above. This dataflow is illustrated in Fig. 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.

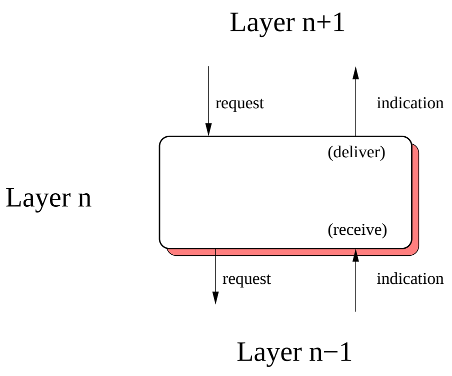

Figure 1.2. Layering

The lowest layer in the software stack interfaces the *links* that provide connectivity among processes. At the highest level of abstraction, 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. Concrete examples, such as the ones illustrated in Fig. 1.3, include the use of a broadcast medium (such as an Ethernet), a ring, or a mesh of links interconnected by bridges and routers (the Internet). Many implementation refine the abstract network model to make use of the properties of the underlying topology.

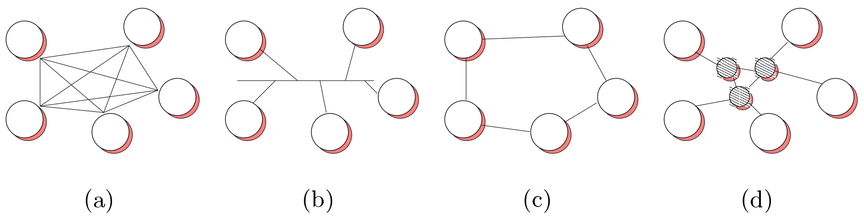

Figure 1.3. The link abstraction and different instances.

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. Using this interface, a simple application that consists in keeping track of the set of processes in a group would look like the code depicted in Fig. 1.4.

```
upon event ⟨ Init ⟩ do
    trigger ⟨ JoinRequest, myGroup, myId ⟩;

upon event ⟨ GroupViewIndication, groupView ⟩ do
    GroupView nv = e.groupView;
    print (”We are now” + nv.size() + ”members”);
    trigger ⟨ MulticastRequest, myGroup, QoS, ”Hello world” ⟩;

upon event ⟨ MulticastIndication, group, from, QoS, data ⟩ do
    print (e.data);
```

Figure 1.4. Simple example interface.

2. Models

It does not matter what I believe, it matters what I am able to prove.
(Inspector Derrick)

Reasoning about distributed algorithms in general, and in particular about algorithms that implement agreement abstractions, first goes through defining a clear model of the distributed system environment where these algorithms are supposed to operate. In a distributed system, a model can be viewed as a set of assumptions about the allowable behavior of the processes and their communication links in the distributed system.

2.1 Processes and Links

2.1.1 Local and Global Events

We consider here N processes in the system, denoted by $p_1, p_2, ..p_N$ (sometimes we also denote the processes by p, q, r). These processes communicate by exchanging messages. Unless it *fails*, every 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; if it fails, all its components are supposed to have failed as well: otherwise, all the components are supposed to correctly execute the part of the algorithm assigned to them. Models differ according to the nature of the failures that are considered.

When executing its algorithm, a process (that has not failed) alternates between executing some local computation (local event) and exchanging messages (global event): it is important to notice here that the communication between local components of the very same process is viewed as a local computation and not as a communication. Again, the process is the unit of communication, just like it is the unit of failures.

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* received from process *src*.

Properties:

PL1: *Reliable delivery:* If p_i and p_j are correct, then every message sent by p_i to p_j is eventually delivered by p_j.

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

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

Module 2.1 Interface and properties of perfect point-to-point links.

2.1.2 Perfect Links

It is convenient when devising a distributed algorithm to assume that links are perfect and never fail. The perfect link abstraction is captured by the "Perfect Point To Point Link" module, i.e., Mod.2.1. The interface of this module consists of two events: a request event (to send messages) and an indication event (used to deliver the received messages). Perfect links are characterized by the properties PL1-PL3.

It may seem awkward to assume that links are perfect when it is known that real links may crash, lose and duplicate messages. However, this assumption only encapsulates the fact that these problems can be addressed by some low level protocol. As long as the network remains connected, and processes do not commit an unbouded number of omission failures, link crashes may be masked by routing algorithms. The loss of messages can be masked through re-transmission. This functionality is often found in standard transport level protocols such at 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 agreement algorithms. On the other hand, when developping actual distributed applications, these details become relevant. For instance, it may happen that some agreement algorithm requires the use of sequence numbers and message retransmissions, regardless of the properties of the underlying links. In this case, in order to avoid the redundant use of similar recovery mechanims at different layers, it may be more efficient to rely just on weaker links.

2.1.3 Algorithms and Steps

It is convenient to view the execution of an algorithm by a process as a sequence of steps: steps are executed according to ticks of the internal clocks of processes: one step per clock tick. A step consists in receiving a message from another process, executing a local computation and sending a message to some 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 the *nil* message.

When we present an algorithm we analyse its cost using two metrics: (1) the number of messages required to terminate the algorithm and (2) the number of *communication steps*. In short, a communication step of the algorithm occurs when a process sends a message and another process has to wait for this message to arrive to complete a phase in the algorithm. The number of communication steps is associated with the latency an implementation exhibits, since the network latency is typically a limiting factor on the performance of distributed algorithms. Distributed system models also differ according to the restrictions imposed on the speed at which local steps are performed and messages are exchanged.

In short, two aspects are of utmost importance when defining a model of a distributed environment: the assumptions about the possible failure modes of the processes and the assumptions about the behavior of the processes in the time domain. We discuss these two aspects below.

2.2 Failures

The very characteristic of a distributed environment is the possibility of partial failures: some of the processes might fail whereas others might be correct. In fact, we consider this characteristic as the definition of a distributed system and that is exactly how we differentiate it from a parallel system. It is usual to quote Leslie Lamport here: "a distributed system is one that stops your application because a machine you have never heard of has crashed". The challenge of researchers in distributed computing is precisely to devise algorithms that provide those processes that did not fail with enough coherent information so that they can operate correctly despite the failure of others.

2.2.1 Omissions

One typical kind of faults to consider is the *omission*. An omission fault occurs when a process does not send (resp receive) a message it is supposed to send (resp receive): typically, omission faults are due to buffer overflows or network congestions and 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 the external world.

2.2.2 Crashes

An interesting particular case of omissions is when a process executes its algoritm correctly, including the exchange of messages with other processes, 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 and never recovers. We talk here about a *crash fault* and a *crash stop* model.

2.2.3 Crashes and Recoveries

The *crash-recover* fault model considers that processes can indeed crash, and hence stop sending messages, but later recover. This can be viewed as a model with omissions with one exception however: a process migh suffer *amnesia* when it crashes and looses its internal state. We typically assume here that every process has, in addition to its regular volatile memory, a *stable storage*, which can be accessed through *store* and *retrieve* primitives. When a process recovers, it preserves all data that has been saved in the stable storage but looses all the remaining data that was preserved in volatile memory. Clearly, we obtain a model with omissions if we consider that every process stores every update to its state in stable storage. This is not very practical because access to stable storage is usually expensive. A crucial issue in devising algorithms for the crash-recover model is to minimize the access to stable storage.

2.2.4 Arbitrary Behaviors

Finally, the *arbitrary fault* model is the most general one. 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. This kind of fault model is also called the *malicious* or the *Byzantine* model (Lamport, Shostak, and Pease 1982).

Not surprisingly, arbitrary faults are the most expensive to tolerate, but this is the only acceptable model when an extremely high coverage is required or when there is the risk of some processes being controlled by malicious users that deliberately try to prevent correct system operation.

The failure model we shall mainly consider throughout the manuscript is the *crash* model. Hence, unless indicated otherwise, we exclude other kinds of failures. Furthermore, as discussed below, we shall assume sometimes that failures can be detected through some failure detector modules. Such modules can however only detect crashes that have occured in the past and cannot guess the future.

2.3 Timing Assumptions

When considering timing assumptions, two extreme approaches can be used to characterize the behavior of a distributed system in the time domain: *synchronous* and *asynchronous* models. We introduce these two models below and discuss their properties. We also point out the need for intermediate *partial synchronous* models and present an interesting property of such models.

2.3.1 Synchronous Model

Properties. Assuming a *synchronous* model 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 local step is always less than this bound. Remember that a local step groups (1) the reception of a message (possibly *nil*) sent by some other process, (2) a local computation, and (3) 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 time at which a message is sent and the time at which the message is received by the destination process is less than this bound.
3. *Synchronous clocks.* There is a known upper bound on the rate at which the local physical clock of each process drifts from a global real time clock (we make here the approximate assumption that such a global real time clock exists in our universe, i.e., at least to God). Remember here that every process has a local physical clock and every process executes a local step at every tick of its clock. The synchronous clock property means that the local clock differs from the global one up to a known upper bound.

Perfect Failure Detection. In synchronous environments, if we assume that processes can only fail by crashing, and processes that crash do not recover, failures 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). Hence, a simple method for detecting a crash is to rely on *timeouts*: using its own clock, the process can measure the worst case delay required to obtain a response and detect a crash in its absence (the crash detection will usually trigger a corrective procedure).

It is a good practice to encapsulate the way the failures are detected in a synchronous model through the use of a *perfect failure detector* module.

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 suspected by every correct process.

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

Module 2.2 Interface and properties of the perfect failured detector.

In short, this module outputs, at every process, the set of processes that are suspected to have crashed. A perfect failure detector can be described by the *accuracy* and *completeness* properties (Chandra and Toueg 1996) of Mod.2.2.

As we shall see in the next chapters, this module provides a very convenient abstraction upon which to build agreement abstractions. In particular, none of the agreement algorithms that we will present explicitely manipulates timing assumptions: these assumptions are encapsulated within a failure detector module.

Physical Time. In most distributed environments, a perfect failure detector can only be implemented if the system is synchronous. On the other hand, there are several characteristics of a synchronous system that are not captured by the perfect failure detection abstraction. For instance, in a synchronous system the following additional services can also be implemented:

- *Timed failure detection.* Every failure is detected within bounded time. Hence, the processes can implement an even stronger failure detection service such that, whenever a process detects the crash of some other process q, all processes that did not crash yet, detect the crash of q within a known bounded time.
- *Measure of transit delays.* It is possible to measure the delays in the 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 (Gray and Cheriton 1989), e.g., manipulating a specific file.
- *Worst case performance:* by establishing a bound on the number of faults and on the load of the system, it is possible to derive *worst case execution times* for a given algorithm. This allows a process to know when a message of his has been received by all correct processes. This can be achieved even if we assume that processes commit omission failures without crashing, as long as we bound the number of these failures.

- *Clock synchronization.* The synchronous model 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.[1]

2.3.2 Asynchronous Model

Assuming an *asynchronous* model comes down to not making any timing assumption. If a problem can be solved in the asynchronous model, the solution can also be applied to the synchronous model without risking any correctness violation when the synchronous assumptions do not hold.

In the asynchronous model, processes have no access to synchronized clocks. Therefore, the passage of time can only be measured based on the transmission and reception of messages, i.e., time is defined with respect to communication. Time measured this way is called *logical time.* In probably the most influential paper in the area of distributed computing, Leslie Lamport proposed the following rules to measure the passage of time in an asynchronous distributed environment (Lamport 1978):

- Each process p keeps an integer called *logical clock* l_p, initially 0.
- Any time a local event occurs at process p (i.e., p executes a local step), 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.
- When a process receives a message m with timestamp l_m process 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 ocurred before e_2 (Fig. 2.1 (a)).

[1] 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. Practical synchronous system delay are bounded but variable. This means that 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.

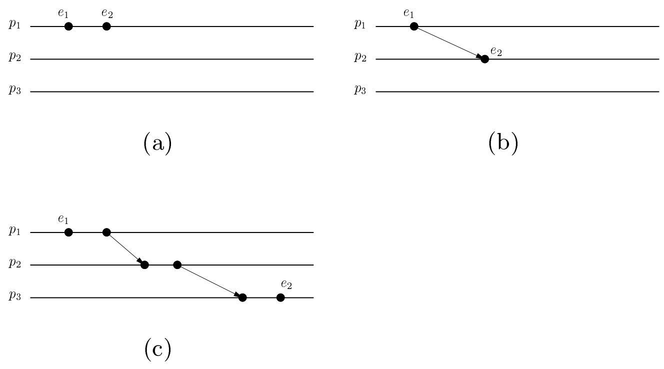

Figure 2.1. The *happened-before* relation.

- 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 (Fig. 2.1 (b)).
- there exists some event e' such that $e_1 \rightarrow e'$ and $e' \rightarrow e_2(Fig.\ 2.1(c))$.

It can be shown that if the events are timestamped with logical cloks, 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 synchrony assumption, and using only a logical notion of time, we can solve some (weak) agreement problems. Stronger agreement problems do however need some synchrony assumptions as we shall point out in the next section.

2.3.3 Partially Synchronous Model

Synchronous vs Asynchronous. A major limitation of the synchronous model is its *coverage*, i.e., the difficulty of building a system where the 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. In fact, there is an active area of research devoted to the study of techniques that allow to construct distributed systems with such characteristics (these systems are sometimes called *real-time systems*). Many issues are open, and, except for specific environments, it is not possible or even desirable to emulate a synchronous system model. The reasons why it is not possible are clear: in many systems, e.g., on the Internet, there are periods where messages can take days to arrive to their destination. The reason why it is not desirable is more subtle. One could consider very large values for the processing, communication, and clock bounds. This however would mean considering worst cases and the slowest processes and links. The system would hence perform at the speed of its slowest part.

What makes the asynchronous model very generic is its simplicity and its weakness. It is simple because there is no need to manipulate time in the algorithms: such manipulations are usually very error prone. It is weak because it does not make any timing assumption. Consequently, there is no need to make strong assumptions on the reliability of the communication either. In an asynchronous system, since bounded execution is not guaranteed, only *eventual* termination needs to be enforced: it is for instance often enough to ensure that if a message is retransmitted an infinite number of times it is eventually received by the destination process.

Because of its weakness, several important distributed computing problems cannot be solved in an asynchronous system model. One seminal paper in distributed computing (Fischer, Lynch, and Paterson 1985) proved that even a very simple form of agreement, namely *consensus*, is impossible to solve with a deterministic algorithm in an asynchronous environement even if only one process fails, and it can only do so by crashing. The consequence of this result is immediate for the impossibility of deriving dependable algorithms for many agreement abstractions, including group membership or totally ordered group communication. Additionally, most practical environments can exhibit some timely behaviour most of the time. This positive aspect, that makes real systems usable, is not captured by the asynchronous model. In fact, and as we have discussed, a very desirable feature of synchronous environments, is that they enable crash failures to be accurately detected using timeouts. What makes problems like consensus in asynchronous systems impossible, is precisely that failures cannot be reliably detected, since it is impossible to distinguish a crashed process from a process that is arbitrarily slow.

A significant amount of research has been devoted to the definition of intermediate models between the synchronous and the asynchronous model (C. Dwork and Stockmeyer 1988). The goal is to find whether one can identify reasonable timing assumptions that, on the one hand, are weak enough to be practical, and on the other hand, are strong enough to make interesting distributed computing problems solvable. There are currently two approaches in this quest.

1. The top-down approach consists in addressing a specific problem (e.g., consensus) and trying to define the minimum set of synchrony assumptions that needs to be added to an asynchronous model in order to solve the problem. This approach provides a better insight on the requirements imposed by the problem. On the other hand, it may happen that the additional properties may be extremely hard to obtain with acceptable coverage.
2. The bottom-up approach consists in looking to concrete distributed environments and identifying additional synchrony assumptions that can realistically be enforced using the inherent characteristics of such environments. Based on these assumptions, one identifies what classes of problems can be solved.

In the following section, we give an example of the bottom-up approach.

Eventual Timing Assumptions. Generally, distributed environments are most of the time completely synchronous. There are however periods where the synchronous assumptions do not hold, e.g., periods where the network is overloaded, or some process has a shortage of memory that slows it down. One way to capture the practical fact that most of the time systems are synchronous is to assume that the timing assumptions only hold eventually (without precising when exactly). That is, we assume that there is a time after which these assumptions hold, but this time is not known. In a way, instead of assuming a synchronous system, we assume a system that is eventually synchronous.

In fact, the timing assumption properties do not need to hold forever after a certain time, they only need to hold long enough for the algorithm of interest to terminate. In order not to bound the assumptions to a given algorithm, we simply assume that they hold forever after some time. Interestingly, in such a *partially synchronous model*, one can provide some information about failures that makes many problems (such as consensus) solvable. In fact, we can encapsulate the information about failures we can obtain in such models within an *eventually perfect failure detector* module.

Eventually Perfect Failure Detection. Basically, this module guarantees that there is a time after which crashes can be accurately detected. Again, this captures the intuition that, most of the time, timeouts can be adjusted such that so they can accuraterly detect crashes.

To implement an eventually perfect failure detector module, the idea is to use a timeout, and to suspect processes that have timed-out. Obviously, a suspicion might be wrong here. A process p might suspect a process q even if q has not crashed simply because the timeout chosen by p to suspect the crash of q, was too short. In this case, p's suspicion is false. When p receives a message from q, and it will if p and q are correct because we assume reliable channels, p revises its judgment and stops suspecting q. Process p also increases its timeout with respect to q. Clearly, if q now crashes, p will eventually permanently suspect it. If q does not crash, then there is a time after which p will stop suspecting q, i.e., the timeout of p for q will be large enough. 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 (PFD1-2),(Chandra and Toueg 1996) of Mod.2.3.

2.3.4 Indulgence

Algorithms which assume that processes can only fail by crashing and every process has accurate information about which process has crashed will be

Module:

Name: Eventually Perfect FailureDetector ($\diamond\mathcal{P}$).

Events:

Indication:

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

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

Properties:

PFD1: *Eventual strong completeness:* Eventually every process that crashes is permanently suspected by every correct process.

PFD2: *Eventual Strong accuracy:* Eventually, no correct process is suspected by any correct process.

Module 2.3 Interface and properties of the eventually perfect failured detector.

called *fail-stop* algorithms, following the terminology of (Schneider, Gries, and Schlichting 1984).

Algorithms that do not make any failure detection assumption at all, or those which rely only on eventually perfect failure detectors are called *indulgent* algorithms, following the terminology of (Guerraoui 2000). These algorithms are indulgent towards their failure detector: they forgive its mistakes. With these algorithms, no process can ever know if any other process has crashed or not. Without any failure detection assumption, indulgence is obvious. With an eventually perfect failure detector, processes know that they will eventually suspect crashed processes and that they will eventually stop suspecting correct processes. They do not know when their suspicions will become accurate. Hence, at any point in time, no process knows if any other process has crashed or not.

Indulgent algorithms tolerate asynchronous periods of the system where process relative speeds and communication delays are unbounded. They also inherently tolerate temporary partitions. We discuss some interesting properties of indulgent algorithms for various agreement abstractions.

Exercises

1. When can we say that a model where any process can crash and recover and a model where any process can commit omission failures are similar?

2. Which property of a perfect failure detector is a liveness property?

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

4. Consider any indulgent algorithm A that solves a problem M. Can A violate the safety property of M if the system turns out to be completely asynchronous?

5. Give a problem M and an indulgent algorithm A solving M, such that the liveness of M is violated if the system turns out to be asynchronous.

6. Can we build a perfect failure detector (to detect crashes) if we cannot bound the number of omission faults? What if processes that can commit omission failures commit a limited and known number of such failures and then crash?

7. Using a perfect failure detector, 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?

Corrections

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 thay have crashed and recovered, then the two models are similar.

2. The strong completeness is a liveness property. Strong accuracy is a safety property.

3. 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.

4. No. Assume by contradiction that A violates the safety property of M if the system turns out to be completely asynchronous. 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 (the system is not asynchronous).

5. The problem is simply that of building an eventually perfect failure detector.

6. No. There is no way to accurately detect the crash of a process that commits an unbounded number of omission failures. The only way to see if a process is up is through the messages it sends. If the process commits a bounded number of omission failures and this bound is known in a synchronous system, we can use it to calibrate the time-out of the processes to accurately detect failures. If the time 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.

7. 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.

3. Reliable Broadcast

The man thinks. The dog thinks. The fish does not think. Because the fish knows. (G. Bregovic)

This chapter covers the specifications of a typical form of agreement abstractions: broadcast. Roughly speaking, these abstractions ensure that the processes agree on the set of messages they deliver. We study here four variants of such abstractions: *best-effort broadcast*, *regular reliable broadcast*, *uniform reliable broadcast* and *causal broadcast*. We do so in a distributed system model where communication channels are reliable but processes can fail by crashing.

3.1 Intuition

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-tier 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 locations, 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 are very limited 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 (we shall consider other forms of broadcast abstractions 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.*

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

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.

with enforcing the reliability of received messages. Nevertheless, no guarantees are offered in case the sender fails. More precisely, best-effort broadcast is characterized by the properties BEB1-3 depicted in Mod. 3.1. Note that broadcast messages are implicitly addressed to all processes.

3.2.2 Algorithm

To provide best effort broadcast on top of perfect links is quite simple. It suffices to send a copy of the message to every process in the system, as illustrated in Fig. 3.1. The code of the algorithm is given in Alg. 3.1. As long as the sender of the message is correct, the properties of perfect links ensure that all correct processes will deliver the message. Note however that, if the sender crashes during a transmission, it may send a message to some processes but not to others. 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.

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 $(N-1)$ messages (we do not count loopback messages).

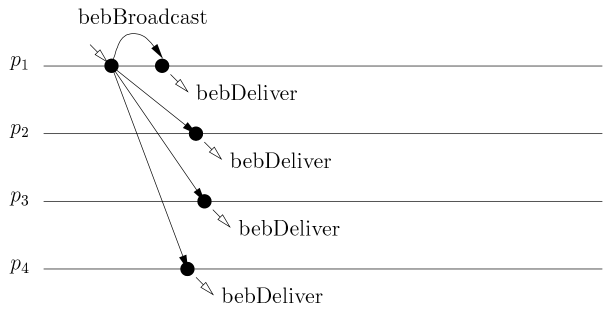

Figure 3.1. Sample execution of best-effort broadcast using perfect links.

Algorithm 3.1 Best-effort broadcast using perfect links.

```
Implements:
    BestEffortBroadcast (beb).

Uses:
    perfectPointToPointLinks (pp2p).

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

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

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. 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 Specifications

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 Mod. 3.2.

3.3.2 An Optimistic Reliable Broadcast Algorithm

To implement regular reliable broadcast, we make use of the best-effort abstraction described in the previous section as well as the perfect failure de-

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.

tector module introduced earlier in the manuscript. We give an algorithm in Alg. 3.2.

To broadcast (rbBroadcast) a message, a process uses the best-effort broadcast primitive to disseminate the message to all, i.e., it bebBroadcasts the message. 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. Our algorithm is *optimistic* in the sense that it favors the case where the sender does not crash.

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 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 $N-1$ messages. Otherwise, at the worst case, if the processes crash in sequence, $N-1$ steps and $(N-1)^2$ messages are required to terminate the algorithm.

Algorithm 3.2 Optimistic reliable broadcast.

```
Implements:
    ReliableBroadcast (rb).

Uses:
    BestEffortBroadcast (beb).
    PerfectFailureDetector (p).

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

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

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

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

3.3.3 A Pessimistic Reliable Broadcast Algorithm

In the previous algorithm, we basically make use of the completeness property of the failure detector to ensure agreement. If the failure detector does not ensure accuracy, then the processes might be relaying messages when it is not really necessary.

In fact, we can circumvent the need for a completeness property as well by adopting a *pessimistic* 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.

Our pessimistic algorithm, given in Alg. 3.3, makes use only of the best-effort primitive described in the previous section.

In Fig. 3.2a we illustrate how the algorithm ensures agreement event if the sender crashes: process p_1 crashes and its message is not bebDelivered

Algorithm 3.3 Pessimistic reliable broadcast.

```
Implements:
    ReliableBroadcast (rb).

Uses:
    BestEffortBroadcast (beb).

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

upon event ⟨ rbBroadcast, $m$ ⟩ do
    delivered := delivered $\cup$ {$m$}
    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 $\cup$ {$m$}
        trigger ⟨ rbDeliver, $s_m, m$ ⟩;
        trigger ⟨ bebBroadcast, [DATA, $s_m$, $m$] ⟩;
```

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 optimistic one), p_2 will be relaying the message only after it has detected the crash of p_1.

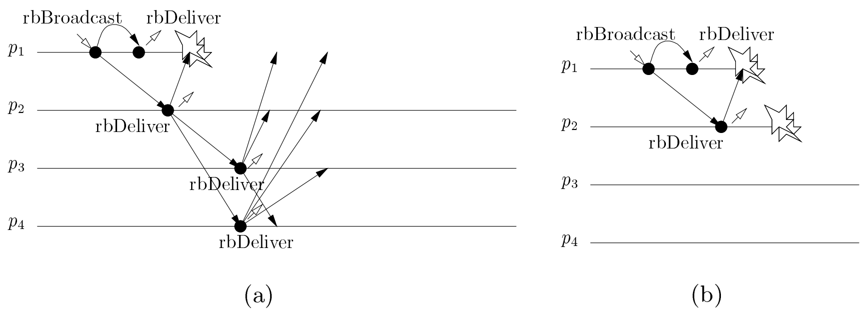

Figure 3.2. Sample executions of pessimistic reliable broadcast.

Correctness. All properties, except *agreement*, are ensured as in the previous 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.

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.

Performance. In the best case, to rbDeliver a message to all processes, the algorithm requires a single communication step and $(N-1)^2$ messages. In the worst case, if processes crash in sequence, N steps and $(N-1)^2$ messages are required to terminate the algorithm.

3.4 Uniform Reliable Broadcast

3.4.1 Specifications

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

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.

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.

Fig. 3.2b shows why our reliable broadcast algorithms do not ensure uniformity. Both process p_1 aNd p_2 rbDeliver the message as soon as they beb-Deliver 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).

Algorithm 3.4 Uniform reliable broadcast.

```
Implements:
    UniformReliableBroadcast (urb).

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

upon event ⟨ Init ⟩ do
    delivered := forward := $\emptyset$;
    correct := $\Pi$;
    $\text{ack}_m := \emptyset, \forall_m$;

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

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

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

upon (correct $\subset \text{ack}_m$) $\wedge$ ($m \notin$ delivered) do
    delivered := delivered $\cup \{m\}$;
    trigger ⟨ urbDeliver, $s_m, m$ ⟩;
```

3.4.2 A Uniform Reliable Broadcast Algorithm

Basically, our previous algorithms do 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 considered the same algorithms and replaced the best-effort broadcast with a reliable broadcast, they would still not implement a uniform broadcast abstraction.

We give an algorithm that implements the uniform version of reliable broadcast in Alg. 3.4. Basically, in this algorithm, a process only delivers a message when it knows that the message has been seen by all correct processes. As in our pessimistic reliable broadcast algorithm, 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 Fig. 3.4.

Correctness. As before, except for uniform agreement, all properties are trivially derived from the properties of the best-effort broadcast. Uniform agree-

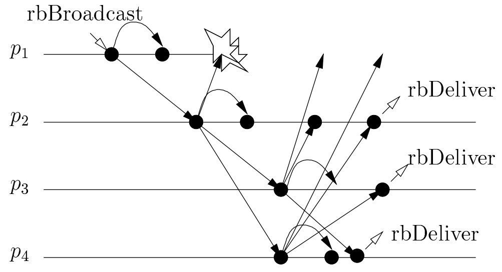

Figure 3.3. Sample execution of uniform reliable broadcast.

ment 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.

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 $(N - 1)^2$ messages in each step. Therefore, uniform reliable broadcast requires one more step to deliver the messages than its regular counterpart.

3.4.3 An Indulgent Uniform Reliable Broadcast Algorithm

The previous algorithm is not correct if the failure detector is not perfect. Uniform agreement would be violated if strong accuracy is not satisfied and validity would be violated if strong completeness is not satifsied.

We give in the following a uniform reliable broadcast algorithm that does not rely on a perfect failure detector but assumes a majority of correct processes. In the example above, this means that at most one out of three processes can crash in any given execution. The algorithm is given in Alg. 3.5. It is similar to the previous uniform reliable broadcast algorithm except that processes do not wait until all correct processes have seen a message, but until a majority has seen the message.

Correctness. The no-duplication and no-creation properties follow from the properties of best effort broadcast. Validity follows from the validity of the best effort broadcast primitive and the assumption of a correct majority. Uniform agreement is ensured because any two majorities intersect.

Performance. Similar to the previous algorithm.

3.5 Causal Order Broadcast

So far, we did not consider any ordering guarantee among messages delivered by different processes. In this section, we discuss the issue of ensuring delivery

Algorithm 3.5 Indulgent uniform reliable broadcast.

```
Implements:
      UniformReliableBroadcast (urb).

Uses:
      BestEffortBroadcast (beb).

upon event ⟨ Init ⟩ do
      delivered := forward := $\emptyset$;
      ack$_m$ := $\emptyset, \forall_m$;

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

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

upon (#ack$_m$ > N/2) $\wedge$ ($m \notin$ delivered) do
      delivered := delivered $\cup \{m\}$;
      trigger ⟨ urbDeliver, $s_m, m$ ⟩;
```

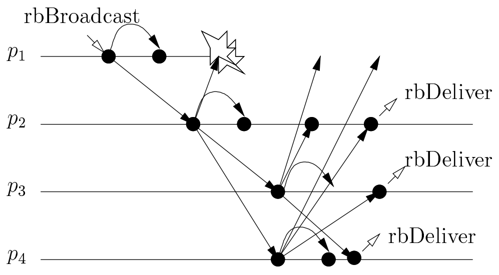

Figure 3.4. Sample execution of uniform reliable broadcast.

according to *causal ordering*. This is a generalization of FIFO (*first-in-first-out* ordering where messages from the same process should be delivered in the order according to which they were broadcast.

3.5.1 Specifications

As the name indicates, a causal order protocol ensures that messages are delivered respecting cause-effect relations, expressed by the *happened-before* relation introduced by Lamport (Lamport 1978). This relation, called the *causal order* relation, when applied to the messages exchanged among processes, can be written in terms of broadcast and delivery events. In this case,

Module:

Name: CausalOrder (co).

Events:

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

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

Properties:

CB: *Causal delivery:* If some process p_i delivers a message m_2 and there is a message m_1 such that $m_1 \to m_2$, then m_1 is delivered by p_i before m_2.

Module 3.4 Properties of causal broadcast.

we say that a message m_1 may potentially have caused another message m_2 (or m_1 happened before m_2), denoted as $m_1 \to m_2$ if the following relation, applies:

- m_1 and m_2 were broadcast by the same process p and m_1 was broadcast before m_2 (Fig. 3.5a).
- m_1 was delivered by process p and m_2 was broadcast also by process p and m_2 was broadcast after the delivery of m_1 (Fig. 3.5b).
- there exists some message m' such that $m_1 \to m'$ and $m' \to m_2 (Fig. 3.5c)$.

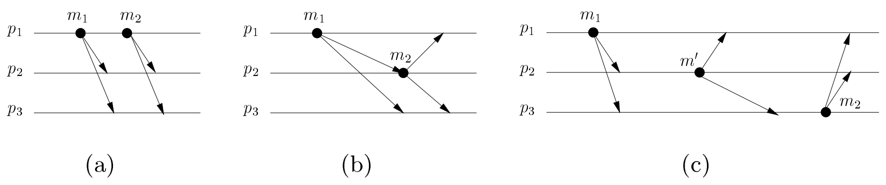

Figure 3.5. Causal order of messages.

Using the causal order relation, a causal order broadcast can be defined by the property CB in Mod. 3.4. The property states that messages are delivered by the protocol according to the causal order relation. There must be no "holes" in the causal past, i.e., when a message is delivered, all preceding messages have already been delivered.

Before describing our algorithm, we give a simple example of the practical advantage of a causal order primitive. Consider the case of a distributed message board that manages two types of messages: proposals and comments to previous proposals. To make the interface user-friendly, comments are depicted attached to the proposal they referring to. Assume that we implement

Module:

Name: ReliableCausalOrder (rco).

Events:

⟨ *rcoBroadcast*, *m* ⟩ and ⟨ *rcoDeliver*, src, m ⟩: with the same meaning and interface as the causal order interface.

Properties:

RB1-RB3, **URB4**, from reliable broadcast and and **CB** from causal order broadcast.

Module 3.5 Properties of reliable causal broadcast.

Module:

Name: UniformReliableCausalOrder (urco).

Events:

⟨ *urcoBroadcast*, *m* ⟩ and ⟨ *urcoDeliver*, src, m ⟩: with the same meaning and interface as the causal order interface.

Properties:

URB1-URB4 and **CB**, from uniform reliable broadcast and causal order.

Module 3.6 Properties of uniform reliable causal broadcast.

the application by replicating all the information at all participants. This can be achieved through the use of a reliable broadcast primitive to disseminate both proposals and comments. Without causal ordering, the following sequence would be possible: participant p_1 broadcasts a message m_1 containing a new proposal; participant p_2 delivers m_1 and disseminates a comment in message m_2; due to message delays, another participant p_3 delivers m_2 before m_1. In this case, the application at p_3 would be forced to log m_2 and wait for m_1, to avoid presenting the comment before the proposal being commented. Since $m_1 \rightarrow m_2$, a causal order primitive would make sure that m_1 would have been delivered before m_2, relieving the application programmer of such a task.

It is also worth noting that causal order can be combined with both reliable broadcast and uniform reliable broadcast. These combinations would have the interface and properties of Mod. 3.5 and Mod. 3.6, respectively.

In the following, we present algorithms that implement causal broadcast. The algorithms assume that all messages are broadcast to all group members. It is also possible to ensure causal delivery in the cases where individual messages may be sent to an arbitrary subset of group members, but the algorithms require a significantly larger amount of control information (Raynal, Schiper, and Toueg 1991).

Algorithm 3.6 Non-blocking reliable causal broadcast.

```
Implements:
    ReliableCausalOrder (rco).

Uses:
    ReliableBroadcast (rb).

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

upon event ⟨ rcoBroadcast, $m$ ⟩ do
    trigger ⟨ rbBroadcast, [DATA, past, $m$] ⟩;
    past := past $\cup$ { [self,$m$] };

upon event ⟨ rbDeliver, $p_i$, [DATA, $past_m$, $m$] ⟩ do
    if $m \notin$ delivered then
        forall $[s_n, n] \in past_m$ do //in order
            if $n \notin$ delivered then
                trigger ⟨ rcoDeliver, $s_n, n$ ⟩;
                delivered := delivered $\cup \{n\}$
                past := past $\cup \{[s_n, n]\}$;
        trigger ⟨ rcoDeliver, $p_i, m$ ⟩;
        delivered := delivered $\cup \{m\}$
        past := past $\cup \{[p_i, m]\}$;
```

3.5.2 A Non-Blocking Algorithm

The first algorithm we present is a reliable causal broadcast algorithm. It is inspired by one of the earliest implementations of causal ordering, included in the ISIS toolkit (Birman and Joseph 1987). The algorithm uses an underlying reliable broadcast communication primitive defined through rbBroadcast and rbDeliver primitives. The same algorithm could be used to implement a uniform reliable causal broadcast primitive, simply by replacing the underlying reliable broadcast module by a uniform reliable broadcast module.

The algorithm is said to be *non-blocking* in the following sense: whenever a process rbDeliver a message m, it can rcoDeliver m without waiting for other messages to be rbDelivered. Each message m carries a control field called $past_m$. The $past_m$ field of a message m includes all messages that causally precede m. When a message m is rbDelivered, $past_m$ is first inspected: messages in $past_m$ that have not been rcoDelivered must be rcoDelivered before m itself is also rcoDelivered. In order to record its own causal past, each process p memorizes all the messages it has rcoBroadcast or rcoDelivered in a local variable $past_p$. Note that $past_p$ (and $past_m$) are ordered sets. The algorithm is depicted in Alg. 3.6.

The biggest advantage of this algorithm is that the delivery of a message is never delayed in order to enforce causal order. This is illustrated in Fig. 3.6.

Consider for instance process p_4 and message m_2. Process p_4 rbDelivers m_2. Since m_2 carries m_1 in its past, m_1 and m_2 are delivered in order. Finally, when m_1 is rbDelivered from p_1, it is discarded.

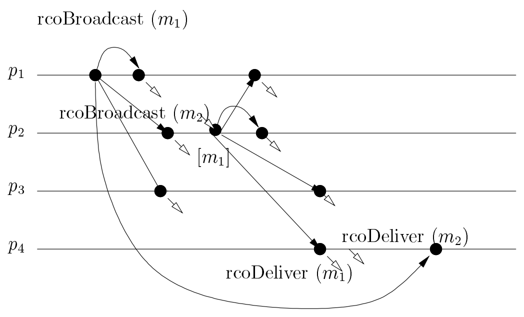

Figure 3.6. Sample execution of causal broadcast with complete past.

There is a clear inconvenience however: the $past_m$ field may become extremely large, since it includes the complete causal past of m. In the next subsection we illustrate a simple scheme to reduce the size of *past*. However, even with this optimization, this approach consumes too much bandwidth to be used in practice. Note also that no effort is made to prevent the size of the *delivered* set from growing indefinitely.

Correctness. All properties of reliable broadcast follow from the use of an underlying reliable broadcast primitive and the non-blocking flavour of the algorithm. The causal order property is enforced by having every message carry its causal past and every process making sure that it rcoDelivers the causal past of a message before rcoDelivering the message.

Performance. The algorithm does not add additional communication steps or messages to the underlying uniform reliable boradcast protocols. However, the size of the messages grows linearly with time, unless some companion garbage collection algorithm to purge *past* is executed.

3.5.3 Garbage Collection

We now present a very simple algorithm to delete messages from the *past* set. The algorithm is aimed to be used in conjonction with Alg. 3.6. It works as follows: when a process delivers a message, it rbBroadcasts an *Ack* message to all other processes; when an *Ack* for message m has been rbDelivered from all correct processes, m is purged from *past*. The pseudo-code of the algorithm is presented in Alg. 3.7.

Algorithm 3.7 Garbage collection of past.

```
Implements:
    GarbageCollectionOfPast.
    (extends Alg. 3.6).

Uses:
    ReliableBroadcast (rb).
    PerfectFailureDetector ($\mathcal{P}$);

upon event ⟨ Init ⟩ do
    delivered := past := $\emptyset$;
    correct := $\Pi$;
    ack$_m$ := $\emptyset, \forall_m$;

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

upon $\exists_m \in$ delivered: self $\notin$ ack$_m$ do
    ack$_m$ := ack$_m$ $\cup$ { self };
    trigger ⟨ rbBroadcast, [ACK, $m$] ⟩;

upon event ⟨ rbDeliver, $p_i$, [ACK, $m$] ⟩ do
    ack$_m$ := ack$_m$ $\cup$ $\{p_i\}$;
    if correct $\subset$ ack$_m$do
        past := past $\setminus\{[s_m, m]\}$;
```

3.5.4 A Blocking Algorithm

We now present an alternative algorithm that circumvents the main limitation of the previous algorithm: the huge size of the messages. Instead of keeping a record of all past messages, we keep just the sequence number of the last message rcoBroadcast. In this way, $past_p$ is reduced to an array of integers. Temporal information stored in this way is known as a *vector clock* (). The algorithm uses an underlying reliable broadcast communication primitive defined through rbBroadcast and rbDeliver primitives. The algorithm is depicted in Alg. 3.8.

With this algorithm, messages do not carry the complete past, only a summary of the past in the form of the vector clock. It is possible that a message may be prevented from being rcoDelivered immediately when it is rbDelivered, because some of the preceding messages have not been rbDelivered yet. It is also possible that the rbDelivery of a single message triggers the rcoDelivery of several messages that were waiting to be rcoDelivered. For instance, in Fig. 3.7 message m_2 is rbDelivered at p_4 before message m_1, but its rcoDelivery is delayed until m_1 is rbDelivered and rcoDelivered.

As with the non-blocking variant, this algorithm could also be used to implement uniform reliable causal broadcast, simply by replacing the underlying reliable broadcast module by a uniform reliable broadcast module.

Algorithm 3.8 Blocking causal broadcast.

```
Implements:
    ReliableCausalOrder (rco).

Uses:
    ReliableBroadcast (rb).

upon event ⟨ init ⟩ do
    $\forall_{p_i \in \Pi} : VC[p_i] := 0$;

upon event ⟨ rcoBroadcast, m ⟩ do
    VC[self] := VC[self]+1;
    trigger ⟨ rbBroadcast, [Data, self, VC, m] ⟩;

upon event ⟨ rbDeliver, $p_i$, [Data, $s_m$, $VC_m$, m] ⟩ do
    wait until $((VC[s_m] \geq VC_m[s_m] - 1)$ and $(\forall_{p_j \neq s_m} : VC_m[p_j] \leq VC[p_j])$
    trigger ⟨ rcoDeliver, $s_m$, m ⟩;
    if $s_m \neq$ self then
        VC[self] := VC[self]+1;
```

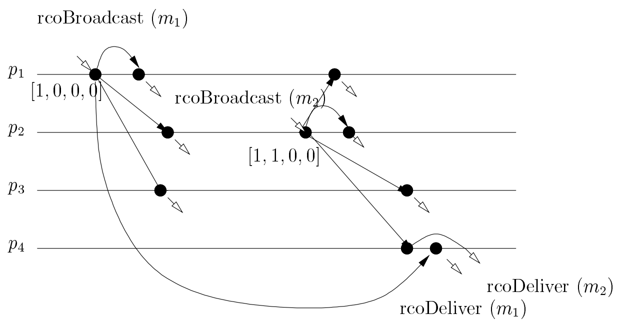

Figure 3.7. Sample execution of causal broadcast with vector clocks.

Performance. The algorithm does not add any additional communication steps or messages to the underlying reliable boradcast algorithm. The size of the message header is linear with regard to the number of processes in the system.

Exercises

1. Modify the optimistic reliable broadcast algorithm to reduce the number of messages sent in case of failures.

2. The algorithms 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 optimistic reliable broadcast algorithm
2. *delivered* in all reliable broadcast algorithms
3. $forward$ in the uniform reliable broadcast algorithm

3. What do we gain if we replace bebBroadcast with rbBroadcast in the uniform reliable broadcast algorithm?

4. What happens in the reliable broadcast and the uniform broadcast algorithms if the following properties of the failure detector are violated:

1. accuracy
2. completeness

5. Our uniform reliable broadcast algorithm using a perfect failure detector can be viewed as an extension of our pessimistic reliable broadcast algorithm. Would we gain anything by devising a uniform reliable broadcast algorithm that would be an extension of our optimistic reliable algorithm? i.e., can we have the processes not relay messages unless they suspect the sender.

6. Explain, say for a system of three processes, why our uniform reliable broadcast algorithm using a perfect failure detector would be incorrect if the failure detector turns out not to be perfect?

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

8. Compare our causal broadcast property with the following property: *"If a process delivers messages m_1 and m_2, and $m_1 \rightarrow m_2$, then the process must deliver m_1 before m_2".*

9. Does it make sense to have a causal order primitive that is best effort but not reliable?

10. Suggest a blocking and a non-blocking algorithm that ensure only FIFO delivery (i.e. causality is restricted to messages from the same process).

11. Suggest a modification of the garbage collection scheme to collect messages sooner in Algorithm 1.

Corrections

1. In our optimistic 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.

2. From $from[p_i]$ in the optimistic 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.

3. 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.

4. 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.

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.

5. The advantage of the optimistic 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 optimistic scheme would be of no benefit here.

6. 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. 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.

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.

8. We have the two following causal properties:

1. If a process delivers a message m_2, then it must have delivered every message m_1 such that $m_1 \rightarrow m_2$.
2. If a process delivers messages m_1 and m_2, and $m_1 \rightarrow m_2$, then the process must deliver m_1 before m_2.

Property 1 says that *any* message m_1 that causally precedes m_2 must only be delivered before m_2 if m_2 is delivered. Property 2 says that *any delivered* message m_1 that causally precedes m_2 must only be delivered before m_2 if m_2 is delivered.

Both properties are safety properties (because if a process does not receive a message m_2 (Property 1) or messages m_1 and m_2 (Property 2), it does not violate anything). The first property is clearly stronger than the second. If the first is satisfied then the second is. However, it can be the case with the second property that a process delivers a message m_2 without delivering a message m_1 that causally preceedes m_1.

9. No. Assume that we have a causal order broadcast that is not reliable but best effort. We define this abstraction with primitives: coBroadcast and coDeliver. Consider for instance the case where a process coBroadcasts a message m and crashes, and only one correct process p coDelivers m. If p then coBroadcasts a message m' such that $m \rightarrow m'$, then every correct process must coDeliver m'. To respect causal ordering, no process should coDeliver m' before m: a contradiction. Hence, any best-effort causal order broadcast is also reliable.

10. For a non-blocking algorithm that only ensures FIFO delivery among the messages of the same process, it is enough if each process sends along with each message all the messages *it* sent in the past. A process that receives such a message starts by delivering messages of that process from the past that it has not received yet, before delivering the new message.

For a blocking algorithm that only ensures FIFO delivery among the messages of the same process, it is enough if each process sends along with each message the number of messages that it has previously sent. A process receiving such a message waits to have delivered that many messages from the first process before delivering the new message.

11. When removing a message m from the past, we can also remove all the messages that causally depend on this message—and then recursively those that causally precedethese. This means that a message stored in the past must be stored with its own, distinct past.

4. Consensus

Life is what happens to you while you are making other plans.
(John Lennon)

This chapter considers the consensus abstraction. The processes use this abstraction to individually propose an initial value and eventually agree on a common final value among one of these initial values. We give specifications of this abstraction and algorithms that implement these specifications. We do so in a distributed system model where communication channels are reliable but processes can fail by crashing.

We show later in the chapter how consensus can be used to build a strong form of reliable broadcast: total order broadcast. Later in the manuscript, we will use the consensus abstractions to build more sophisticated forms of agreements.

4.1 Regular Consensus

4.2 Specifications

Consensus (sometimes we say regular consensus) is specified in terms of two primitives: *propose* and *decide*. Each process has an initial value that it proposes to the others (through the primitive *propose*). (Note that the act of proposing is local and typically triggers broadcast events.) All correct processes have to decide on a single value (through the primitive *decide*) that has to be one of the proposed values. Consensus must satisfy the properties C1-4 listed in Mod. 4.1.

In the following, we present two different algorithms to implement consensus. Both algorithms use a perfect failure detector module.

Module:

Name: (regular) Consensus (c).

Events:

Request: ⟨ *cPropose*, v ⟩: Used to propose a value for consensus.

Indication: ⟨ *cDecide*, v ⟩: Used to indicate the decided value for consensus.

Properties:

C1: *Termination:* Every correct process eventually decides some value.

C2: *Validity:* If a process decides v, then v was proposed by some process.

C3: *Integrity:* No process decides twice.

C4: *Agreement:* No two correct processes decide differently.

Module 4.1 Interface and properties of consensus.

4.2.1 A Flooding Algorithm

The first algorithm is presented in Alg. 4.1. Besides a perfect failure detector, it uses a best effort broadcast and a reliable broadcast abstractions. The basic idea is the following. The processes follow sequential rounds. Each process keeps a set of proposed values that it augments when moving from a round to the next (and new proposed values are known). In each round, each process disseminates its own set to all processes using a best effort broadcast, i.e., it floods the system with all proposals it has seen. When a process gets a proposal set from another process, it merges this set with its own. Basically, in each round every process tries to make a global union of all proposal sets.

A round terminates when a set has been included from every process that has not been suspected in that round. Consensus is reached when all processes have the same set of proposed values. In a round where a new failure is detected, a process p_i is not sure of having exactly the same set of values as the other processes. This happens because the crashed process(es) may have broadcast some values to the other processes but not to p_i. Note that this issue is not specifically related to the use of a best-effort broadcast.

In order to know when it is safe to decide, each process keeps a record of how many processes were not suspected in the previous round and from how many processes it has got an input in the current round. If a round terminates with the same number of non-suspected processes as in the previous round, a decision can be made. The process applies some deterministic function to the set of accumulated values. In our case, it picks the minimum value and decides it. Then it disseminates the decision to all processes using a reliable broadcast.

An execution of the algorithm is illustrated in Fig. 4.1. Process p_1 crashes during the first round after broadcasting its proposal. Only p_2 sees that proposal. No other process crashes. Therefore, p_2 sees the proposals of all pro-

Algorithm 4.1 A flooding consensus algorithm.

```
Implements:
    Consensus (c);

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

upon event ⟨ Init ⟩ do
    correct := correct-last-round := $\Pi$;
    proposal-set := correct-this-round := $\emptyset$;
    decided := $\perp$;
    round := 1;

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

upon event ⟨ cPropose, $v$ ⟩ do
    proposal-set := $\{v\}$;
    trigger ⟨ bebBroadcast, [MySet, round, proposal-set] ⟩;

upon event ⟨ bebDeliver, $p_i$, [MySet, round, set] ⟩ do
    correct-this-round := correct-this-round $\cup$ $\{p_i\}$;
    proposal-set := proposal-set $\cup$ set;

upon correct $\subset$ correct-this-round do
    if correct-this-round = correct-last-round then
        trigger ⟨ rbBroadcast, [Decided, min (proposal-set)] ⟩;
    else
        correct-last-round := correct-this-round;
        correct-this-round := $\emptyset$;
        round := round +1;
        trigger ⟨ bebBroadcast, [MySet, round, proposal-set] ⟩;

upon event ⟨ rbDeliver, $p_i$, [Decided, v] ⟩ $\wedge$ (decided = $\perp$) do
    decided := v;
    trigger ⟨ cDecided, v ⟩;
```

cesses and may decide. It takes the *min* of the proposals and decides the value 3. Processes p_3 and p_4 detect the failure and cannot decide. So they advance to the next round. Note that if these processes took *min* of the proposals they had after round 1, they would decide differently. Since p_2 has decided, it disseminates its decision through a reliable broadcast. When the decision is delivered, processes p_3 and p_4 also decide 3.

Correctness. Validity and *integrity* follow from the algorithm and the properties of the communication abstractions. *Termination* follows from the fact that at round N at the latest, all processes that did not decide decide. *Agree-*

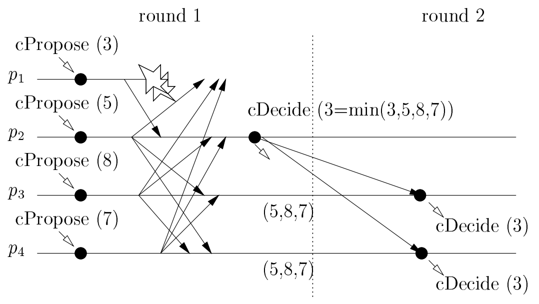

Figure 4.1. Sample execution of the flooding consensus algorithm.

ment is ensured because the min function is deterministic and is applied by all correct processes on the same set.

Performance. If there are no failures, the algorithm requires a single communication step. Each failure may cause at most one additional communication step (therefore, in the worst case the algorithm requires N steps, if $N-1$ processes crash in sequence). If there are no failures, the algorithm exchanges N^2 messages plus N messages for the reliable broadcast, for a grand total of $(N+1)N$ messages. There is an additional N^2 message exchanges for each round where a process crashes.

4.2.2 A Hierarchical Algorithm

We give here an alternative algorithm for regular consensus. The algorithm is interesting because it uses less messages and enables one process to decide before exchanging any message with the rest of the processes (0-latency). However, to reach a global decision, where all processes decide, the algorithm requires N communication steps. This algorithm is particularly useful if consensus is used as a service implemented by a set of server processes where the clients are happy with one value, as long as this value is returned very rapidly.

The algorithm is given in Alg. 4.2. It makes use of the fact that processes can be ranked according to their identity and this rank is used to totally order them a priori, i.e., $p_1 > p_2 > p3 > .. > p_N$. In short, the algorithm ensures that the correct process with the highest rank, i.e., the highest in the hierarchy, imposes its value on all the other processes. Basically, if p_1 does not crash, then it will impose its value to all: all correct processes will decide the value proposed by p_1. If p_1 crashes initially and p_2 is correct, then the algorithm ensures that p_2's proposal will be decided. A tricky issue that the algorithm handles is the case where p_1 is faulty but does not initially crash and p_2 is correct.

The algorithm works in rounds and uses a best effort broadcast abstraction. In the kth round, process p_k decides its proposal, and broadcasts it to all processes: all other processes in this round wait to deliver the message of p_k or to suspect p_k. None of these processes broadcast any message in this round. When a process p_k delivers in round $i < k$ the proposal of p_i, p_k adopts this proposal as its own new proposal.

Consider the example depicted in Fig. 4.2. Process p_1 broadcasts its proposal to all processes and crashes. Process p_2 and p_3 detect the crash before they deliver the proposal of p_1 and advance to the next round. Process p_4 delivers the value of p_1 and changes its own proposal accordingly. In round 2, process p_2 broadcasts its own proposal. This causes p_4 to change its proposal again. From this point on, there are no further failures and the processes decide in sequence the same value.

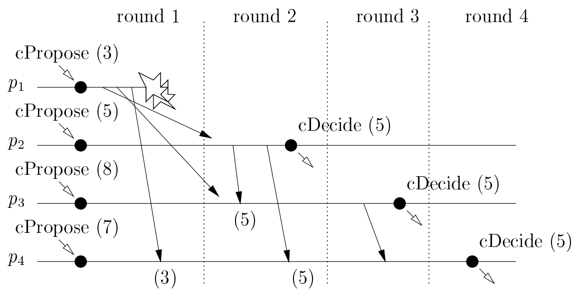

Figure 4.2. Sample execution of hierarchical consensus.

Correctness:. The *validity* and *integrity* properties follow from the algorithm and the use of an underlying best effort broadcast abstraction. *Termination* follows from the perfect failure detection assumption and the validity property of best effort broadcast: no process will remain indefinitely blocked in a round and every correct process p_i will eventually reach round i and decide in that round. Concerning *agreement*, assume the correct process p_i with the highest rank decides a value v. By the algorithm, every process p_j such that $j > i$ decides v: no process will suspect p_i and every process will adopt p_i's decision.

Performance. The algorithm exchanges $(N-1)$ messages in each round and can clearly be optimized such that it exchanges only $N(N-1)/2$: a process does not need to send a message to processes with a higher rank. The algorithm also requires N communication steps to terminate.

4.3 Uniform Consensus

4.3.1 Specification

As with reliable broadcast, we can define both regular and uniform variants of consensus. The uniform specification is presented in Mod. 4.2: correct pro-

Algorithm 4.2 A hierarchical consensus algorithm.

```
Implements:
    Consensus (c);

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

upon event ⟨ Init ⟩ do
    suspected := $\emptyset$;
    round := 1;
    proposal := nil;
    for i = 1 to N do pset[$i$] := $p_i$;
    for i = 1 to N do delivered[$round$] := false;

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

upon event ⟨ cPropose, $v$ ⟩ do
    proposal := $v$;

upon (pset[round] = self) $\wedge$ (proposal $\neq$ nil) do
    trigger ⟨ cDecided, proposal ⟩;
    trigger ⟨ bebBroadcast, proposal ⟩;

upon (pset[round] $\in$ suspected) $\vee$ (delivered[$round$] = true) do
    round := round + 1;

upon event ⟨ bebDeliver, pset[round],value ⟩ do
    if self.id $\geq$ round then
        proposal := value;
    delivered[$round$] := true;
```

cesses decide a value that must be consistent with values decided by crashed processes.

None of the consensus algorithms presented so far ensure uniform agreement. Roughly speaking, this is because some of the processes decide too early: without making sure that their decision has been seen by enough processes. Should they crash, other processes might have no choice but to decide something different.

In the following, we present two different algorithms to solve uniform consensus: each algorithm can be viewed as a uniform variant of one of our regular consensus algorithms above.

Module:

Name: UniformConsensus (uc).

Events:

$\langle$ *ucPropose*, v $\rangle$, $\langle$ *ucDecide*, v $\rangle$: with the same meaning and interface of the consensus interface.

Properties:

C1-C3: from consensus.

C4': *Uniform Agreement:* no two processes decide differently..

Module 4.2 Interface and properties of uniform consensus.

4.3.2 A Flooding Uniform Consensus Algorithm

We give our first uniform consensus algorithm in Alg. 4.3. The processes follow sequential rounds. As in our first consensus algorithm, each process gathers a set of proposals that it has seen and disseminates its own set to all processes using a best effort broadcast. An important difference here is that processes wait for round N before deciding.

Correctness:. Validity and *integrity* follow from the algorithm and the properties of best-effort broadcast. *Termination* is ensured here because all correct processes decide after round N. *Uniform agreement* is ensured because all processes that reach round N have the same set of values.

Performance. The algorithm requires N communication steps and $N * (N - 1)^2$ messages for all correct processes to decide.

4.3.3 A Hierarchical Uniform Consensus Algorithm

We give here an alternative algorithm that implements uniform consensus. The algorithm is depicted in Alg. 4.4. It is round-based and is similar to our second regular consensus algorithm. It is also hierarchical. It uses both a best-effort broadcast abstraction to exchange messages and a reliable broadcast abstraction to disseminate a decision.

Every round has a leader: process p_i is leader of round i. Unlike our hierarchical regular consensus algorithm, however, a round here consists of two communication steps: within the same round, the leader broadcasts a message to all, trying to impose its value, and then expects to get an acknowledgement from all. Processes that get a proposal from the coordinator of the round adopt this proposal as their own and send an acknowledgement back to the leader of the round. If it succeeds to collect an acknowledgement from all correct processes, the leader decides and disseminates the decided value using a reliable broadcast abstraction.

If the leader of a round fails, the correct processes detect this and change round. The leader is consequently changed. Processes in our algorithm do not

Algorithm 4.3 A flooding uniform consensus algorithm.

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

upon event ⟨ Init ⟩ do
    correct := $\Pi$;
    round := 1;
    for i = 1 to N do set[$i$] := delivered[$i$] := $\emptyset$;
    proposal-set := $\emptyset$;
    decided := false;

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

upon event ⟨ ucPropose, $v$ ⟩ do
    proposal-set := $\{v\}$;
    trigger ⟨ bebBroadcast, [MySet, round, proposal-set] ⟩;

upon event ⟨ bebDeliver, $p_i$, [MySet, round, newSet] ⟩ $\wedge$ ($p_i \in$ correct) do
    set[round] := set[round] $\cup$ newSet;
    delivered[round] := delivered[round] $\cup\{p_i\}$;

upon (correct $\subset$ delivered[round]) $\wedge$ (decided = false) do
    if round = N then
        decided := true;
        trigger ⟨ ucDecided, min(proposal-set) ⟩;
    else
        proposal-set := proposal-set $\cup$ set[round];
        round := round + 1;
        trigger ⟨ bebBroadcast, [MySet, round, proposal-set] ⟩;
```

move sequentially from one round to another: they might jump to a higher round if they get a message from that higher round.

An execution of the algorithm is illustrated in Fig. 4.3. Process p_1 imposes its value to all processes and receives an acknowledgment back from every process. Therefore, it can decide immediately. However, it crashes before disseminating the decision using the reliable broadcast. Its failure is detected by the remaining processes that, in consequence, move to the next round. The next leader, p_2 will in turn impose its proposal, which is now that of p_1: remember that p_2 has adopted the proposal of p_1. Since there are no further failures, process p_2 gets an acknowledgment from the remaining processes and disseminates the decision using a reliable broadcast.

Correctness:. *Validity* and *integrity* follow trivially from the algorithm and the properties of the underlying communication abstractions. Consider *termination.* If some correct process decides, it decides through the reliable broadcast abstraction, i.e., by rbDelivering a decision message. By the properties

Algorithm 4.4 A hierarchical uniform consensus algorithm.

```
Uses:
    PerfectPointToPointLinks (pp2p);
    ReliableBroadcast (rb).
    BestEffortBroadcast (rb).
    PerfectFailureDetector ($\mathcal{P}$);

upon event ⟨ Init ⟩ do
    proposal := decided := $\perp$;
    round := 1;
    suspected := ack-set := $\emptyset$;
    for i = 1 to N do pset[$i$] := $p_i$;

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

upon event ⟨ ucPropose, $v$ ⟩ do
    proposal := $v$;

upon (pset[round] = self) $\wedge$ (proposed $\neq \perp$) $\wedge$ (decided = $\perp$) do
    trigger ⟨ bebBroadcast, [PROPOSE, round, proposal] ⟩;

upon event ⟨ bebDeliver, $p_i$, [PROPOSE, round, v] ⟩ do
    proposal := v;
    trigger ⟨ pp2pSend, $p_i$, [ACK, Ack,round] ⟩;
    round := round +1;

upon event (pset[round] $\in$ suspected) do
    round := round +1;

upon event ⟨ pp2pDeliver, $p_i$, [ACK] ⟩ do
    ack-set := ack-set $\cup\{p_i\}$;

upon event (ack-set $\cup$ suspected = $\Pi$) do
    trigger ⟨ rbBroadcast, [DECIDED, proposal] ⟩;

upon event ⟨ rbDeliver, $p_i$, [DECIDED, v] ⟩ $\wedge$ (decided = $\perp$)do
    decided := v;
    trigger ⟨ ucDecide, v ⟩;
```

of this broadcast abstraction, every correct process rbDelivers the decision message and decides. Hence, either all correct processes decide or no correct process decides. Assume by contradiction that there is at least one correct process and no correct process decides. Let p_i be the correct process with the highest rank. By the *completeness* property of the perfect failure detector, every correct process suspects the processes with higher ranks than i (or bebDelivers their message). Hence, all correct processes reach round i and, by the *accuracy* property of the failure detector, no process suspects process p_i or moves to a higher round, i.e., all correct processes wait until a message

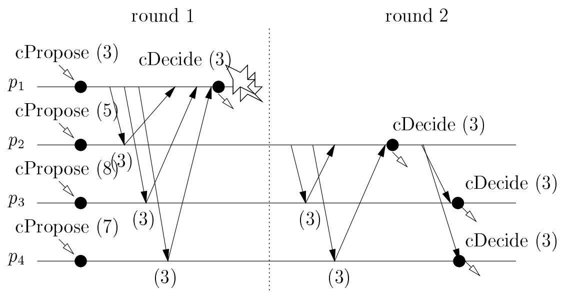

Figure 4.3. Sample execution of hierarchical uniform consensus.

from p_i is bebDelivered. In this round, process p_i hence succeeds in imposing a decision and decides. Consider now *agreement* and assume by contradiction that two processes decide differently. This can only be possible if two processes rbBroadcast two decision messages with two different propositions. Consider any two processes p_i and p_j, such that j is the closest integer to i such that $j > i$ and p_i and p_j proposed two different decision values v and v', i.e., rbBroadcast v and v'. Because p_i is correct and because of the accuracy property of the failure detector, process p_j must have adopted v before reaching round j. Given that j is the closest integer to i such that some process proposed v' different from v, after v was proposed, we have a contradiction.

Performance. If there are no failures, the algorithm terminates in 3 communication steps: 2 steps for the first round and 1 step for the reliable broadcast. It exchanges $3(N-1)$ messages. Each failure of a leader adds 2 additional communication steps and $2(N-1)$ additional messages.

4.4 Indulgent Consensus Algorithms

So far, the consensus and uniform consensus algorithms we have given rely on the assumption of a perfect failure detector. It is easy to see that in any of those algorithms, a false failure suspicion might lead to violation of the *agreement* property (exercice at the end of this chapter). That is, if a process is suspected to have crashed whereas the process is correct, agreement would be violated and two processes might decide differently. In other words, the consensus algorithms we have considered so far are not indulgent. In the following, we give two uniform consensus algorithms that rely only on the assumption of an eventually perfect failure detector. The algorithms can be viewed as *indulgent* variants of our second uniform consensus algorithm.

Our algorithms implement the uniform variant of consensus and rely on the assumption of a correct majority of processes. We leave it as exercices to show that any indulgent consensus algorithm that solves consensus solves uniform consensus (Chandra and Toueg 1996), and no indulgent algorithm can solve consensus without a correct majority of processes (Chandra and Toueg 1996; Guerraoui 2000).

4.4.1 The Traffic Light Consensus Algorithm

Our first algorithm is from (Chandra and Toueg 1996): it is given in Alg. 4.5. The algorithm is round-based and every process goes sequentially from round i to round $i+1$. Every round has a leader: the leader of round i is process $p_{i \ mod \ N+1}$, e.g., p_2 is the leader of rounds 2, $N+2$, $2N+2$, etc. Besides an eventually perfect failure detector, the algorithm uses a best-effort and a reliable broadcast communication abstractions.

We call this algorithm the *traffic light* consensus algorithm because the processes behave as cars in a cross-road controlled by a traffic light. Crossing the road in our context means deciding on a consensus value and having the chance to cross the road in our context means being leader of the current round. If two cars try to cross the road at the same time, none of them might succeed. If it does not cross the road, the car has to wait until all others are given their chance too. The guarantee that there eventually will only be one green light conveys the fact that only eventually, some correct process is not suspected and will hence be the only leader.

The process that is leader in a round computes a new proposal and tries to impose that proposal to all: every process that gets the proposal from the current leader adopts this proposal and assigns it the current round number as a timestamp. Then it acknowledges that proposal back to the leader. If the leader gets a majority of acknowledgements, it decides and disseminates that decision using a reliable broadcast abstraction. More precisely, a round consists a priori (i.e., if the leader is not suspected) of five phases.

1. *Computation.* The leader first computes its current proposal. All processes send their current proposal to the leader which selects the proposal with the highest timestamp. The leader starts the selection process after it has received the current proposal from at least a majority of the processes.
2. *Adoption.* The leader broadcasts its current proposal to all. Any process that gets that proposal adopts it and assigns it the current round number as a timestamp.
3. *Acknowledgment.* Every process that adopts a value from the leader sends an acknowledgement message back to the leader.
4. *Decision.* If the leader gets a majority of acknowledgement messages, it decides its proposal and uses a reliable broadcast primitive to disseminate the decision to all.
5. *Global decision.* Any process that delivers a decision message decides.

In every round, there is a critical point where processes need the input of their failure detector model. When the processes are waiting for a proposal from the leader of that round, the processes should not wait indefinitely if the leader has crashed without having broadcast its proposal. In this case, the processes consult their failure detector module to get a hint as to whether the leader process has crashed. Given that an eventually perfect detector ensures that, eventually, every crashed process is suspected by every correct process, the process that is waiting for a crashed leader will eventually suspect it and send a specific message *nack* to the leader, then move to the next round. In fact, a leader that is waiting for acknowledgements might get some *nacks*: in this case it moves to the next round without deciding.

Note also that processes after acknowledging a proposal move to the next round directly: they do not need to wait for a decision. They might deliver it in an asynchronous way: through the reliable broadcast dissemination phase. In that case, they will simply stop their algorithm.

Correctness:. *Validity* and *integrity* follow from the algorithm and the properties of the underlying communication abstractions. Consider *termination.* If some correct process decides, it decides through the reliable broadcast abstraction, i.e., by bebDelivering a decision message. By the properties of this broadcast abstraction, every correct process rbDelivers the decision message and decides. Assume by contradiction that there is at least one correct process and no correct process decides. Consider the time t after which all faulty processes crashed, all faulty processes are suspected by every correct process forever and no correct process is ever suspected. Let p_i be the first correct process that is leader after time t and let r denote the round at which that process is leader. If no process has decided, then all correct processes reach round r and p_i eventually reaches a decision and rbBroadcasts that decision. Consider now *agreement.* Consider by contradition any two rounds i and j, j is the closest integer to i such that $j > i$ and $p_{i\ mod\ N+1}$, and $p_{j\ mod\ N+1}$, proposed two different decision values v and v'. Process $p_{j\ mod\ N+1}$ must have adopted v before reaching round j. This is because $p_{j\ mod\ N+1}$ selects the value with the highest timestamp and $p_{j\ mod\ N+1}$ cannot miss the value of $p_{i\ mod\ N+1}$: any two majorities always intersect. Given that j is the closest integer to i such that some process proposed v' different from v, after v was proposed, we have a contradiction.

Performance:. If no process fails or is suspected to have failed, then 4 communication steps and $4(N-1)$ messages are needed for all correct processes to decide.

4.4.2 The Round-About Consensus Algorithm

Our second algorithm is from (Lamport 1989): it is given in Alg. 4.6. We call this algorithm the *round-about* consensus algorithm because the processes behave as cars in a cross-road controlled by a round-about.

Unlike in the previous indulgent consensus algorithm, if a process does not succeed in crossing the road, it might try again immediately and does not need to wait for all other processes to have had their chance to cross the road. In fact, the first to go on the round-about might succeed if no other process goes on the round-about. Basically, a process that goes on the round-about is the one that believes it is leader. The guarantee that there will be only one process on the round-about follows from the eventually perfect failure detector guarantee.

The algorithm is round-based but, unlike the previous algorithm, processes do not go sequentially from round i to round $i + 1$. The leader of round i is process $p_{i \ mod \ N+1}$: if it is the process among those that are not suspected with the highest rank, i.e., p_1 if no process is suspected. Besides an eventually perfect failure detector, the algorithm uses a best-effort and a reliable broadcast communication abstractions.

Like in the previous indulgent consensus algorithm, the process that is leader in a round computes a new proposal based on the timestamps of the estimates of the processes. The leader then tries to impose that proposal to all: every process that gets the proposal from the current leader, and that is not engaged in a higher round, adopts this proposal and assigns it as a timestamp the current round number. It then ackowledges that proposal back to the leader. If the process was engaged in a higher round or suspects the leader, it sends a *nack* message to the leader. If the leader gets a majority of acknowledgements, it decides and disseminates that decision using a reliable broadcast abstraction.

Correctness. Similar to that of the traffic light indulgent consensus algorithm.

Performance. If no process fails or is suspected to have failed, then 5 communication steps and $5(N - 1)$ messages are needed for all correct processes to decide.

4.5 Total Order Broadcast

A total order broadcast abstraction is a reliable broadcast abstraction which ensures that all processes deliver the same set of messages exactly in the same order. This abstraction is sometimes also called *atomic broadcast* because the message delivery occurs as an indivisible operation: the message is delivered to all or to none of the processes and, if it is delivered, other messages are ordered before or after this message.

This sort of ordering eases the maintenance of a global consistent state. In particular, if each participant is programmed as a state machine, i.e., its state at a given point depends exclusively of the initial state and of the sequence of messages received, the use of total order broadcast ensures consistent replicated behavior. The replicated state machine is one of the fundamental techniques to achieve fault-tolerance.

Note that total order is orthogonal to the causal order discussed in Sec. 3.5. It is possible to have a total-order abstraction that does not respects causal order, as well as it is possible to overlay a total order abstraction on top of a causal order primitive.

Note that with a causal order primitive, processes deliver messages according to cause-effect relations. Messages that are not related by causal order are said to be *concurrent* messages. A causal order abstraction that does not enforce total order may deliver concurrent messages in different order to different processes.

4.5.1 Specification

Two variants of the abstraction can be defined: A regular variant only ensure total order among the processes that remain correct; and a uniform variant that ensures total order with regard to the crashed processes as well. Total order is captured by the properties TO1 and RB1-4 depicted in Mod. 4.3 and TO1 and RB1-4 in Mod. 4.4.

Note that the total order property (uniform or not) can be combined with the properties of a uniform reliable broadcast or those of a causal broadcast abstraction (for conciseness, we omit to present the interface of these modules).

4.5.2 A total order broadcast algorithm

In the following, we give a uniform total order broadcast algorithm. More precisely, the algorithm ensures the properties of uniform reliable broadcast plus the uniform total order property. The algorithm, inspired by (Chandra and Toueg 1996), uses a uniform reliable broadcast and a uniform consensus abstractions as underlying building blocks. In this algorithm, messages are first disseminated using a uniform (but unordered) reliable broadcast primitive. Messages delivered this way are stored in a bag of unordered messages at every process. The processes then use the consensus abstraction to order the messages in this bag.

More precisely, the algorithm works in consecutive rounds. Processes go sequentially from round i to $i+1$: as long as new messages are broadcast, the processes keep on moving from one round to the other. There is one consensus instance per round. The consensus instance of a given round is used to make the processes agree on a set of messages to assign to the sequence number corresponding to that round: these messages will be delivered in that round. For instance, the first round decides which messages are assigned sequence number 1, i.e., which messages are delivered in round 1. The second round decides which messages are assigned sequence number 2, etc. All messages that are assigned round number 2 are delivered after the messages assigned round number 1. Messages with the same sequence number are delivered according to some deterministic order (e.g., based on message identifiers). That is, once the processes have agreed on a set of messages for a given round, they simply apply a deterministic function to sort the messages of the same set.

In each instance of the consensus, every process proposes a (potentially different) set of messages to be ordered. The properties of consensus ensure that all processes decide the same set of messages for that sequence number. The total order algorithm is given in Alg.4.7. The *wait* flag is used to ensure that a new round is not started before the previous round has terminated.

An execution of the algorithm is illustrated in Fig. 4.4. The figure is unfolded into two parallel flows: That of the reliable broadcasts, used to disseminate the messages, and that of the consensus instances, used to order the messages. As messages are received from the reliable module they are proposed to the next instance of consensus. For instance, process p_4 proposes message m_2 to the first instance of consensus. Since the first instance of consensus decides message m_1, process p_4 re-submits m_2 (along with m_3 that was received meanwhile) to the second instance of consensus.

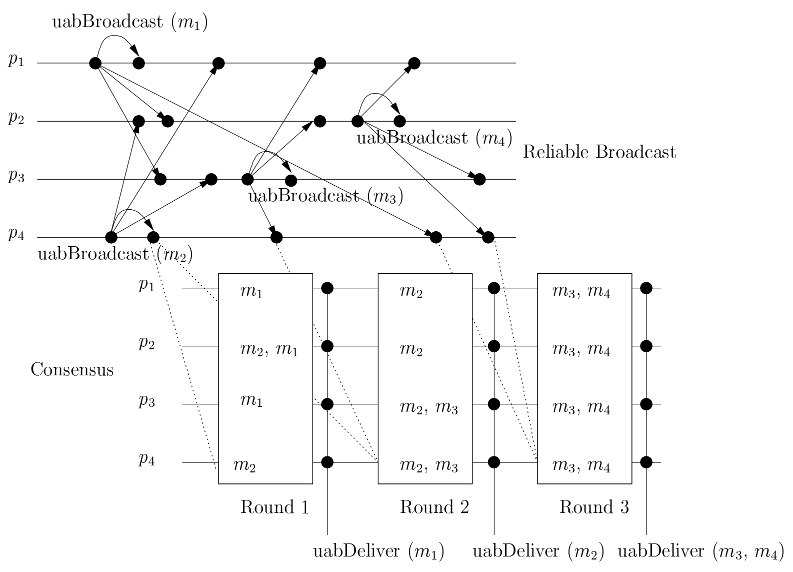

Figure 4.4. Sample excution of the uniform atomic broadcast algorithm.

Correctness. The integrity property follows from (1) the integrity property of the reliable broadcast abstraction and (2) the validity property of consensus. The no-duplication property follows from (1) the no-duplication property of the reliable broadcast abstraction, and (2) the integrity property of consensus (more precisely, the use of the variable delivery). Consider the agreement property. Assume that some correct process p_i toDelivers some message m. By the algorithm, p_i must have decided a batch of messages with m inside that batch. Every correct process will reach that point, because of the algorithm and the termination property of consensus, will decide that batch, and will toDeliver m. Consider the validity property of total order broadcast, and let p_i be some correct process that toDelivers a message m. Assume by contradiction that p_i never toDelivers m. This means that m is never included in a batch of messages that some correct process decides. By the validity property of reliable broadcast, every correct process will eventually rbDeliver and propose m in a batch of messages to consensus. By the validity property of consensus, p_i will decide a batch of messages including m and will toDeliver m. Consider now the total order property. Let p_i and p_j be any two processes that toDeliver some message m_2. Assume that p_i toDelivers some message m_1 before m_2. If p_i toDelivers m_1 and m_2 in the same batch (i.e., the same round number), then by the agreement property of consensus, p_j must have also decided the same batch. Thus, p_j must toDeliver m_1 before m_2 since we assume a deterministic funtion to order the messages for the same batch before their toDelivery. Assume that m_1 is from a previous batch at p_i. By the agreement property of consensus, p_j must have decided the batch of m_1 as well. Given that processes proceed sequentially from one round to the other, then p_j must have toDelivered m_1 before m_2.

Performance. The algorithm requires at least one communication step to execute the reliable broadcast and at least two communication steps to execute the consensus. Therefore, even if no failures occur, at least three communication steps are required.

Variations. It is easy to see that a regular total order broadcast algorithm is automatically obtained by replacing the uniform consensus abstraction by a regular one. Similarly, one could obtain a total order broadcast that satisfies uniform agreement if we used a uniform reliable broadcast abstraction instead of regular reliable broadcast abstraction. Finally, the algorithm can trivially be made to ensure in addition causal ordering, for instance if we add past information with every message (see our non-blocking causal order broadcast algorithm).

4.6 Exercices

1. Improve our hierarchical regular consensus algorithm to save one communication step. (The algorithm requires N communication steps for all correct processes to decide. By a slight modification, it can run in $N-1$ steps: suggest such a modification.)

2. Explain why none of our (regular) consensus algorithms ensures uniform consensus.

3. Can we optimize our flooding uniform consensus algorithms to save one communication step, i.e., such that all correct processes always decide after $N-1$ communication steps? Consider the case of a system of two processes.

4. What would happen in our flooding uniform consensus algorithm if:

1. we did not use $set[round]$ but directly update $proposedSet$ in **upon event** bebDeliver?
2. we accepted any bebDeliver event, even if $p_i \notin correct$?

5. Consider our consensus algorithms using a perfect failure detector. Explain why none of those algorithms would be correct if the failure detector turns out not to be perfect.

6. Explain why any indulgent algorithm that solves consensus actually solves uniform consensus.

7. Explain why any indulgent consensus algorithm needs a majority of correct processes.

8. Suggest improvements of our traffic light and round-about indulgent consensus algorithms such that, if no process fails or is suspected to have failed, only 3 communication steps and $3(N-1)$ messages are needed for all correct processes to decide.

9. What happens in our total order broadcast algorithm if the set of messages decided on are not sorted deterministically after the decision but prior to the proposal? What happens if in our total order broadcast algorithm if the set of messages decided on is not sorted deterministically, neither a priori nor a posteriori?

4.7 Corrections

1. The last process (say, p_N) does not need to broadcast its message. Indeed, the only process that uses p_N's broadcast value is p_N itself, and p_N anyway decides its proposal just *before* it broadcasts it (not when it delivers it). Clearly, no process ever uses p_N's broadcast. More generally, no process p_i ever uses the value broadcast from any process p_j such that $i \geq j$.

2. Consider our flooding algorithm first and the scenario of Fig. 4.1: if p_1 crashes after deciding 3, p_2 and p_3 would decide 5. Now consider our hierarchical algorithm and the scenario of Fig. 4.2. In the case where p_1 decides and crashes and no other process sees p_1's proposal (i.e., 3), then p_1 would decide differently from the other processes.

3. No. Rather than giving a general proof, we give a counter example for the particular case of $N = 2$. The interested reader will then easily extend beyond this case to the general case of any N. Consider the system made of two processes p_1 and p_2. We exhibit an execution where processes do not reach uniform agreement after one round, thus they need at least two rounds. More precisely, consider the execution where p_1 and p_2 propose two different values, that is, $v_1 \neq v_2$, where v_i is the value proposed by p_i ($i = 1, 2$). Without loss of generality, consider that $v_1 < v_2$. We shall consider the following execution where p_1 is a faulty process.

During round 1, p_1 and p_2 respectively send their message to each other. Process p_1 receives its own value and p_2's message (p_2 is correct), and decides. Assume that p_1 decides its own value v_1, which is different from p_2's value, and then crashes. Now, assume that the message p_1 sent to p_2 in round 1 is arbitrarily delayed (this is possible in an asynchronous system). There is a time after which p_2 permanently suspects p_1 because of the Strong Completeness property of the perfect failure detector. As p_2 does not know that p_1 did send a message, p_2 decides at the end of round 1 on its own value v_2. Hence the violation of uniform agreement.

Note that if we allow processes to decide only after 2 rounds, the above scenario does not happen, because p_1 crashes *before* deciding (i.e. it never decides), and later on, p_2 decides v_2.

4. For case (1), it would not change anything. Intuitively, the algorithm is correct (more specifically, preserves uniform agreement), because any process executes for N rounds before deciding. Thus there exists a round r during which no process crashes. Because at each round, every process broadcasts the values it knows from the previous rounds, after executing round r, all processes that are not crashed know exactly the same information. If we now update *proposedSet before* the beginning of the next round (and in particular before the beginning of round r), the processes will still have the information on time. In conclusion, the fact they get the information earlier is not a problem since they must execute N rounds anyway.

In case (2), the algorithm is not correct anymore. In the following, we discuss an execution that leads to disagreement. More precisely, consider the system made of three processes p_1, p_2 and p_3.

The processes propose 0, 1 and 1, respectively. During the first round, the messages of p_1 are delayed and p_2 and p_3 never receive them. Process p_1 crashes at the end of round 2, but p_2 still receives p_1's round 2 message (that is, 0) in round 2 (possible because channels are not FIFO). Process p_3 does not receive p_1's message in round 2 though. In round 3, the message from p_2

to p_3 (that is, the set $\{0, 1\}$) is delayed and process p_2 crashes at the end of round 3, so that p_3 never receives p_2's message. Before crashing, p_2 decides on value 0, whereas p_3 decides on 1. Hence the disagreement.

5. In all our algorithms using a perfect failure detector, there is at least one critical point where a correct process p waits to deliver a message from a process q or to suspect the process q. Should q crash and p never suspect q, p would remain blocked forever and never decide. In short, in any of our algorithm using a perfect failure detector, a violation of *strong completeness* could lead to violate the *termination* property of consensus.

Consider now *strong accuracy*. Consider our flooding algorithm first and the scenario of Fig. 4.1: if p_1 crashes after deciding 3, and p_1 is suspected to have crashed by p_2 and p_3, then p_2 and p_3 would decide 5. The same scenario can happen for our hierarchical consensus algorithm.

6. Consider any indulgent consensus algorithm that does not solve uniform consensus. This means that there is an execution scenario where two processes p and q decide differently and one of them crashes: the algorithm violates uniform agreement. Assume that process q crashes. With an eventually perfect failure detector, it might be the case that q is not crashed but just falsely suspected by all other processes. Process p would decide the same as in the previous scenario and the algorithm would violate (non-uniform) agreement.

7. We explain this for the case of a system of four processes $\{p_1, p_2, p_3, p_4\}$. Assume by contradiction that there is an indulgent consensus algorithm that tolerates the crash of two processes. Assume that p_1 and p_2 propose a value v whereas p_3 and p_4 propose a different value v'. Consider a scenario E_1 where p_1 and p_2 crash initially: in this scenario, p_3 and p_4 decide v' to respect the *validity* property of consensus. Consider also a scenario E_2 where p_3 and p_4 crash initially: in this scenario, p_1 and p_2 decide v. With an eventually perfect failure detector, a third scenario E_3 is possible: the one where no process crashes, p_1 and p_2 falsely suspect p_3 and p_4 whereas p_3 and p_4 falsely suspect p_1 and p_2. In this scenario E_3, p_1 and p_2 decide v, just as in scenario E_1, whereas p_3 and p_4 decide v', just as in scenario E_2. *Agreement* would hence be violated.

8. The optimization of the traffic light algorithm consists in skipping the first communication step of the algorithm during the first round. In this case, process p_1 does not really need to compute a proposal based on the estimates of other processes. This computation phase is actually only needed to make sure that the leader will propose any value that might have been proposed. For the first round, p_1 is sure that no decision has been made and can save one communication phase by directly proposing its own proposal.

A similar optimization can be applied to the round-about algorithm: we can safely remove the two first communication steps and have process p_1, when it is indeed leader in round 1, go ahead directly and propose its initial value whitout waiting for other values.

9. If the deterministic sorting is done prior to the proposal, and not a posteriori upon a decision, the processes would not agree on a set but on a sequence, i.e., an ordered set. If they then toDeliver the messages according to this order, we would still ensure the total order property.

If the messages that we agree on through consensus are not sorted deterministically within every batch (neither a priori nor a posteriori), then the total order property is not ensured. Even if the processes decide on the same batch of messages, they might toDeliver the messages within this batch in a different order.

In fact, the total order property would only be ensured up to the *batches* of messages, and not to the messages themselves. We thus get a coarser granularity in the total order.

We could avoid using the deterministic sort function at the cost of proposing a single message at a time in the consensus abstraction. This means that we would need exactly as many consensus instances as there are messages exchanged between the processes. If messages are generated very slowly by processes, the algorithm ends up using one consensus instance per message anyway. If the messages are generated rapidly, then it is beneficial to use several messages per instance: within one instance of consensus, several messages would be gathered, i.e., every message of the consensus algorithm would concern several messages to toDeliver. Agreeing on several messages at the same time reduces the number of times we use the consensus protocol.

Algorithm 4.5 The traffic light consensus algorithm.

```
Uses:
    PerfectPointToPointLinks (pp2p);
    ReliableBroadcast (rb).
    BestEffortBroadcast (beb).
    EventuallyPerfectFailureDetector ($\diamond\mathcal{P}$);

upon event ⟨ Init ⟩ do
    proposal := decided := ⊥;
    round := 1;
    suspected:= estimate-set[] := ack-set[] := ∅;
    estimate[] := ack[] := false;
    for i = 1 to N do ps[$i$] := $p_i$;

upon event ⟨ suspect, $p_i$ ⟩ do
    suspected := suspected ∪{$p_i$};

upon event ⟨ restore, $p_i$ ⟩ do
    suspected := suspected \{$p_i$};

upon event ⟨ ucPropose, $v$ ⟩ do
    proposal := [$v, 0$];

upon event (proposal ≠ ⊥) ∧ (estimate[round] = false) do
    estimate[round] := true;
    trigger ⟨ pp2pSend, ps[round mod N], [ESTIMATE, round, proposal] ⟩;

upon event(ps[round mod N + 1]= self) ∧ ⟨ pp2pDeliver, $p_i$, [ESTIMATE, round, estimate] ⟩ do
    estimate-set[round] := estimate-set[round] ∪{$estimate$};

upon (ps[round mod N + 1]= self) ∧ (#estimate[round] > N/2)) do
    proposal := highest(estimate-set[round]);
    trigger ⟨ bebBroadcast, [PROPOSE, round, proposal] ⟩;

upon event ⟨ bebDeliver, $p_i$, [PROPOSE, round, value] ⟩ ∧ (ack[round] = false) do
    ack[round] := true;
    proposal := [$value, round$];
    trigger ⟨ pp2pSend, ps[round mod N], [ACK, round] ⟩;
    round := round + 1;

upon event (ps[round mod N] ∈ suspected) ∧ (ack[round] = false) do
    ack[round] := true;
    trigger ⟨ pp2pSend, ps[round mod N], [NACK, round] ⟩;
    round := round + 1;

upon event(ps[round mod N + 1]= self) ∧ ⟨ pp2pDeliver, $p_i$, [ACK, round] ⟩ do
    ack-set[round] := ack-set[round] ∪{ value };

upon (ps[round mod N + 1]= self) ∧ ⟨ pp2pDeliver, $p_i$, [NACK, round] ⟩ do
    round := round + 1;

upon (ps[round mod N + 1]= self) ∧ (#ack-set[round] > N/2) do
    trigger ⟨ rbBroadcast, [DECIDE, proposal] ⟩;

upon event ⟨ rbDeliver, $p_i$, [DECIDED, v] ⟩ ∧ (decided = ⊥) do
    decided := v;
    trigger ⟨ ucDecided, v ⟩;
```

Algorithm 4.6 The round-about consensus algorithm.

```
Uses:
    PerfectPointToPointLinks (pp2p);
    ReliableBroadcast (rb).
    BestEffortBroadcast (beb).
    EventuallyPerfectFailureDetector ($\Diamond\mathcal{P}$);

upon event ⟨ Init ⟩ do
    proposal := decided := $\bot$;
    estimate-set[] := ack-set[] := $\emptyset$;
    estimate[] := ack[] := false;
    $\text{round}_1$ := $\text{round}_2$ := 1;
    correct := $\Pi$;
    for i = 1 to N do pset[$i$] := $p_i$;

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

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

upon event ⟨ ucPropose, $v$ ⟩ do
    proposal := $v$;
    for i = 1 to N do
        if pset[$i$] = self then
            $\text{round}_1$ := $i$;

upon (highest(correct) = self) $\wedge$ (estimate[$\text{round}_1$] = false) $\wedge$ (decided = $\bot$) do
    estimate[$\text{round}_1$] = true;
    trigger ⟨ bebBroadcast, [ReqEstimate, $\text{round}_1$] ⟩;

upon event ⟨ pp2pDeliver, $p_i$, [ReqEstimate, r] ⟩ do
    if $\text{round}_2$ > r then
        trigger ⟨ pp2pSend, $\text{p}_i$, [Nack, $\text{round}_2$] ⟩;
    else
        $\text{round}_2$ := r;
        trigger ⟨ pp2pSend, $\text{p}_i$, [Estimate, $\text{round}_2$, proposal] ⟩;

upon event ⟨ pp2pDeliver, $p_i$, [Nack, round] ⟩ do
    repeat
        $\text{round}_1$ := $\text{round}_1$ + N;
    until ($\text{round}_1 \geq$ round);

upon event ⟨ pp2pDeliver, $p_i$, [Estimate, $\text{round}_1$,value] ⟩ do
    estimate-set[$\text{round}_1$] := estimate-set[$\text{round}_1$] $\cup\{value\}$;

upon (highest(correct) = self) $\wedge$ (#estimate[$\text{round}_1$] > N/2) do
    proposal := highest(estimate-set[$\text{round}_1$]);
    trigger ⟨ bebBroadcast, [Propose, $\text{round}_1$, proposal] ⟩;

upon event ⟨ pp2pDeliver, $p_i$, [Propose, r, proposal] ⟩ do
    if $\text{round}_2$ > r then
        trigger ⟨ pp2pSend, $\text{p}_i$, [Nack, $\text{round}_2$] ⟩;
    else
    $\text{round}_2$ := r;
        trigger ⟨ pp2pSend, $\text{p}_i$, [Ack, $\text{round}_2$] ⟩;

upon event (highest(correct) = self)  $\wedge$ ⟨ pp2pDeliver, $p_i$, [Ack, round] ⟩ do
    ack-set := ack-set $\cup\{$ value $\}$;

upon (highest(correct) = self) $\wedge$ (#ack-set[round] > N/2) do
    trigger ⟨ rbBroadcast, [Decide, proposal] ⟩;

upon event ⟨ rbDeliver, $p_i$, [Decided, v] ⟩ $\wedge$ (decided = $\bot$) do
    decided := v;
    trigger ⟨ ucDecided, v ⟩;
```

Module:

Name: TotalOrder (to).

Events:

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

Indication: $\langle$ *toDeliver*, src, m $\rangle$: Used to deliver message m sent by process src.

Properties:

TO1: *Total order:* Let m_1 and m_2 be any two messages. Let p_i and p_j be any two correct processes that deliver m_2. If p_i delivers m_1 before m_2, then p_j delivers m_1 before m_2.

RB1-RB4: from reliable broadcast.

Module 4.3 Interface and properties of total order broadcast.

Module:

Name: UniformTotalOrder (uto).

Events:

$\langle$ *utoBroadcast*, m $\rangle$, $\langle$ *utoDeliver*, src, m $\rangle$: with the same meaning and interface of the consensus interface.

Properties:

UTO1: *Uniform total order:* Let m_1 and m_2 be any two messages. Let p_i and p_j be any two processes that deliver m_2. If p_i delivers m_1 before m_2, then p_j delivers m_1 before m_2.

RB1-RB4: from reliable broadcast.

Module 4.4 Interface and properties of uniform total order broadcast.

Algorithm 4.7 Uniform total order broadcast algorithm.

```
Implements:
    UniformTotalOrderBroadcast (uab);

Uses:
    UniformReliableBroadcast (rb).
    UniformConsensus (uc);

upon event ⟨ Init ⟩ do
    unordered := $\emptyset$;
    wait := false;
    sn := 1;

upon event ⟨ toBroadcast, $m$ ⟩ do
    trigger ⟨ rbBroadcast, $m$ ⟩;

upon event ⟨ rbDeliver, $s_m$, $m$ ⟩ do
    unordered := unordered $\cup\{(s_m, m)\}$

upon (unordered $\neq \emptyset$) $\wedge$ ($\neg$ wait) do
    wait := true;
    trigger ⟨ ucPropose, sn,unordered ⟩;

upon event ⟨ ucDecided, sn, decided ⟩ do
    unordered := unordered \ decided;
    decided := sort (decided); // some deterministic order;
    $\forall_{(s_m,m)} \in$ decided: trigger ⟨ toDeliver, $s_m, m$ ⟩; //following a deterministic order
    sn := sn +1;
    wait := false;
```

5. Failure-Sensitive Agreement

So when they continued asking him, he lifted up himself, and said unto to them, he that is without sin among you, let him first cast a stone at her.
(John (not Lennon) 8:7)

This chapter considers *failure sensitive* agreement abstractions. These abstractions are similar to consensus in that processes need to agree on some common value. The very characteristic of these abstractions is that the value decided might depend on whether processes have crashed or not. They are in this sense failure-sensitive. The impact of this failure-sensitivity is that these abstractions cannot be implemented with indulgent algorithms. They do not forget the mistakes of their failure detector.

Examples of such abstractions include terminating reliable broadcast, (non-blocking) atomic commitment, leader election, and group membership. We give in the following the specifications of these abstractions as well as algorithms to implement them. We do so in a distributed system model where communication channels are reliable but processes can fail by crashing. We shall make use throughout the chapter of a perfect failure detector underlying module and discuss the non-indulgence of these failure-sensitive abstractions through the exercices.

5.1 Terminating Reliable Broadcast

5.1.1 Intuition

As its name indicates, terminating reliable broadcast is a form of reliable broadcast with a termination property.

Consider the case where a given process p_i is known to have the obligation of broadcasting some message to all processes in the system. In other words, p_i is a source of information in the system and all processes must perform some specific processing according to the message m got from p_i. All the remaining processes are thus waiting for p_i's message. If p_i uses a best effort broadcast and does not crash, then its message will be seen by all correct processes. Consider now the case where p_i crashed and some process p_j detects that p_i has crashed without having seen m. Does this means that m was not broadcast? Not really. It is possible that p_i crashed while broadcasting m: some processes may have received m whereas others have not. Process p_j needs to know whether it should keep on waiting for m, or if it can know at some point that m will never be delivered by any process.

At this point, one may think that the problem could be avoided if p_i had used a uniform reliable broadcast primitive to broadcast m. Unfortunately, this is not the case. Consider process p_j in the example above. The use of a uniform reliable broadcast primitive would ensure that, if some other process p_k delivered m, then p_j would eventually deliver m also. However, p_j cannot decide if it should wait for m or not.

The *terminating reliable broadcast (TRB)* abstraction precisely gives to p_j either the message m or some indication F that m will not be delivered. This indication is given in the form of a specific message to the processes: it is however assumed that the indication is not like any other message, i.e., it does not belong to the set of possible messages.

5.1.2 Specifications

The properties of this broadcast abstraction are depicted in Mod.5.1. It is important to notice that the abstraction is defined for a specific originator process, denoted by *src* in Mod.5.1.

5.1.3 Algorithm

We now present an algorithm that implements uniform TRB using three underlying abstractions: a perfect failure detector, a uniform consensus and a best-effort broadcast. The algorithm is given in Alg. 5.1.

The algorithm works by having the source of the message m disseminate m to all correct processes using a best-effort broadcast. Every correct process waits until it gets the message broadcast by the sender process or detects the crash of the originator process. Then all processes run a consensus to agree on whether to deliver m or a failure notification. The processes that got m propose it to consensus and those who detected the crash of the sender, *src*, propose F. The result of the consensus is the value delivered by the TRB algorithm.

An execution of the algorithm is illustrated in Fig. 5.1. Process p_1 crashes while broadcasting m. Therefore p_2 and p_3 get m but p_4 does not. The remaining processes use the consensus module to decide which value must be delivered. In the example of the figure the processes decide to deliver m but F would be also a possible outcome (since p_1 has crashed).

Module:

Name: TerminatingReliableBroadcast (trb).

Events:

Request: ⟨ *trbBroadcast*, src, m ⟩: Used to initiate a terminating reliable broadcast for process *src*.

Indication: ⟨ *trbDeliver*, src, m ⟩: Used to deliver message m broadcast by process src (or F in the case src crashes).

Properties:

TRB1: *Termination:* Every correct process eventually delivers exactly one message.

TRB2: *Validity:* If the sender src is correct and broadcasts a message m, then src eventually delivers m.

TRB3: *Integrity:* If a correct process delivers a message m then either $m = F$ or m was previously broadcast by src.

TRB5: *Uniform Agreement:* If any process delivers a message m, then every correct process eventually delivers m.

Module 5.1 Interface and properties of terminating reliable broadcast.

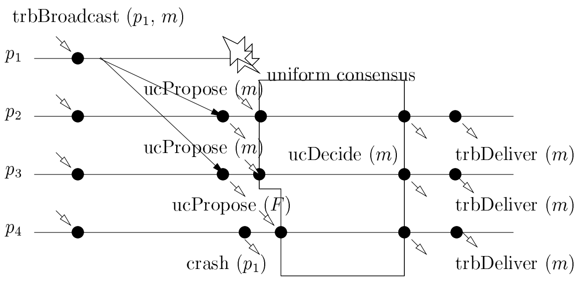

Figure 5.1. Sample execution of terminating reliable broadcast.

Correctness. The *integrity* property of best-effort broadcast and the *validity* property of consensus ensure that if a process trbDelivers a message m, then either m is F or m was trbBroadcast by src. The *no-duplication* property of best-effort broadcast and the *integrity* property of consensus ensure that no process trbDelivers more than one message. The *completeness* property of the failure detector, the *validity* property of best-effort broadcast and the *termination* property of consensus ensure that every correct process eventually trbDelivers a message. The *agreement* property of consensus ensures that of terminating reliable broadcast. Consider now the *validity* property of terminating reliable broadcast. Consider that src does not crash and trbBroadcasts a message $m \neq F$. By the *accuracy* property of the failure detector, no process detects the crash of src. By the *validity* property of best-effort broadcast, every correct process bebDelivers m and proposes m to consensus. By

Algorithm 5.1 A uniform terminating reliable broadcast algorithm.

```
Implements:
    TerminatingReliableBroadcast (trb).

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

upon event ⟨ Init ⟩ do
    proposal := ⊥;
    correct := $\Pi$;

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

upon event ⟨ trbBroadcast, src, m ⟩ do
    if (src = self) do trigger ⟨ bebBroadcast, $m$ ⟩;

upon event ⟨ bebDeliver, $p_i$, $m$ ⟩ do
    proposal := m;

upon (src ∉correct) do
    proposal := $F_{src}$;

upon (proposal ≠ ⊥) do
    trigger ⟨ ucPropose, proposal ⟩;

upon event ⟨ ucDecide, decided ⟩ do
    trigger ⟨ trbDeliver, src, decided ⟩
```

the *termination* property of consensus, all correct processes, including *src*, eventually decide and trbDeliver a message m.

Performance. The algorithm requires the execution of the consensus abstraction. In addition to the cost of consensus, the algorithm exchanges $N-1$ messages and requires one additional communication step (for the initial best-effort broadcast).

Variation. Our TRB specification has a uniform agreement property. As for reliable broadcast, we could specify a regular variant of TRB with a regular agreement property. By using a regular consensus abstraction instead of uniform consensus, we can automatically obtain a regular terminating reliable broadcast abstraction.

5.2 Non-blocking Atomic Commit

5.2.1 Intuition

The *non-blocking atomic commit* (NBAC) abstraction is used to make a set of processes, each representing a data manager, agree on the outcome of a transaction. The outcome is either to *commit* the transaction, say to decide 1, or to *abort* the transaction, say to decide 0. The outcome depends of the initial proposals of the processes. Every process proposes an initial vote for the transaction: 0 or 1. Voting 1 for a process means that the process is willing and able to commit the transaction.

Typically, by voting 1, a data manager process witnesses the absence of any problem during the execution of the transaction. Furthermore, the data manager promises to make the update of the transaction permanent. This in particular means that the process has stored the temporary update of the transaction in stable storage: should it crash and recover, it can install a consistent state including all updates of the committed transaction.

By voting 0, a data manager process vetos the commitment of the transaction. Typically, this can occur if the process cannot commit the transaction for an application-related reason, e.g., not enough money for a bank transfer in a specific node, for a concurrency control reason, e.g., there is a risk of violating serialisability in a database system, or a storage reason, e.g., the disk is full and there is no way to guarantee the persistence of the transaction's updates.

At first glance, the problem looks like consensus: the processes propose 0 or 1 and need to decide on a common final value 0 or 1. There is however a fundamental difference: in consensus, any value decided is valid as long as it was proposed. In the atomic commitment problem, the decision 1 cannot be taken if any of the processes had proposed 0. It is indeed a veto right that is expressed with a 0 vote.

5.2.2 Specifications

NBAC is characterized by the properties listed in Mod.5.2. Without the termination property, the abstraction is simply called *atomic commit* (or *atomic commitment*). Note that NBAC is inherently uniform. In a distributed database system for instance, the very fact that some process has decided to commit a transaction is important, say the process has delivered some cash through an ATM. Even if that process has crashed, its decision is important and other processes should reach the same outcome.

5.2.3 Algorithm

We now present an algorithm that solves NBAC using three underlying abstractions: a perfect failure detector, a consensus and a best-effort broadcast. The algorithm is given in Alg. 5.2.

Module:

Name: Non-Blocking Atomic Commit (nbac).

Events:

Request: ⟨ *nbacPropose*, v ⟩: Used to propose a value for the commit (0 or 1).

Indication: ⟨ *nbacDecide*, v ⟩: Used to indicate the decided value for nbac.

Properties:

NBAC1: Agreement No two processes decide different values.

NBAC2: Integrity No process decides twice.

NBAC3: Abort-Validity 0 can only be decided if some process proposes 0 or crashes.

NBAC4: Commit-Validity 1 can only be decided if no process proposes 0.

NBAC5: Termination Every correct process eventually decides.

Module 5.2 Interfaces and properties of NBAC.

The algorithm works as follows. Every correct process p_i broadcasts its proposal (0 or 1) to all, and waits, for every process p_j, either to get the proposal of p_j or to detect the crash of p_j. If p_i detects the crash of any other process or gets a proposal 0 from any process, then p_i invokes consensus with 0 as its proposal. If p_i gets the proposal 1 from all processes, then p_i invokes consensus with 1 as a proposal. Then the processes decide for NBAC the outcome of consensus.

Correctness. The *agreement* property of NBAC directly follows from that of consensus. The *no-duplication* property of best-effort broadcast and the *integrity* property of consensus ensure that no process nbacDecides twice. The *termination* property of NBAC follows from the *validity* property of best-effort broadcast, the *termination* property of consensus, and the *completeness* property of the failure detector. Consider now the *validity* properties of NBAC. The *commit-validity* property requires that 1 is decided only if all processes propose 1. Assume by contradiction that some process p_i nbacProposes 0 whereas some process p_j nbacDecides 1. By the algorithm, for p_j to nbacDecide 1, it must have decided 1, i.e., through the consensus abstraction. By the *validity* property of consensus, some process p_k must have proposed 1 to the consensus abstraction. By the *validity* property of best-effort broadcast, there are two cases to consider: (1) either p_i crashes before p_k bebDelivers p_i's proposal or (2) p_k bebDelivers p_i's proposal. In both cases, by the algorithm, p_k proposes 0 to consensus: a contradiction. Consider now the *abort-validity* property of NBAC. This property requires that 0 is decided only if some process nbacProposes 0 or crashes. Assume by contradiction that all processes nbacPropose 1 and no process crashes, whereas some process p_i nbacDecides

Algorithm 5.2 Non-blocking atomic commit.

```
Implements:
    NonBlockingAtomicCommit (nbac).

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

upon event ⟨ Init ⟩ do
    delivered := $\emptyset$;
    correct := $\Pi$;
    proposal := 1;

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

upon event ⟨ nbacPropose, $v$ ⟩ do
    trigger ⟨ bebBroadcast, $v$ ⟩;

upon event ⟨ bebDeliver, $p_i$, $v$ ⟩ do
    delivered := delivered $\cup\{p_i\}$ ;
    proposal := proposal * $v$;

upon (correct $\setminus$ delivered = $\emptyset$) do
    if correct $\neq \Pi$ then
        proposal := 0;
    trigger ⟨ ucPropose, proposal ⟩;

upon event ⟨ ucDecide, decided ⟩ do
    trigger ⟨ nbacDecide, decided ⟩
```

0. For p_i to nbacDecide 0, by the *validity* property of consensus, some process p_k must propose 0. By the algorithm and the *accuracy* property of the failure detector, p_k would only propose 0 if some process nbacProposes 0 or crashes: a contradiction.

Performance. The algorithm requires the execution of the consensus abstraction. In addition to the cost of consensus, the algorithm exchanges N^2 messages and requires one additional communication step (for the initial best-effort broadcast).

Variation. One could define a non-uniform variant of NBAC, i.e., by requiring only *agreement* and not *uniform agreement*. However, this abstraction would not be useful in a practical setting to control the termination of a transaction in a distributed database system. A database server is obviously supposed to recover after a crash and even communicate the outcome of the transaction to the outside world before crashing. The very fact that it has committed (or

Module:

Name: LeaderElection (le).

Events:

Indication: $\langle$ *leLeader* $\rangle p_i$: Used to indicate that process p_i is now the leader.

Properties:

LE1: Either there is no correct process, or some correct process is eventually permanently the leader.

LE2: A process p is leader only if all processes in $O(p)$ have crashed.

Module 5.3 Interface and properties of leader election.

aborted) a transaction is important: other processes must nbacDecide the same value.

5.3 Leader Election

5.3.1 Intuition

The leader election abstraction consists in choosing one process to be selected as a unique representative of the group of processes in the system. This abstraction is very useful in a primary-backup replication scheme for instance. Following this scheme, a set of replica processes coordinate their activities to provide the illusion of a fault-tolerant service. Among the set of replica processes, one is chosen as the leader. This leader process, sometimes called primary, is supposed to treat the requests submitted by the client processes, on behalf of the other replicas, called backups. Before a leader returns a reply to a given client, it updates its backups to keep them up to date. If the leader crashes, one of the backups is elected as the new leader, i.e., the new primary.

5.3.2 Specification

We define the leader election abstraction through the properties given in Mod.5.3. Processes are totally ordered according to some function O, which is known to the user of the leader election abstraction, e.g., the clients of a primary-backup replication scheme. This function O associates to every process, those that precede it in the ranking. A process can only become leader if those that precede it have crashed. In a sense, the function represents the royal ordering in a monarchical system. The prince becomes leader if and only if the queen dies. If the prince does, may be his little brother is the next on the list, etc. Typically, we would assume that $\mathrm{O}(p_1) = \emptyset$, $\mathrm{O}(p_2) = \{p_1\}$, $\mathrm{O}(p_3) = \{p_1, o_2\}$, and so forth. The order in this case is $p_1; p_2; p_3; ...$

Algorithm 5.3 Leader election algorithm.

```
Implements:
    LeaderElection (le);

Uses:
    PerfectFailureDetector (P);

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

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

upon event O(self) $\subset$ suspected do
    trigger ⟨ leLeader, leader ⟩;
```

5.3.3 Algorithm

Correctness. Property *LE1* follows from the *completeness* property of the failure detector whereas property *LE2* follows from the *accuracy* property of the failure detector.

Performance. The process of becoming a leader is a local operation. The time to react to a failure and become the new leader depends on the latency of the failure detector.

5.4 Group Membership

5.4.1 Intuition

In the previous sections, our algorithms were required to make decisions based on the information about which processes were operational or crashed. This information is provided by the failure detector module available at each process. However, the output of failure detector modules at different processes is not coordinated. This means that different processes may get notification of failures of other processes in different orders and, in this way, obtain a different perspective of the system evolution. One of the roles of a membership service is to provide consistent information about which processes are correct and which processes have crashed.

Another role of a membership service is to allow new processes to leave and join the set of processes that are participating in the computation, or let old processes voluntarily leave this set. As with failure information, the result of leave and join operations should be provided to correct processes in a consistent way.

Module:

Name: Membership (memb).

Events:

Indication: $\langle$ *membVview*, g, V^i $\rangle$ Used to deliver update membership information in the form of a *view*. The variable g denotes the group id. A view V^i is a tuple (i, M), where i is a unique view identifier and M is the set of processes that belong to the view.

Properties:

Memb1: *Self inclusion:* If a process p installs view $V^i = (i, M_i)$, then $p \in M_i$.

Memb2: *Local Monotonicity:* If a process p installs view $V^j = (j, M_j)$ after installing $V^i = (i, M_i)$, then $j > i$.

Memb3: *Initial view:* Every correct process installs $V^0 = (0, \Pi)$.

Memb4: *Agreement:* If a correct process installs V^i, then every correct process also installs V^i.

Memb5: *Completeness:* If a process p crashes, then eventually every correct process installs $V^i = (i, M_i) : q \notin M_i$.

Memb6: *Accuracy:* If some process installs a view $V^i = (i, M_i) : q \notin M_i$, then q has crashed.

Module 5.4 Interface and properties of a group membership service.

To simplify the presentation, we will consider here just the case of process crashes, i.e., the initial membership of the group is the complete set of processes and subsequent membership changes are solely caused by crashes. Hence, we do not consider explicit join and leave operations.

5.4.2 Specifications

We name the set of processes that participate in the computation a *group*. The current membership of the group is called a *group view*. Each view $V^i = (i, M_i)$ is a tuple that contains a unique view identifier i and a set of member processes M. We consider here a *linear group membership* service, where all correct processes see the same sequence of views: $V^0 = (0, M_0), V^1 = (1, M_1), \ldots$. As we have noted before, the initial view of all processes V^0 includes the complete set of processes Π in the system. A process that delivers a view V^i is said to *install* view V^i. The membership service is characterized by the properties listed in Mod.5.4.

5.4.3 Algorithm

We now present a group membership algorithm based on consensus and a perfect failure detector. The algorithm is depicted in Alg. 5.4. At initialization, each process delivers the initial view with all the processes in the system.

Algorithm 5.4 Group membership properties.

```
Uses:
    UniformConsensus (uc);
    PerfectFailureDetector ($\mathcal{P}$);

upon event ⟨ Init ⟩ do
    current-id := 0;
    current-membership := $\Pi$;
    next-membership := $\Pi$;
    current-view := (current-id, current-membership);
    wait := false;
    trigger ⟨ memView, $g$, current-view ⟩;

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

upon (current-membership $\neq$ next-membership) $\wedge$ ($\neg$ wait) do
    wait := true;
    trigger ⟨ ucPropose, current-id+1, next-membership ⟩;

upon event ⟨ ucDecided, id, memb ⟩ do
    current-id := id;
    current-membership := memb;
    next-membership := current-membership $\cap$ next-membership;
    current-view := (current-id, current-membership);
    wait := false;
    trigger ⟨ membView, $g$, current-view ⟩
```

From that point on, the algorithm remains idle until a process is detected to have crashed. Since different processes may detect crashes in different orders, a new view is not generated immediately. Instead, a consensus is executed to decide which processes are to be included in the next view. The *wait* flag is used to prevent a process to start a new consensus before the previous consensus terminates. When consensus decides, a new view is delivered and the *current-membership* and *next-membership* are updated. Note that a process may install a view containing a process that it already knows to be crashed. In this case it will initiate a new consensus to trigger the installation of another view.

An execution of the membership algorithm is illustrated in Fig. 5.2. In the execution both p_1 and p_2 crash. Process p_3 detects the crash of p_2 and initiates the consensus to define a new view. Process p_4 detects the crash of p_1 and proposes a different view to consensus. As a result of the first consensus, p_1 is excluded from the view. Since p_3 has already detected the crash of p_2, p_3 starts a new consensus to exclude p_2. Eventually, p_4 also detects the crash of p_2 and also participates in the consensus for the third view, that only includes the correct processes.

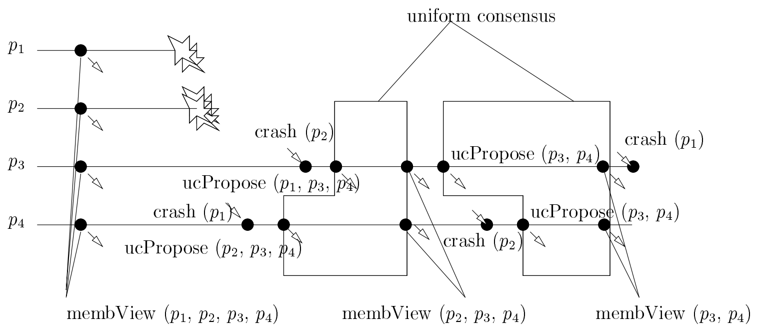

Figure 5.2. Sample execution of the membership algorithm.

Correctness. self inclusion, *local monotonicity*, and *initial view* follow from the algorithm. The *agreement* property follows from consensus. The *completeness* property follows from the *completeness* property of the failure detector and the *accuracy* property follows from the *accuracy* property of the failure detector.

Performance. The algorithm requires at most one consensus execution for each process that crashes.

5.5 Exercices

1. Can we implement TRB with the eventually perfect failure detector $\diamond\mathcal{P}$ if we assume that at least one process can crash?

2. Do we need the perfect failure detector $\mathcal{P}$ to implement TRB (assuming that any number of processes can crash and every process can trbBroadcast messages)?

3. Devise two algorithms that, without consensus, implement weaker specifications of NBAC where we replace the *termination* property with the following ones:

- *(1) weak termination*: let p_i be some process: if p_i does not crash then all correct processes eventually decide;
- *(2) very weak termination*: if no process crashes, then all processes decide.

4. Can we implement NBAC with the eventually perfect failure detector $\diamond\mathcal{P}$ if we assume that at least one process can crash? What if we consider a weaker specification of NBAC where the *agreement* was not required?

5. Do we need the perfect failure detector $\mathcal{P}$ to implement NBAC if we consider a system where at least two processes can crash but a majority is correct?

6. Do we need the perfect failure detector $\mathcal{P}$ to implement NBAC if we assume that at most one process can crash?

7. Consider a specification of leader election where we require that (1) *there cannot be two leaders at the same time* and (2) *either there is no correct process, or some correct process is eventually leader*. Is this specification sound? e.g., would it be useful for a primary-backup replication scheme?

8. What is the difference between the specification of leader election given in the core of the chapter and a specification with the two properties of the previous exercice and the following property: (3) *(stability) a leader remains leader until it crashes.*

9. Do we need the perfect failure detector $\mathcal{P}$ to implement leader election?

5.6 Corrections

1. No. Consider TRB$_i$, i.e., the sender is process p_i. We discuss below why it is impossible to implement TRB$_i$ with $\diamond\mathcal{P}$ if one process can crash. Consider an execution E_1 where process p_i crashes initially and consider some correct process p_j. By the *termination* property of TRB$_i$, there must be a time T at which p_j trbDelivers F_i. Consider an execution E_2 that is similar to E_1 up to time T, except that p_i is correct: p_i's messages are delayed until after time T and the failure detector behaves as in E_1 until after time T. This is possible because the failure detector is only eventually perfect. Up to time T, p_j cannot distinguish E_1 from E_2 and trbDelibevers F_i. By the *agreement* property of TRB$_i$, p_i must trbDeliver F_i as well. By the *termination* property, p_i cannot trbDeliver two messages and will contadict the *validity* property of TRB$_i$.

2. Do we need the perfect failure detector to implement TRB if we assume that any number of processes can crash?

Yes. More precisely, we discuss below that if we have TRB$_i$ abstractions, for every process p_i, and if we consider a model where failures cannot be predicted, then we can *emulate* a perfect failure detector. This means that the perfect failure detector is not only sufficient to solve TRB, but also necessary. The *emulation* idea is simple. Every process trbBroadcasts a series of messages to all processes. Every process p_j that trbDelivers F_i, suspects process p_i. The *strong completeness* property would trivially be satisfied. Consider the *strong accuracy* property (i.e., no process is suspected before it crashes). If p_j trbDelivers F_i, then p_i is faulty. Given that we consider a model where failures cannot be predicted, p_i must have crashed.

3. The idea of the first algorithm is the following. It uses a perfect failure detector. All processes bebBroadcast their proposal to process p_i. This process would collect the proposals from all that it does not suspect and compute the decision: 1 if all processes propose 1 and 0 otherwise, i.e., if some process proposes 0 or is suspected to have crashed. Then p_i bebBroadcasts the decision to all and decide. Any process that bebDelivers the message decides accordingly. If p_i crashes, then all processes are blocked. Of course, the processes can figure out the decision by themselves if p_i crashes after some correct process has decided, or if some correct process decides 0. However, if all correct processes propose 1 and p_i crashes before any correct process, then no correct process can decide.

This algorithm is also called the *Two-Phase Commit (2PC)* algorithm. It implements a variant of atomic commitment that is *blocking*.

The second algorithm is simpler. All processes bebBroadcast their proposals to all. Every process waits from proposals from all. If a process bebDelivers 1 from all it decides 1, otherwise, it decides 0. (This algorithm does not make use of any failure detector.)

4. No. The reason is similar to that of exercise 1. Consider an execution E_1 where all processes are correct and propose 1, except some process p_i which proposes 0 and crashes initially. By the *abort-validity* property, all correct processes decide 0. Let T be the time at which one of these processes, say p_j, decides 0. Consider an execution E_2 that is similar to E_1 except that p_i proposes 1. Process p_j cannot distinguish the two executions (because p_i did not send any message) and decides 0 at time T. Consider now an execution E_3 that is similar to E_2, except that p_i is correct but its messages are all delayed until after time T. The failure detector behaves in E_3 as in E_2: this is possible because it is only eventually perfect. In E_3, p_j decides 0 and violates *commit-validity*: all processes are correct and propose 1.

In this argumentation, the *agreement* property of NBAC was not explicitly needed. This shows that even a specification of NBAC where *agreement* was not needed could not be implemented with an eventually perfect failure detector if some process crashes.

5. Do we need the perfect failure detector to implement NBAC if we assume that a minority of the processes can crash? What if we assume that at most one process can crash? What if we assume that any number of processes can crash?

If we assume that a minority of processes can crash, then the perfect failure detector is not needed. To show that, we exhibit a failure detector that, in a precise sense, is strictly weaker than the perfect failure detector and that helps solving NBAC.

The failure detector in question is denoted by $?P$, and called the *anonymously perfect* perfect failure detector. This failure detector ensures the *strong completess* and *eventual strong accuracy* of an eventually perfect failure detector, plus the following *anonymous detection* property: every correct process suspects outputs a specific value F iff some process has crashed.

Given that we assume a majority of correct processes, then the $?P$ failure detector solves uniform consensus and we can build a consensus module. Now we give the idea of an algorithm that uses $?P$ and a consensus module to solve NBAC.

The idea of the algorithm is the following. All processes bebBroadcast their proposal to all. Every process p_i waits either (1) to bebDeliver 1 from all processes, (2) to bebDeliver 0 from some process, or (3) to output F. In case (1), p_i invokes consensus with 1 as a proposed value. In cases (2) and (3), p_i invokes consensus with 0. Then p_i decides the value output by the consensus module.

Now we discuss in which sense ?P is strictly weaker than P. Assume a system where at least two processes can crash. Consider an execution E_1 where two processes p_i and p_j crash initially and E_2 is an execution where only p_i initially crashes. Let p_k be any correct process. Using ?P, at any time T, process p_k can confuse executions E_1 and E_2 if the messages of p_j are delayed. Indeed, p_k will output F and know that some process has indeed crashed but will not know which one.

Hence, in a system where two processes can crash but a majority is correct, then P is not needed to solve NBAC. There is a failure detector that is strictly weaker and this failure detector solves NBAC.

6. We show below that in a system where at most one process can crash, we can emulate a perfect failure detector if we can solve NBAC. Indeed, the processes go through sequential rounds. In each round, the processes bebBrodcast a message *I-Am-Alive* to all and trigger an instance of NBAC (two instances are distinguished by the round number at which they were triggered). In a given round r, every process waits to decide the outcome of NBAC: if this outcome is 1, then p_i moves to the next round. If the outcome is 0, then p_i waits to bebDeliver $N-1$ messages and suspects the missing message. Clearly, this algorithm emulates the behavior of a perfect failure detector P in a system where at most one process crashes.

7. The specification looks simple but is actually bogus. Indeed, nothing prevents an algorithm from changing leaders all the time: this would comply with the specification. Such a leader election abstraction would be useless, say for a primary-backup replication scheme, because even if a process is leader, it would not know for how long and that would prevent it from treating any request from the client. This is because we do not explicitly handle any notion of time. In this context, to be useful, a leader must be *stable*: once it is elected, it should remain leader until it crashes.

8. A specification with properties (1), (2) and (3) makes more sense but still has an issue: we leave it up to the algorithm that implements the leader election abstraction to choose the leader. In practice, we typically expect the clients of a replicated service to know which process is the first leader, which is the second to be elected if the first has crashed, etc. This is important for instance in failure-free executions where the clients of a replicated service would consider sending their requests directly to the actual leader instead of broadcasting the requests to all, i.e., for optimization issues. Our specification, given in the core of the chapter, is based on the knowledge of an ordering function that the processes should follow in the leader election process. This function is not decided by the algorithm and can be made available to the client of the leader election abstraction.

9. Yes. More precisely, we discuss below that if we have a leader election abstraction, then we can emulate a perfect failure detector. This means that the perfect failure detector is not only sufficient to solve leader election, but also necessary. The *emulation* idea is simple. Every process p_i triggers $N-1$ instances of leader election, each one for a process p_j different from p_i. In instance j, $O(p_j) = \emptyset$ and $O(p_i) = \{p_j\}$, for every $p_j \neq p_i$. Whenever p_i is elected leader in some instance j, p_i accurately detects the crash of p_j.

References

Birman, K. and T. Joseph (1987, February). Reliable Communication in the Presence of Failures. *ACM, Transactions on Computer Systems 5*(1).

C. Dwork, N. L. and L. Stockmeyer (1988). Consensus in the presence of partial synchrony. *Journal of the ACM 35*(2).

Chandra, T. and S. Toueg (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM 43*(2), 225–267.

Fischer, M., N. Lynch, and M. Paterson (1985, April). Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinery 32*(2), 374–382.

Gray, C. and D. Cheriton (1989, December). Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, Litchfield Park, Arizona, pp. 202–210.

Guerraoui, R. (2000, July). Indulgent algorithms. In *Proceedings of the Nineteenth ACM Symposium on Principles of Distributed Computing*, Portland, Oregon, pp. 289–298.

Lamport, L. (1978, July). Time, clocks and the ordering of events in a distributed system. *CACM 21*(7), 558–565.

Lamport, L. (1989). The part-time parliament. Technical Report 49, DEC Systems Research Center. Also published in ACM Transactions on Computer Systems (TOCS), Vol. 16, No. 2, 1998.

Lamport, L., R. Shostak, and M. Pease (1982, July). The byzantine generals problem. *ACM Transactions on Prog. Lang. and Systems 4*(3).

Raynal, M., A. Schiper, and S. Toueg (1991, September). The causal ordering abstraction and a simple way to implement it. *Information processing letters 39*(6), 343–350.

Saltzer, J., D. Reed, and D. Clark (1984, November). End-to-end arguments in system design. *ACM Transactions on Computer Systems 2*(4).

Schneider, F., D. Gries, and R. Schlichting (1984). Fault-tolerant broadcasts. *Science of Computer Programming* (4), 1–15.

References

JL1 lzwarth F., Lenz J. et al. (1998) 1liesdas. Weitere Hinweise zum Layout und LaTeX Code. Springer, Berlin Heidelberg

JL2 lzwarth F., Lenz J. et al. (1998) 1readme. Further Details on Layout and LaTeX code. Springer, Berlin Heidelberg