Technical Report

UCAM-CL-TR-667
ISSN 1476-2986

Number 667

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Minimizing latency of agreement protocols

Piotr Zieliński

June 2006

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted September 2005 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity Hall.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Summary

Maintaining consistency of fault-tolerant distributed systems is notoriously difficult to achieve. It often requires non-trivial agreement abstractions, such as Consensus, Atomic Broadcast, or Atomic Commitment. This thesis investigates implementations of such abstractions in the asynchronous model, extended with unreliable failure detectors or eventual synchrony. The main objective is to develop protocols that minimize the number of communication steps required in failure-free scenarios but remain correct if failures occur. For several agreement problems and their numerous variants, this thesis presents such low-latency algorithms and lower-bound theorems proving their optimality.

The observation that many agreement protocols share the same round-based structure helps to cope with a large number of agreement problems in a uniform way. One of the main contributions of this thesis is *Optimistically Terminating Consensus* (OTC) – a new lightweight agreement abstraction that formalizes the notion of a round. It is used to provide simple modular solutions to a large variety of agreement problems, including Consensus, Atomic Commitment, and Interactive Consistency. The OTC abstraction tolerates malicious participants and has no latency overhead; agreement protocols constructed in the OTC framework require no more communication steps than their ad-hoc counterparts.

The attractiveness of this approach lies in the fact that the correctness of OTC algorithms can be tested automatically. A theory developed in this thesis allows us to quickly evaluate OTC algorithm candidates without the time-consuming examination of their entire state space. This technique is then used to scan the space of possible solutions in order to automatically discover new low-latency OTC algorithms. From these, one can now easily obtain new implementations of Consensus and similar agreement problems such as Atomic Commitment or Interactive Consistency.

Because of its continuous nature, Atomic Broadcast is considered separately from other agreement abstractions. I first show that no algorithm can guarantee a latency of less than three communication steps in all failure-free scenarios. Then, I present new Atomic Broadcast algorithms that achieve the two-step latency in some special cases, while still guaranteeing three steps for other failure-free scenarios. The special cases considered here are: Optimistic Atomic Broadcast, (Optimistic) Generic Broadcast, and closed-group Atomic Broadcast. For each of these, I present an appropriate algorithm and prove its latency to be optimal.

Acknowledgements

First, I would like to thank my family for always being supportive and helpful in all my life decisions, which eventually led me to start my PhD research in Cambridge. This would also not have been possible without the guidance of my B.Eng. and M.Sc. supervisors, Prof. Jerzy Nawrocki and Prof. Jerzy Brzeziński, who introduced me to the area of distributed computing and encouraged my scientific interests in this direction.

I am indebted to my current supervisor Dr. Markus Kuhn for always having time to discuss my research and for proof-reading an earlier version of this thesis. I would also like to thank other members of the Computer Lab Security Group for many challenging research projects, which served as enjoyable breaks from my PhD research.

I wish thank all those who enriched by social life in Cambridge, especially my housemates, Salomé, Ria, Bernhard, and others, for all these memorable moments in the last four years. I would also like to thank my lab colleagues for all not strictly academic undertakings that contributed to a pleasantly informal working atmosphere.

I spent a summer as an intern at Microsoft Research Cambridge. I would like to thank Mike Roe and Tuomas Aura for working with me on an interesting and challenging project, which gave me a new perspective on my own research.

Generous financial support for this work was provided by the Thaddeus Mann scholarship from Trinity Hall, a scholarship from Cambridge Overseas Trust, and an ORS award from Universities UK. The Computer Laboratory and Trinity Hall also covered my conference and travel expenses.

Contents

List of symbols

n	number of acceptors
f	maximum number of faulty acceptors
m	maximum number of malicious acceptors
q	maximum number of faulty acceptors in Optimistic Termination (q, k)
d	maximum (supremum) message latency between correct processes
p_i	proposers
a_i	acceptors
l_i	learners
c_i	coordinators
i, j	indices for rounds, processes, etc.
k	number of communication steps, number of acceptors in a sequence
t	time, timeframe
δ	length of a timeframe
x, y	proposed values
m	message (in Chapter 5)
A	set of all acceptors, $A = \{a_1, a_2, \ldots, a_n\}$
F	set of all faulty acceptors (unknown to the application)
M	set of all malicious acceptors (unknown to the application)
D	decision rule
T	termination rule
$\mathcal{F}$	set of possible sets F of faulty acceptors
$\mathcal{M}$	set of possible sets M of malicious acceptors
$\mathcal{D}$	set of decision rules D
$\mathcal{T}$	set of termination rules T
α	sequence of acceptors, $\alpha = e_1 e_2 \ldots e_k$
αX	set of all sequences of acceptors ending with some $a \in X$
e_i	symbolic acceptors in events $\langle x : e_1 e_2 \ldots e_k \rangle$
S	state, a set of events $\langle x : e_1 e_2 \ldots e_k \rangle$
$S(x)$	set of all events from S of the form $\langle x : e_1 e_2 \ldots e_k \rangle$ with the given x

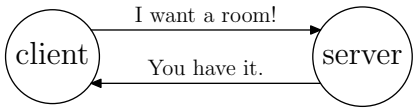
$\emptyset$	the empty set
ε	the empty sequence
$\bullet$	denotes "any" number of steps, used in Optimistic Termination $(q, \bullet)$
$\perp$	an empty variable (Onecast) or an artificial message (Generic Broadcast)
$\rightarrow$	partial order on conflicting messages
$\leftarrow$	variable assignment, for example, $x \leftarrow 5$

Chapter 1

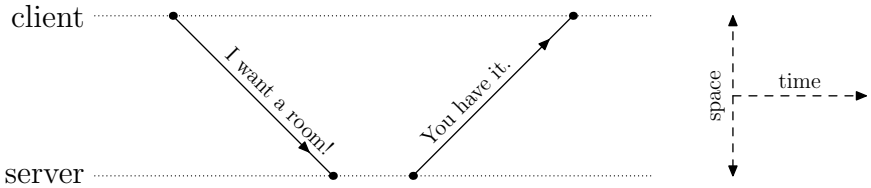
Introduction

A number of real-world processes can be modelled as interactions between two kinds of participants: *clients* who issue requests and *servers* who fulfil them. Examples range from human-human interaction, such as an investor ordering his trust fund to sell some shares, to computer-computer interactions, for example an operating system automatically requesting the latest security patches from the vendor's Internet site. In this thesis, we consider the latter case, in which one computer (the client), possibly at a human's request, uses a network to communicate with another computer (the server) in order to obtain some service.

As an example, take a hotel room booking system, in which customers (the clients) can access the hotel webpage (the server) to make reservations. In order to book a room, a client sends a request to the server, who replies with a confirmation message:

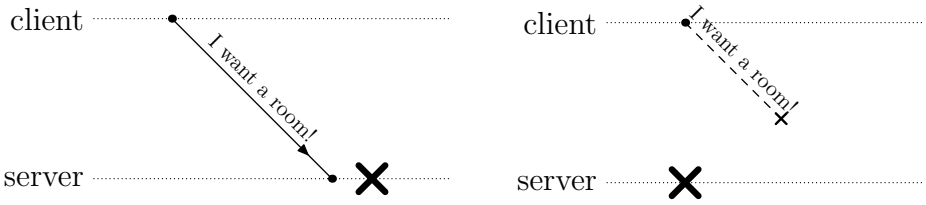


The same message exchange can be represented graphically as



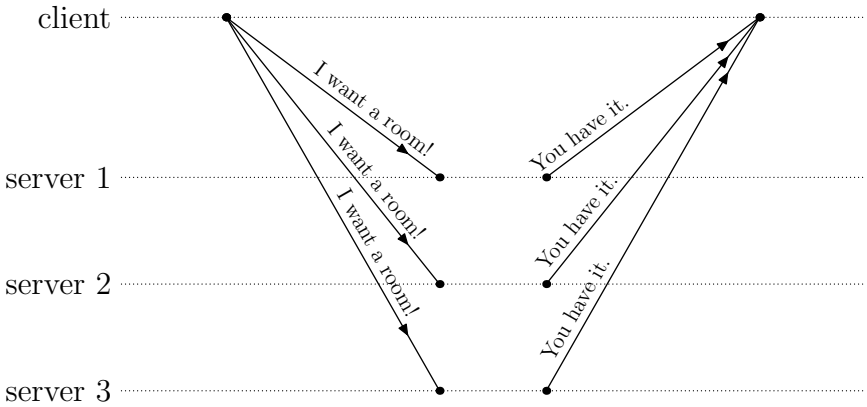
In this diagram, the horizontal axis represents time and the vertical one space. Both the client and the server, which we will collectively call *processes*, are represented by dotted horizontal lines, because they exists in a single point in space but span over time. Events, such as sending or receiving a message, happen at a particular process at a particular time, so they are represented as dots. Each message is depicted as an arrow from the event of sending the message to the event of receiving it.

In our example, the client-server model works well; the client sends a message to the server, and after some time receives the reply. The problem with this approach is that it introduces a single point of failure. If the server crashes, no client can access the system:

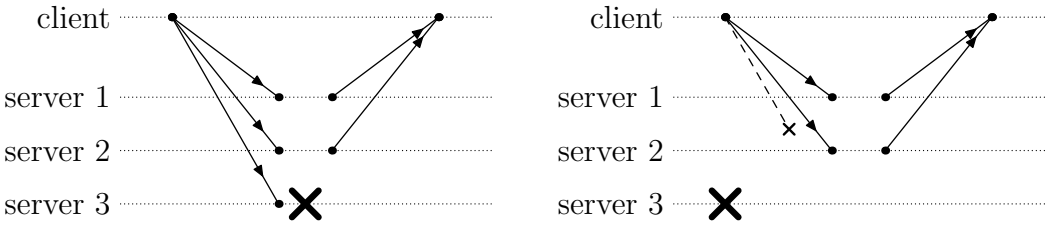


In the first example, the server crashes just after receiving the client's request, but before sending the reply. In the other example, the crash occurs even before the client request arrives at the server, so the client's request simply gets lost. In both cases, the client does not receive any reply.

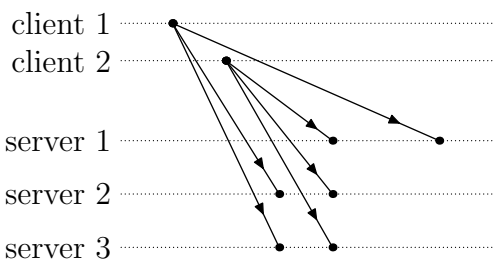
The standard solution to this problem is *replication*, which avoids a single point of failure by replacing a single server with many identical replicas:



This way, a failure of an individual server will not block the system; as long as some servers remain operational, the client will receive the reply:



With replication, however, maintaining consistency of the system becomes an issue. Client requests may arrive at the servers in different orders:



If both clients try to book the same room, who will get it? If rooms are allocated on a first-come first-served basis, server 1 will assign the room to client 2, whereas the other servers will assign the room to client 1. The clients will become confused by inconsistent

responses. Not only that, the supposedly identical states of the servers will start to differ, thereby making the whole system enter an inconsistent state.

To maintain consistency, clients need a broadcasting algorithm that ensures that all servers receive their requests in the same order. This problem is known in the literature as *Atomic Broadcast* [25]. It is relatively easy to solve in systems in which failures do not occur, but becomes significantly more difficult once the mere possibility of failure is introduced to the model.

One of the general approaches to implementing Atomic Broadcast consists of two phases: clients broadcasting their requests to the servers and the servers agreeing on the order in which the requests will be delivered [26]. This agreement phase is an interesting problem in itself, which is called *Consensus*. In this abstraction, each server issues a single proposal, for example a number or a sequence of requests, and all of them eventually agree on one of these proposals. Consensus is a fundamental problem in distributed computing because it can be used to implement many other abstractions such as Atomic Broadcast or Atomic Commitment [58]. In fact, Consensus is universal in the sense that any sequential object, such as a single server, can be implemented in a distributed way using Consensus [62].

As with Atomic Broadcast, the possibility of failures makes the Consensus problem far from trivial. One possible method would be that the first server broadcasts its proposal and imposes it on others. However, this simple solution fails when the first server crashes. One can try to avoid this problem by having a second server take over if the first experiences difficulties, but this creates many new problems. What if the main server managed to send its proposal only to some servers, but not all? How long should a server wait until it can assume that the main server has crashed? What if different servers have different opinions about whether the main server crashed or not? These and similar questions make Consensus and other agreement problems interesting.

Goal of this thesis

This thesis investigates efficient implementations of agreement problems in distributed systems. Efficiency can be measured in a number of ways: as processor usage, memory usage, network load, the number of messages transmitted, etc. In this work, we focus on *latency* – the time that passes from the start to the end of the algorithm. For each of the considered agreement abstractions, we provide a number of implementations and examine their latencies in various scenarios. We also present lower bound theorems that prove that the latencies achieved by our algorithms cannot be improved.

There is a trade-off between optimizing algorithms for the typical case (no failures) and the worst case (many failures). Since failures are rare, we focus on minimizing the latency in runs without failures. In other runs, the latency of our algorithms might not be

optimal. Nevertheless, we guarantee the correctness of our algorithms in all runs allowed by the model, including those with failures.

To cope with a large number of agreement problems in a uniform way, Chapter 2 introduces a new lightweight agreement abstraction which we call *Optimistically Terminating Consensus* (OTC). It tolerates malicious participants and has no latency overhead; the latencies of agreement protocols constructed in the OTC framework do not exceed those of their ad-hoc counterparts. Chapter 3 presents a technique for automatic verification and discovery of new OTC algorithms. In Chapter 4, we show how to use both manually and automatically generated OTC algorithms to provide simple modular solutions to a large variety of agreement problems, including Consensus, Atomic Commitment, and Interactive Consistency. Latency-optimal Atomic Broadcast protocols are discussed separately in Chapter 5.

1.1 System model

This section gives more precise definitions of the discussed concepts. We consider a distributed system consisting of a certain number of interconnected processing units, called *processes*. Processes communicate by sending and receiving messages using communication channels [58]. In the real world, processes correspond to computers and communication channels correspond to network connections. It is possible to have several processes running on the same machine; in this case, some communication channels will be local inter-process communication channels provided by the operating system.

1.1.1 Processes

Processes can be thought of as programs running on individual computers. They are specified as a collection of parallel *tasks*. As an example, consider a simple algorithm that equips all processes with two primitives: *bcast*(m) to broadcast a message m, and *deliver*(m) to deliver it to the local user. Each process p runs two parallel tasks:

```
1  task broadcasting at process p is
2    loop forever
3      wait for bcast(m) for some m
4      for all processes q do
5        send m to process q

6  task delivery at process p is
7    loop forever
8      wait for receive(m)
9      deliver(m)
```

The *broadcasting* task contains an infinite loop, whose each iteration waits for a message to be broadcast, and then sends it to all processes, including itself. Each iteration of the infinite loop in the *delivery* task first waits for a message m, and then delivers it to the local user by executing *deliver*(m).

A process can execute many tasks at the same time. However, each task is executed atomically, without interruption, until a **wait** instruction is encountered. At that point, the control can be transferred to another task.

The **wait** instruction comes in two variants: "**wait until** *condition*" and "**wait for** *event*", which suspend the current task until the *condition* holds or the *event* has occurred, respectively. Each occurrence of "**wait for** *event*" ignores the events that it has already used. As a result, the above delivery code will deliver each message the same number of times it received it.

To simplify the notation, we introduce a construct "**when** *event* **do** *body*" as an abbreviation for

```
1  task handle event is
2    loop forever
3      wait for event
4      execute body as a newly created, independent task
```

Line 4 executes *body* as another task, so the original task does not wait until *event* has been handled by *body*.

If a condition is specified in a place where an event is expected, we assume that the event occurs whenever the condition becomes true. With this assumption we can rewrite the broadcast algorithm as

```
1  when bcast(m) at process p do
2    for all processes q do
3      send m to process q

4  when process p received m do
5    deliver(m)
```

Note that, in this implementation, messages are delivered by separate and independent tasks, so the orders of their reception and delivery might differ.

Failures

For various reasons, not all processes behave according to the specification. Some might crash due to hardware errors and stop operating, others might have been subverted and might execute a program completely different from the original one. In our model, we divide processes into two groups: (i) *correct* processes, which behave according to the specification, and (ii) *faulty* processes, which do not. The latter group is subsequently divided into two subgroups: (i) *non-maliciously faulty* processes, which behave correctly but stop operating (crash) at some point, and (ii) *maliciously faulty* processes, which can execute arbitrary code. Processes that are correct or non-maliciously faulty are called *honest*. Our classification is summarized in the table below:

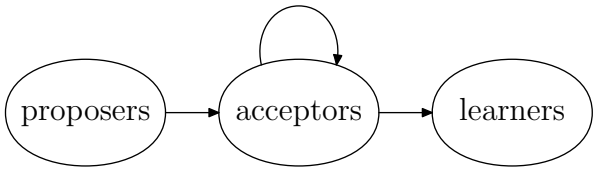
correctness	honesty	behaviour
correct	honest	according to the specification
faulty	honest	according to the specification until it stops
faulty	malicious	arbitrary

Honest processes do not know whether they are correct or not. In other words, they cannot predict whether or when they will crash.

The model that assumes all processes are honest is called the *crash-stop model*, as opposed to the *Byzantine model*, which allows malicious processes. These two models are sometimes referred to as *honest settings* and *malicious settings*, respectively.

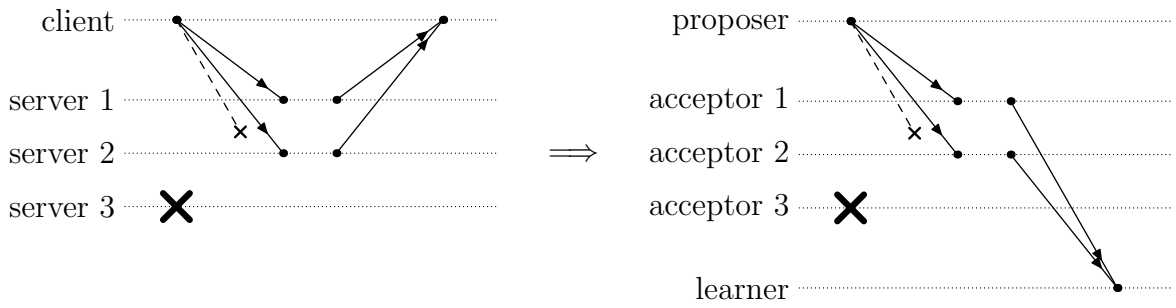
Proposers, acceptors, and learners

In our model, the set of processes is divided into three possibly overlapping groups: *proposers*, *acceptors*, and *learners*. This division was originally proposed by Lamport [76] for Consensus; we generalize it to be problem-independent and defined in terms of message sending capabilities. We assume that proposers can send messages to acceptors, who can send messages to both themselves and the learners.



Acceptors can both send and receive messages from other acceptors, so our definition requires each acceptor to be a proposer and a learner, but not necessarily vice versa.

The proposer-acceptor-learner model can be thought of as a reformulation of the client-server model, with proposers corresponding to clients sending requests, acceptors corresponding to servers receiving requests and sending replies, and learners corresponding to clients receiving replies:



By forbidding direct communication between proposers and learners in general, our model separates these two roles of a client. As in publish-subscribe systems [38], one client (proposer, publisher) sends a message, so that other clients (learners, subscribers)

can deliver it. Acceptors acting as proposers and learners corresponds to the servers being clients in their own system.

In typical applications, clients can come and go, whereas servers are more permanent. To reflect this, we do not impose any restrictions on the number of proposers or learners in the system, nor on the type and number of faults they experience. On the other hand, the number n of acceptors is fixed, we denote these by a_1, a_2, ..., a_n. We assume that at most f acceptors are faulty, out of which at most m are malicious. Note that these numbers denote the *maximum* number of faults allowed by the model. In a particular run, the number of faults can be smaller or even zero.

In our model, learners have no outgoing communication channels, so they cannot affect the rest of the system. Therefore, without loss of generality, we can assume that all learners are honest [76].

1.1.2 Channels

Processes communicate by sending messages through the underlying network. We model this by having pairs of processes connected using dedicated uni-directional *communication channels*. The process model from Section 1.1.1 implies that we need only channels that connect proposers to acceptors, acceptors to acceptors, and acceptors to learners. Each such channel connects two processes known as the *sender* and the *receiver*. The sender can send a message m by invoking $send(m)$. When the receiver receives message m, action $receive(m)$ is invoked.

We assume *asynchronous reliable channels*. This means that all messages from a correct process to a correct process will *eventually* reach their destination (reliability), but there is no upper bound on message transmission time (asynchrony). We assume that channels do not create or modify messages. Formally, we require [57]:

> **No Creation.** If a process q receives message m, then some process p sent m.
>
> **Reliability.** If a correct process p sends a message m to a correct process q, then q will eventually receive m.

We allow channels to duplicate messages, that is, processes can receive the same message twice or more times.

Asynchrony

Our system model is asynchronous, which means that there are no bounds on message transmission times or process speeds. This weak assumption allows us to model a large variety of systems, in which messages or processes can occasionally experience delays. As a consequence, processes cannot use time to synchronize their actions, which means that

the correctness of algorithms developed for the asynchronous model is not prone to timing violations.

Reliability

Our channels are reliable, that is, all messages between correct processes are eventually received. Several reliability conditions have been proposed in the literature [9, 57, 85]: unreliable channels, best-effort channels, stubborn channels, and reliable channels. Among these, the reliable channels provide the strongest guarantees, which makes algorithms designed for this model simpler than those designed for others. Basu et al. [9] showed that reliable channels can be emulated using the (weakest) unreliable channels by periodic retransmission of lost messages. Since this emulation does not incur any additional latency in runs without failures [9], latency-optimal algorithms for reliable channels remain so in weaker communication models.

Reliability is not the strongest channel semantics. For example, *uniformly* reliable channels [9, 57] guarantee correct processes to eventually receive all messages, even those sent by faulty processes. These channels can also be emulated with unreliable ones, but only with a significant latency overhead [9]. For this reason, they are not useful for designing latency-optimal protocols.

Asynchronous reliable channels, which we use here, do not guarantee that messages are received in the same order as they are sent. Stronger guarantees are possible: *FIFO* channels preserve the order of messages sent between a given pair of processes, *causal* channels deliver causally related messages in order. Both semantics can be implemented on top of reliable channels without latency overhead [93].

1.1.3 Features not considered

Many aspects of distributed computing are not discussed in this thesis, but have received a considerable amount of attention in the literature. Below we list three of them:

- **Dynamic groups.** Traditionally, agreement problems have been considered in a model with a fixed number n of processes. Lamport [76] relaxed this condition by dividing processes into proposers, acceptors, and learners, and fixing only the number n of acceptors. Group communication systems [105] went even further by waiving this restriction altogether. In such systems, any process can dynamically join and leave the current group of processes.

- **Recovery.** In our model, when an honest process crashes, it stops all processing forever. In the crash-recovery model [4, 64], crashed processes can eventually resume operating. This is different from a process just being very slow for a while because the recovered processes lose all of their state except for the part of it stored in *stable*

storage such as disks. The main challenge in designing agreement protocols for this model is minimizing the use of stable storage [4, 10, 58, 64, 110].

- **Synchrony.** A significant amount of work on agreement problems has been done in the *synchronous model* [31], in which all messages take exactly one unit of time to reach their destination. A similar *semi-synchronous model* [31] assumes a *known* upper bound on message transmission times. Both models assume that the time constraints always hold, which makes the safety of the algorithms designed for such models susceptible to timing violations. For this reason, our model makes no such timing assumptions. In this sense, the (semi-)synchronous model is stronger than ours, which means that algorithms designed for our model remain correct in the (semi-)synchronous one.

1.2 Consensus

In the Consensus problem, informally introduced in the beginning of this chapter, processes issue proposals and are supposed to reach a common decision. This section will give a formal definition of this abstraction.

Although our system model distinguishes between proposers, acceptors, and learners, most Consensus algorithms presented in the literature do not make this distinction, calling all participants simply "processes" [58]. Differentiating between these three groups of processes, first suggested by Lamport [73], makes the model directly applicable to client-server abstractions, such as Atomic Broadcast.

In this thesis, we define Consensus in terms of two groups of processes: acceptors and learners (proposers do not participate). Each correct acceptor a_i proposes a single value x_i, and all correct learners have to eventually decide on one common value x. Formally, Consensus provides processes with two primitives: an action *propose* available to acceptors, and a predicate *decision* available to learners. When an acceptor a_i executes $propose(x_i)$, we say that "a_i proposes x_i". Similarly, when predicate $decision(x)$ holds at some learner, we say that learner has decided on x. Assuming that honest acceptors propose at most one value, the Consensus problem is defined by the following three properties:

Validity. If all acceptors are honest and $decision(x)$ holds at some learner, then some acceptor proposed x.

Agreement. There is at most one x for which $decision(x)$ holds at some learner.

Termination. If all correct acceptors executed *propose*, then all correct learners will eventually decide.

The Validity property ensures that each decision value has been indeed proposed by some acceptor. This precludes some useless algorithms, such as the one in which all learners decide on 1701 regardless of what values have been proposed. Note that malicious acceptors can propose one value and then behave as if they had proposed another one. Since such behaviour is undetectable, the Validity property must assume that all acceptors are honest. Section 4.1 will discuss this property in more detail.

The Agreement property is the one that gives the Consensus problem its name. It requires that no two learners decide on different values. Some learners might not decide at all; there is nothing to prevent faulty learners from crashing at the very beginning. However, if a learner decides, it has to decide on the same value as other learners, whether it is correct or not. Recall that learners are honest by definition.

The Consensus abstraction considered here is *uniform* [46], which means that Agreement holds for all learners, not only the correct ones. A non-uniform Consensus allows faulty learners to decide on different values than the correct ones. All abstractions considered is this thesis are uniform, unless explicitly stated otherwise.

The Validity and Agreement properties of Consensus guarantee only safety; they merely prevent learners from deciding on "bad" values (those not proposed or different from other decisions). In particular, algorithms in which no learner ever decides satisfy these properties. In order to preclude such algorithms, we need the Termination property, which ensures that if all correct acceptors have proposed, then all correct learners will eventually decide.

For a more detailed introduction to Consensus and other agreement problems, see the tutorials by Guerraoui et al. [58] and Raynal [108].

1.2.1 Safety and liveness properties

Properties of distributed algorithms can be classified into two groups: *safety properties* and *liveness properties*. Safety properties are those that prevent the algorithm from reaching an erroneous state, such as deciding on two different values (Agreement) or on a value that has not been proposed (Validity). Liveness properties ensure that the system will eventually be in a good state, for example, all correct learners will eventually decide (Termination). More precisely [20], if a *safety* property does not hold at some point in time, then it will never hold, no matter what happens (once a wrong decision has been made, it cannot be undone). On the other hand, at any point in time, no matter what happened up to that point, it is still possible for a *liveness* property to hold (a correct process can always decide).

Is every property either a safety or a liveness property? No, it is not. For example, "Agreement and Termination" treated as a single property is neither. It can be shown, however, that every property is an intersection of a safety and a liveness property [6]. In

this case, "Agreement and Termination" can be decomposed into Agreement (safety) and Termination (liveness). For this reason, distributed abstractions are best presented in a canonical form, with each property being either a safety or a liveness property.

The main reason for separating safety and liveness properties of distributed abstractions is that the former are generally considered more important. This is because a violation of a safety property is by definition final. For example, if two learners decide on different values, nothing can be done to remedy the disagreement. Liveness properties, on the other hand, are never violated in a finite execution; if a learner has not decided yet, it might still decide in the future.

1.3 Consensus unsolvable in asynchronous systems

If one distinguished acceptor, say a_1, is guaranteed to be correct, Consensus can easily be implemented by a_1 broadcasting its proposal and the learners adopting it as the decision. This algorithm is correct because the decision value has been proposed by some acceptor, namely a_1 (Validity), it is the same at every learner (Agreement), and every correct learner will eventually receive a_1's proposal and decide on it (Termination).

If acceptor a_1 fails and does not broadcast its proposal, the above algorithm no longer guarantees Termination. It is, of course, possible to design more sophisticated algorithms which would decide in *some* runs with a_1 being faulty. Is it, however, possible to guarantee Termination in *all* such scenarios? No, it is not. Fischer, Lynch, and Paterson [40] proved that there is no Consensus algorithm that would tolerate all runs with even one faulty acceptor. Intuitively, this results from the fact that it is impossible to safely distinguish a crashed process from a very slow process or a process with which the communication is very slow [58]. Moreover, this impossibility is not specific to Consensus; it applies to all non-trivial agreement problems such as Interactive Consistency [97], Atomic Commitment [48], or Atomic Broadcast [26]. Atomic Broadcast is actually equivalent to Consensus; if one problem is solvable in a given model, then so is the other. Therefore, all Consensus solvability discussions that follow will apply to Atomic Broadcast as well.

It is important to understand the Consensus impossibility result correctly [54]. It states that no algorithm can satisfy *all* three Consensus properties (Validity, Agreement, and Termination) at the same time in *all* runs with at most one faulty acceptor. This theorem does not prevent us from designing algorithms that sometimes fail to satisfy one of these properties. In particular, we can design algorithms that are always safe (satisfy Validity and Agreement), but fail to decide in some special runs with failures. These special runs, although allowed by the asynchronous model, occur rarely in practice; they correspond, for example, to the network *permanently* failing to meet its timeliness even for a short period of time. Adding some small realistic extensions can eliminate such runs from the asynchronous model, making Consensus solvable. Numerous such extensions have been

proposed: failure detectors [16], eventual synchrony [37], partial synchrony [37], timed asynchrony [24], weak ordering oracles [104], and randomization [7].

In this thesis, we limit our attention to *safe* extensions, which do not introduce any new *safety* assumptions to the asynchronous system model. In other words, we consider only those extensions that add only *liveness* properties to the asynchronous model. As a result, even if those additional properties do not hold, the safety of algorithms relying on them cannot be jeopardized [46]. The next two sections will briefly describe two such approaches: eventual synchrony and unreliable failure detectors.

1.3.1 Eventual synchrony

One way of making Consensus solvable in asynchronous systems is by adding some assumptions about message transmission times. Different assumptions result in different levels of synchrony in the system [26]. For example, assuming a *known* upper bound on the message transmission times that always holds makes the system virtually synchronous. This extension is too strong, because it adds a *safety* property to the system. On the other hand, assuming no bounds on message transmission times corresponds to the asynchronous system, which is too weak because Consensus is not solvable.

The *eventually synchronous* model [37] lies between these two extremes. This model assumes the existence of an *unknown* upper bound on message transmission times between correct processes. This is a *liveness* assumption because it cannot be violated in a run with finitely many messages, which means that the eventual synchrony extension is safe. The only situation in which no upper bound exists is when some message transmission times keep increasing without any limit.

1.3.2 Unreliable failure detectors

Dolev et al. [31] investigated a number of different timing assumptions that allow for solvability of Consensus, and presented an algorithm for each of them. Since Consensus algorithms use timing assumptions only to guarantee Termination in case of failures, the Consensus algorithms for different timing models are rather similar. To avoid designing a new Consensus algorithm for every new timing assumption, Chandra et al. [17] proposed a way of encapsulating complicated timing assumptions into much simpler objects known as *failure detectors*.

Failure detectors hide most details about timing assumptions, and present the application with the only information it really needs: whether a particular process is suspected to have crashed or not. This simplifies Consensus algorithms, because all failure-detecting work is done inside the failure-detector abstraction. Moreover, failure detectors hide the timing details from the application, which means that a single Consensus algorithm can now work with different timing assumptions. Finally, the algorithm designer can forget

about time; failure detectors are abstract objects whose properties are defined independently of time.

The failure detectors considered in this thesis are unreliable [46], which means that they can make mistakes. For an arbitrary long period of time, they can report crashed processes as correct and vice versa, but *eventually* their output must reflect the reality. Note that this is a liveness property; unreliable failure detectors are *safe* in the sense that they do not introduce any new safety properties to the model.

Originally, failure detectors [16] were defined in a model with a fixed number n of processes, which in this thesis correspond to acceptors. Therefore, in our model, failure detectors are defined for acceptors only. They come in two variants:

- **Crash detectors.** These detectors [18] provide each acceptor with a *set* of acceptors whom they suspect to have crashed. Different types of detectors have different properties. One of the most commonly used detector $\Diamond S$ [17] guarantees:

  Strong Completeness. Eventually every faulty acceptor will be permanently suspected by every correct acceptor.

  Eventual Weak Accuracy. Eventually some correct acceptor will never be suspected by any correct acceptor.

  In other words, $\Diamond S$ guarantees that eventually all faulty acceptors will be suspected and at least one correct acceptor will not.

- **Leader oracles.** These detectors [18, 82] output a single acceptor that they consider correct, also known as the *leader*. The most commonly used detector Ω [18] guarantees:

  Eventual Agreement. Eventually, the failure detector will output the same correct acceptor at all correct acceptors.

  In other words, Ω guarantees that eventually all correct acceptors will agree on the same correct leader.

Both $\Diamond S$ and Ω are the weakest failure detectors in their classes that make Consensus and Atomic Broadcast solvable [17, 18]. All three properties listed above are liveness properties, so the failure detectors $\Diamond S$ and Ω are safe.

Problems other than Consensus and Atomic Broadcast often require other failure detectors: P is necessary for Interactive Consistency [16, 61, 97], $?P$ and Ψ for Atomic Commitment [47, 50], Σ to implement a register [28], etc. For more information on failure detectors see [29, 109].

1.3.3 Comparison

We have presented two extensions of the asynchronous system model: eventual synchrony and unreliable failure detectors. Both of them are safe yet strong enough to ensure implementability of Consensus and Atomic Broadcast. As explained in the previous section, failure detectors are more elegant because they provide the application with just enough information to implement Consensus, while hiding all irrelevant timing details.

The main problem with failure detectors is their implementability. In the eventually synchronous crash-stop model, unreliable failure detectors can be easily implemented using timeouts [16, 58]. This is possible because a crashed process stops *all* its activities with respect to *all* processes [34]. In the Byzantine model, however, a malicious process can send no messages relevant to the algorithm, yet avoid being flagged as crashed by sending other, irrelevant messages. As a result, traditional failure detectors are not implementable in any model that allows malicious processes [34], regardless of the timing assumptions.

Several failure detectors have been proposed for the Byzantine model [34, 70, 86]. As explained above, it is impossible to detect *all* kinds of malicious behaviour, so Byzantine failure detectors aim at detecting special forms of malicious behaviour, called *quietness* [86] or *muteness* [34]. Other kinds of failures, such as sending syntactically correct but semantically invalid or conflicting messages, remain undetected.

To sum up, the elegance and generality of the failure detector approach makes them attractive for systems in which they can be implemented, that is, those without malicious processes. In this thesis, we will prefer using failure detectors in the crash-stop model, and use eventual synchrony in malicious settings.

1.3.4 Well-behaved runs

Even with unreliable failure detectors or eventual synchrony, the asynchronous system model allows many runs that would almost never occur in practice. Examples include all acceptors crashing, messages taking years to reach their destinations, failure detectors behaving randomly for months, etc. As a result, some algorithm properties, such as low latency, cannot be guaranteed for *all* runs allowed by the model. In such cases, we will have to limit our attention to a class of well-behaved runs that occur most often in practice.

We formalize the idea of well-behaved runs by introducing the notion of timeliness. Informally, a run is *timely* if it is similar to a synchronous one. In the failure detector model, this means that correct acceptors are never suspected. In the eventual synchrony model, a run is timely if the upper bound on message transmission time between correct processes is *sufficiently small*. What "sufficiently small" means depends on the algorithm.

In practice, this bound must be significantly smaller than any timeout values used by the algorithm.

In any model, as part of the definition of a timely run, we assume that local computations are instantaneous and incur no delays. If this is not the case, these delays can be modelled as part of the latency of messages that started these computations.

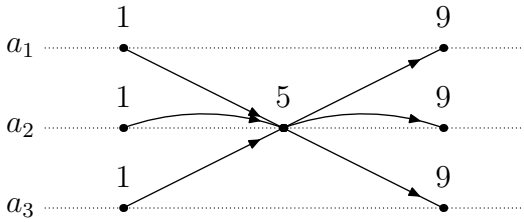
In any model, a run is *good* iff it is timely and all acceptors are correct.

1.4 Latency

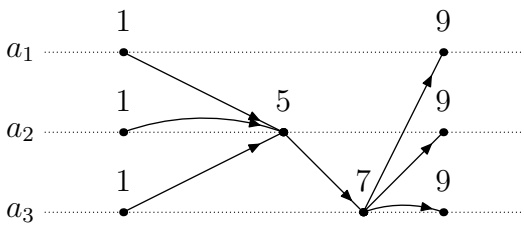
There are many parameters that can be used for evaluating distributed algorithms, such as speed, memory usage, or network usage. In this thesis, we concentrate on speed, or to be more precise, on the latency of the algorithm caused by message delays. We ignore other parameters such as local computation time, memory usage, or the number of messages transmitted. In measuring latency, we limit ourselves to "well-behaved" runs. However, our main concern is always the correctness in *all* runs; we do not even consider algorithms that can be unsafe or not live.

We define the *latency* of an algorithm *run* as the time that passes from the beginning of the run to its end. For example, for Consensus, we measure the time interval from the point when all correct acceptors proposed to the point when all correct learners decided. We will often measure latency in terms of communication steps; for this purpose we define one *communication step* as the maximum (supremum) message transmission time d between correct processes in a particular run. As the asynchronous model does not impose any bounds on the message transmission time, some runs will have $d = \infty$.

For example, the run



starts at time 1 and finishes at 9, so its latency is 8. The longest transmission time is 4, so the run takes $8/4 = 2$ communication steps, which is consistent with our intuition. However, according to our definition, run

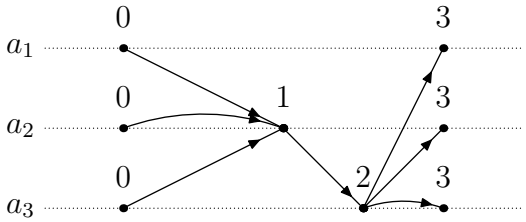


has the same parameters, so it also takes two communication steps. This is no longer consistent with the intuitive number of steps, which is three in this case.

To solve this paradox, we will generalize the notion of time. A *time metric* is a function t from events to real numbers, such that if an event e causally precedes e', then $t(e) \leq t(e')$. Examples of time metrics include real time and the logical time introduced by Lamport [78]. The notions of latency and communication steps clearly depend on the time metric used. We define the number of communication steps required by the algorithm to be the maximum (supremum) taken over all time metrics.

From now on, we assume that any statement referring to these notions must be true in *every* time metric, unless stated otherwise.

Given this assumption, our last example can be given a new time metric



which shows that this run takes exactly three steps in the new metric. It is not difficult to show that this run takes at most three steps in any time metric.

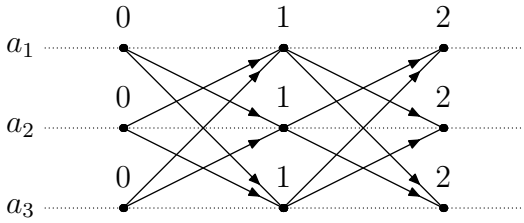
Theorem A.1.1 shows that any asynchronous run r can be assigned a time-metric function in which all messages between correct processes have bounded transmission times, and all possible time values are actually achieved. This allows the statement "event e will happen at time t" to imply "e will eventually happen".

Other definitions of latency

The other approach to measure latency of distributed algorithms is based on Lamport's clocks [78]. It was introduced by Schiper [112] under the name of *latency degree*, and later renamed to *deliver latency* by Pedone and Schiper [99].

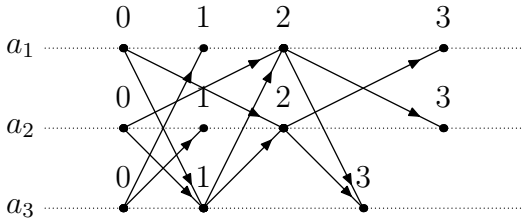
In this approach, each process is equipped with a scalar clock [78], which is an integer variable, initially zero. Any message sent at scalar time t carries the timestamp $t + 1$. When a process with a scalar time smaller than $t + 1$ receives such a message, it updates its time to $t + 1$. Otherwise, the scalar time remains the same. Deliver latency of a run is defined as latency with the scalar time used as the time metric.

As an example, consider an algorithm with the following two-phase structure, which is common to many agreement protocols. In each phase, every acceptor broadcasts and waits for other acceptors' messages to arrive. A typical execution of this algorithm is shown below



Each event has been annotated with its scalar time. The run starts at time 0 and finishes at time 2, therefore the deliver latency is $2 - 0 = 2$.

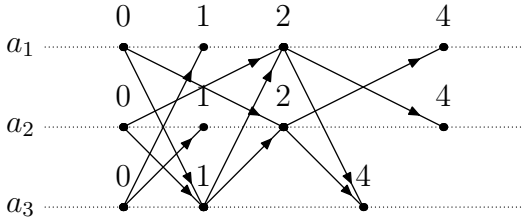
Consider a run of the same algorithm, in which all messages from and to acceptor a_3 are twice as fast as the others.



As the diagram shows, the deliver latency of this run is 3. This is counter-intuitive as the algorithm clearly takes only two steps; speeding up some channels should not increase this number. Pedone and Schiper [99] try to solve this problem by considering only runs "that exhibit minimal synchronization", however, the notion of "minimal synchrony" is unclear and never defined. This shows that deliver latency might not be suitable for measuring the number of communication steps.

How does our measure of communication steps handle this case? Consider the scalar time metric first. The latency of the last example is 3, but since the longest message latency is 2 (e.g., the first message from a_1 to a_2), this run takes only $3/2 = 1.5$ communication steps in the scalar time metric.

Consider another time metric:



Here, we get the latency of 4, which combined with the longest message transmission time of 2, gives the latency of $4/2 = 2$ communication steps in this time metric. It is not difficult to show that this run takes at most 2 communication steps in any time metric, which is consistent with our intuition.

Deciding versus halting

In most algorithms, processing stops (halts) as soon as the algorithm performed its function, for example, delivered a message. However, in some algorithms, processes can continue operating even afterwards [85]. Although early halting is desirable because it makes

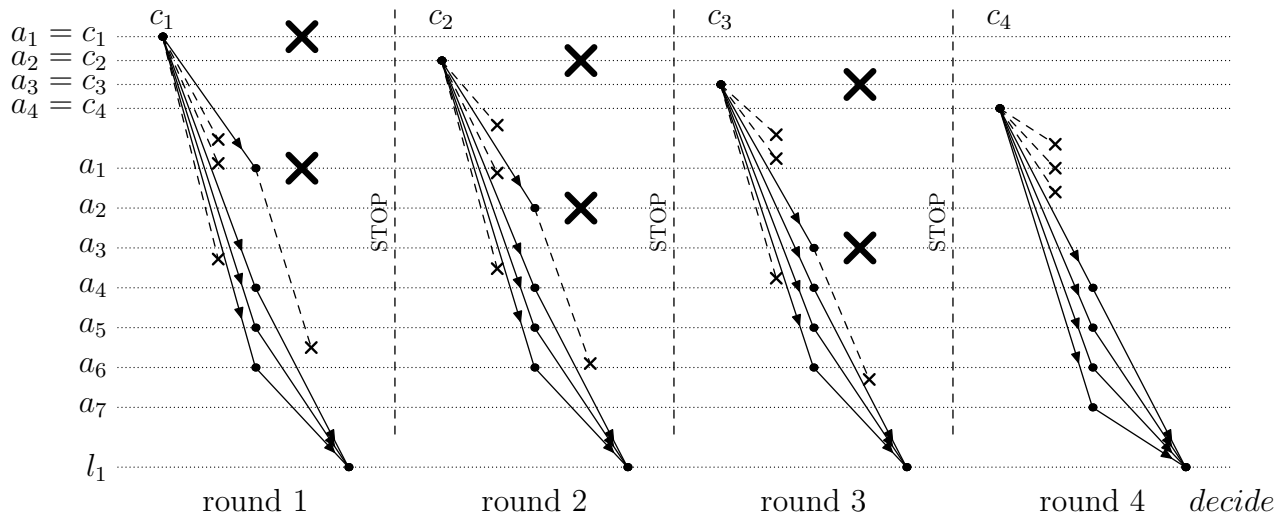


Figure 1.1: Example

better use of system resources, in this thesis, we are mainly concerned with *deciding* as quickly as possible. For this reason, in our definition of time complexity, we stop measuring time when all correct learners have decided, not when all of them have halted. For more information about halting in distributed Consensus, see [32, 95].

1.5 Consensus algorithms

A great number of Consensus algorithms for the asynchronous model have been proposed in the literature, both for crash-stop settings [16, 35, 63, 65, 73, 112], and Byzantine settings [15, 36, 81, 87, 121]. All of them share similar structures and design principles. Each of those algorithms progresses in a sequence of rounds. In each round, a special proposer, called the *round coordinator*, sends its proposal to the acceptors, who cooperate in making it a decision. Depending on the model extension, if the coordinator is suspected or the timeout expires, the round stops and the next one begins. This continues until a round with a correct coordinator manages to decide. By relying on failure detector properties or increasing the timeout period with each round, the algorithm ensures that such a round will eventually happen.

1.5.1 Crash-stop model

Figure 1.1 shows an example of this idea in crash-stop settings with less than half of the acceptors faulty ($n > 2f$). In each round i, the coordinator c_i broadcasts its proposal to the acceptors, who in turn broadcast it to the learners. A learner decides when it has received the proposal from a majority of acceptors. The coordinators c_1, c_2, ... are played by acceptors a_1, a_2, etc. In general, round i is coordinated by acceptor number $\big((i-1) \bmod n\big) + 1$, a scheme known as *rotating coordinator*.

In our example, acceptors a_1, a_2, a_3 are faulty and crash at some point. In the first round, $c_1 = a_1$ broadcasts its proposal to all acceptors, but acceptors a_2, a_3, a_7 do not receive it due to the faultiness of c_1. Moreover, all messages from the faulty a_1 to the learners are lost. Therefore, learners receive messages only from acceptors a_4, a_5, a_6, which is less than a majority. As a result, they cannot make a decision.

The first round is stopped when it has timed out (eventual synchrony) or c_1 is suspected (failure detectors), and the second round coordinator $c_2 = a_2$ broadcasts its proposal. The same problem occurs as in round 1, and no learner decides. Similarly, round 3 does not decide either. Finally, in round 4 with a correct coordinator $c_4 = a_4$, acceptor a_7 receives the proposal from c_4. As a result, the learners receive messages from a majority of acceptors (a_4, a_5, a_6, a_7) and decide.

Note that the coordinators do not always propose their original proposals. In our example, $c_2 = a_2$ cannot know that the round 1 message from a_1 to the learners was lost. As a result, it cannot preclude the possibility that round 1 has decided on c_1's proposal. Instead of issuing its own proposal, it must reissue the one proposed by c_1. But how does $c_2 = a_2$ know what c_1 proposed if c_1's message to a_2 was lost? Coordinator c_1's proposal can only become a decision if a majority of the acceptors have received it. Since we assume less than half of all acceptors are faulty ($n > 2f$), at least one *correct* acceptor must have received c_1's proposal. Coordinator c_2 can learn about this proposal from that correct acceptor. This shows that each coordinator c_i, in addition to being a proposer in round i, must also be a learner in all previous rounds.

The details of how c_2 and other coordinators choose their proposal depend on the algorithm. Algorithms by Chandra and Toueg [16], Schiper [112], Hurfin and Raynal [63], make all acceptors keep a *decision estimate*, which is used as a proposal when the acceptor becomes a coordinator. In an alternative approach, at the beginning of each round, all correct acceptors send their states to the coordinator, which uses them to compute a suitable proposal. This technique is used in Paxos [73, 79] and its variants such as Fast Paxos [11], Disk Paxos [42], and Cheap Paxos [80]. The advantage of this approach is that it can be used in Byzantine settings as well, leading to algorithms such as Byzantine Paxos [15, 81] and its improvements: Byzantine Disk Paxos [1], Paxos at War [121], and DGV [36].

Solvability

As explained in Section 1.3, solvability of Consensus requires an asynchronous system with extensions such as failure detectors or eventual synchrony. The crash detector $\Diamond S$ and leader elector Ω have been both shown to be the weakest failure detectors in their classes to make Consensus solvable [16, 18]. Recall that these extensions are safe, which means that any Consensus algorithm remains safe even if the properties they offer are not

met. A simple partitioning argument shows that solving Consensus requires a majority of correct acceptors ($n > 2f$) in the asynchronous model with any safe extension [46].

Latency

As Figure 1.1 suggests, all Consensus algorithms based on the scheme presented in Section 1.5 require two communication steps to decide, even in good runs. The first step is necessary for the coordinator's proposal to reach the acceptors, and the second for the acceptors' messages to reach the learners. This latency is optimal; no Consensus algorithm can guarantee a latency of below two communication steps in all good runs [19, 66, 67, 74]. Recall that a run is good if it is timely and all acceptors are correct.

This result does not preclude the possibility of one-step decision in *some* good runs. In fact, there are Consensus algorithms [13, 51] that decide in one step if all acceptors propose the same value. This speed comes at a cost, however; any Consensus algorithm capable of deciding in one step requires that less than a third of the acceptors are faulty ($n > 3f$).

1.5.2 Byzantine model

With the introduction of (possibly) malicious processes, the Consensus problem becomes more difficult. In addition to requiring more sophisticated algorithms, the number n of necessary acceptors grows from $n > 2f$ to $n > 2f + m$ [76, 97], where m is the maximum number of malicious acceptors ($m \leq f$). Failure detectors as defined by Chandra et al. [18] cannot be implemented even in the synchronous model [34]. For this reason, we assume the eventual synchrony model [31] in malicious settings.

Castro and Liskov [15] proposed the first asynchronous Consensus algorithm for the Byzantine model. Their algorithm uses a similar structure to the one shown in Figure 1.1, with one more communication step in each round to protect against malicious coordinators. The same algorithm has been presented in the "Paxos framework" by Lampson [81].

Both of the above algorithms require three communication steps, even in good runs. Lamport [76] observed that this latency can be reduced to two steps, provided that the number q of actually faulty acceptors is sufficiently small: $n > f + 2m + 2q$. My Paxos at War algorithm [121] was the first to achieve this bound with the assumption that all faulty processes are malicious ($m = f$). Dutta et al. [36] proposed an algorithm that achieves this bound for any $m \leq f$. Note that, in timely runs with more than q faulty acceptors, both of these algorithms decide in three communication steps. On the other hand, the algorithm proposed earlier by Kursawe [71] decides in two steps only in good runs ($q = 0$); otherwise it waits until the timeout period expires and starts one of the Consensus algorithms above.

1.6 Structure of the thesis and main results

In Section 1.5, we observed that asynchronous Consensus algorithms consist of rounds that share the same general structure: the coordinator proposes a value, and the acceptors collaborate to make this value a decision for the learners. In Chapter 2, we will formalize the notion of a round by introducing a new agreement abstraction called *Optimistically Terminating Consensus* (OTC). A full Consensus algorithm can be obtained by running a sequence of rounds implemented as OTC instances. By changing the parameters of OTC implementations we can "reconstruct" all known latency-optimal Consensus algorithms, both for benign and malicious settings. Some combinations of OTC parameters lead to new, interesting Consensus algorithms. We prove that all our OTC implementations have optimal latency.

The attractiveness of OTC lies in its simplicity and the fact that, unlike Consensus, it can be implemented in purely asynchronous settings. These two properties considerably reduce the space of possible implementations, which makes automatic discovery of new OTC implementations possible. Chapter 3 develops a theory that enables us to check the correctness of OTC algorithms automatically. If the test fails, the method presents us with a scenario in which the given algorithm behaves incorrectly, which can usually be easily generalized to lower bound proofs. However, the main application of such correctness-testing is to search the space of possible OTC implementations to automatically discover new ones.

Chapter 4 gives the detailed algorithm using a sequence of OTC instances to implement Consensus, both in benign and malicious settings. We also show how this method can be easily used to implement a variety of other agreement abstractions, such as Atomic Commitment [48] and Interactive Consistency [97]. All of these implementations have a latency *at most* equal to that of other known algorithms.

Chapter 5 investigates low-latency implementations of Atomic Broadcast, the abstraction informally introduced in the beginning of this chapter. In general good runs, Atomic Broadcast requires three steps. However, if correct acceptors receive all conflicting messages in the same order, Atomic Broadcast can be implemented in two steps. We present such implementations and compare them with existing solutions. We also show an Atomic Broadcast algorithm that exhibits two-step delivery latency in *all* good runs, provided that only acceptors can broadcast messages. We prove latency-optimality of all our solutions.

Chapter 2

Optimistically Terminating Consensus

In Section 1.5, we observed that asynchronous Consensus algorithms share the same round-based structure. In each round, a coordinator proposes some value to the acceptors, who cooperate in making this value a decision. If the first round does not succeed, then a second round is started, and so on, until eventually some round decides.

In this chapter, we consider a single round as an independent abstraction, which we call *Optimistically Terminating Consensus* (OTC). The reason for investigating this abstraction is that OTC is much simpler to implement than Consensus; in particular, it is solvable in the purely asynchronous model, without failure detectors or eventual synchrony. The correctness proofs are also simpler; Chapter 3 will show that they can even be performed automatically. Not only that, automatic testing for correctness can be used to automatically search the protocol space for new OTC algorithms satisfying given requirements.

Chapter 4 will explain how to combine individual rounds (OTC instances) into a complete Consensus algorithm. This method will allow us to reconstruct modularly almost all known asynchronous Consensus algorithms, without increasing latency. It also makes it possible to design a number of new Consensus algorithms, especially for Byzantine settings.

This chapter is structured in the following way. Section 2.1 presents a new simple broadcast abstraction called *onecast*, which will be used to implement OTC. Section 2.2 gives the precise definition of OTC and briefly explains how to use it to implement Consensus. Sections 2.3 and 2.4 present OTC implementations that decide in one and two communication steps, respectively. These two algorithms are used in Section 2.5 to (re)construct a number of known and new Consensus algorithms. More Consensus algorithms are reconstructed in Section 2.6 by running the OTC implementations from Sections 2.3 and 2.4 in parallel. Section 2.7 focuses on minimizing the number of acceptors required by OTC in good runs. Section 2.8 gives several lower bounds that prove that the requirements of

```
1   initially variables sent and received are both empty (⊥)
2   when the owner executes onecast(x) do                    { assume x ≠ ⊥ }
3     if sent = ⊥ then
4       sent ← x
5     broadcast "onecast sent" to all learners
6   initially previous = ⊥ (at learners)
7   when a learner receives "onecast x" with x ≠ ⊥ from the owner do
8     if received = ⊥ then
9       received ← x
10    onedeliver(received)
```

Figure 2.1: Implementation of onecast.

OTC implementations presented in this chapter are optimal. Section 2.9 summarizes and concludes this chapter.

2.1 Onecast

Before formally introducing OTC, we will define a new agreement abstraction called *onecast*. It will be used in the next sections to implement OTC.

In onecast, a single process, called the *owner*, broadcasts a single message to other processes (learners). Subsequent messages broadcast by the owner are ignored. Formally, the onecast abstraction is defined in terms of two actions: *onecast*(x) available to the owner, and *onedeliver*(x) available to the learners. The following properties hold:

Integrity. No learner onedelivers two different messages.

Validity. If the owner is honest and a learner onedelivers x, then the owner must have onecast x.

Agreement. If the owner is honest, then no two learners onedeliver different messages.

Termination. If the owner is correct and executes *onecast*, then all correct learners will execute *onedeliver* in one communication step.

Implementation

Figure 2.1 implements onecast as an ordinary broadcast with two enhancements: the owner does not broadcast any values different from that already broadcast, and learners do not deliver any values different from those already delivered. To this end, the owner uses a variable *sent* to remember the previously broadcast value (if any). Similarly, each

learner uses a variable *received* to remember the previously received value, if any. Both variables are initially empty, that is, they contain a special symbol "⊥". When the owner onecasts x, it first checks whether *sent* is empty, and if so it writes x to *sent*. Then, it broadcasts the contents of *sent*. When a learner receives this value, say x, it writes x to *received*, provided that *received* is empty, and then onedelivers *received*. Since honest owners never onecast "⊥", learners can ignore all "onecast" messages with this value.

All four properties of onecast are easy to prove. Integrity holds because each learner writes to *received* only once, and onedelivers only the contents of *received*. For Validity, note that a learner onedelivers x only if it has received "onecast x". If the owner is honest, then this implies that it must have onecast x (Validity). Moreover, an honest owner writes to *sent* only once, so it cannot broadcast "onecast x" for two different values (Agreement). Finally, any invocation of *onecast* results in broadcasting "onecast", and any reception of "onecast" results in onedelivery (Termination).

Example

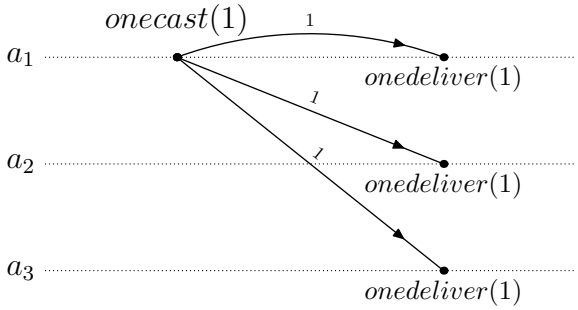
Figure 2.2 shows three scenarios, in which acceptor a_1 (the owner) onecasts 1 to other acceptors. In the first run, all acceptors are correct. As a result, all acceptors onedeliver 1 in one communication step (Termination, Validity).

In the second scenario the owner is non-maliciously faulty. It onecasts two values 1 and 4, and then crashes. The invocation *onecast*(4) knows that 1 has already been onecast, so it broadcasts 1 instead of 4. As a result, both a_2 and a_3 onedeliver the same value 1 (Agreement). Acceptor a_4 does not onedeliver anything; this does not violate Termination because a_1 is faulty (it crashes).

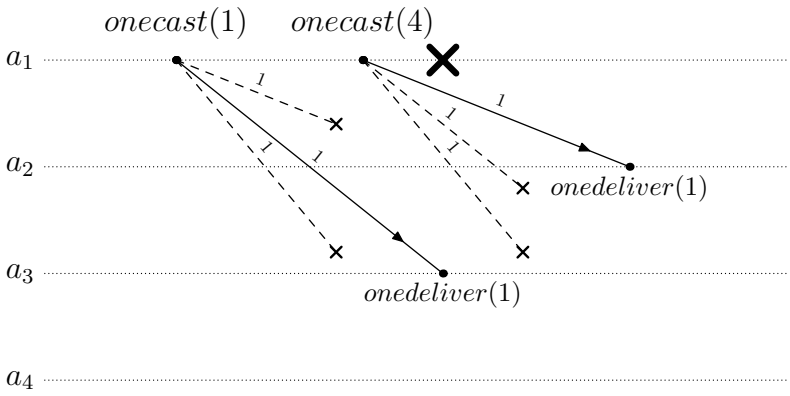
In the third scenario, the owner is malicious. It executes *onecast*(1) but sends values 2 and 3 to a_2 and a_3, which are onedelivered. This does not violate Validity or Agreement because the owner is malicious. Note that Integrity holds despite a malicious owner; although a_1 sends 4 to a_2, acceptor a_2 remembers the previously onedelivered value 2, and onedelivers it again.

2.2 Optimistically Terminating Consensus

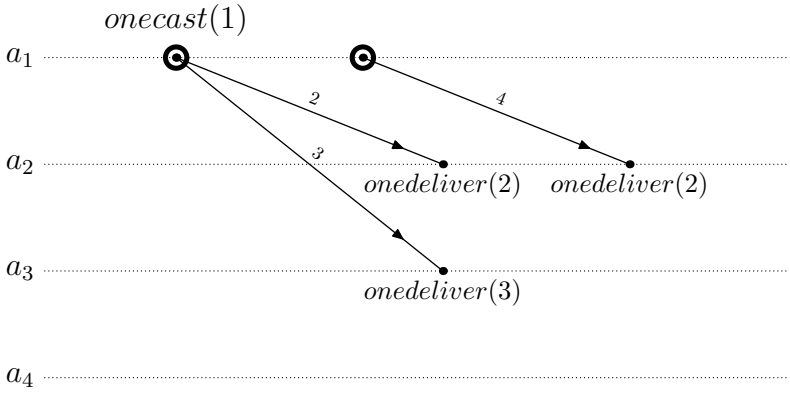
In Section 1.5, we observed that all Consensus algorithms for the asynchronous model share the same structure, presented once again in Figure 2.3. They consist of a sequence of rounds, each starting with a coordinator process broadcasting its proposal to the acceptors. Each of these coordinators must ensure that the value it proposes does not differ from any decision made by previous rounds. This can be achieved by examining states of acceptors in previous rounds (Figure 2.3).



(a) All processes correct.



(b) The owner fails by crashing, but is not malicious.



(c) The owner is maliciously faulty.

Figure 2.2: Three onecast executions.

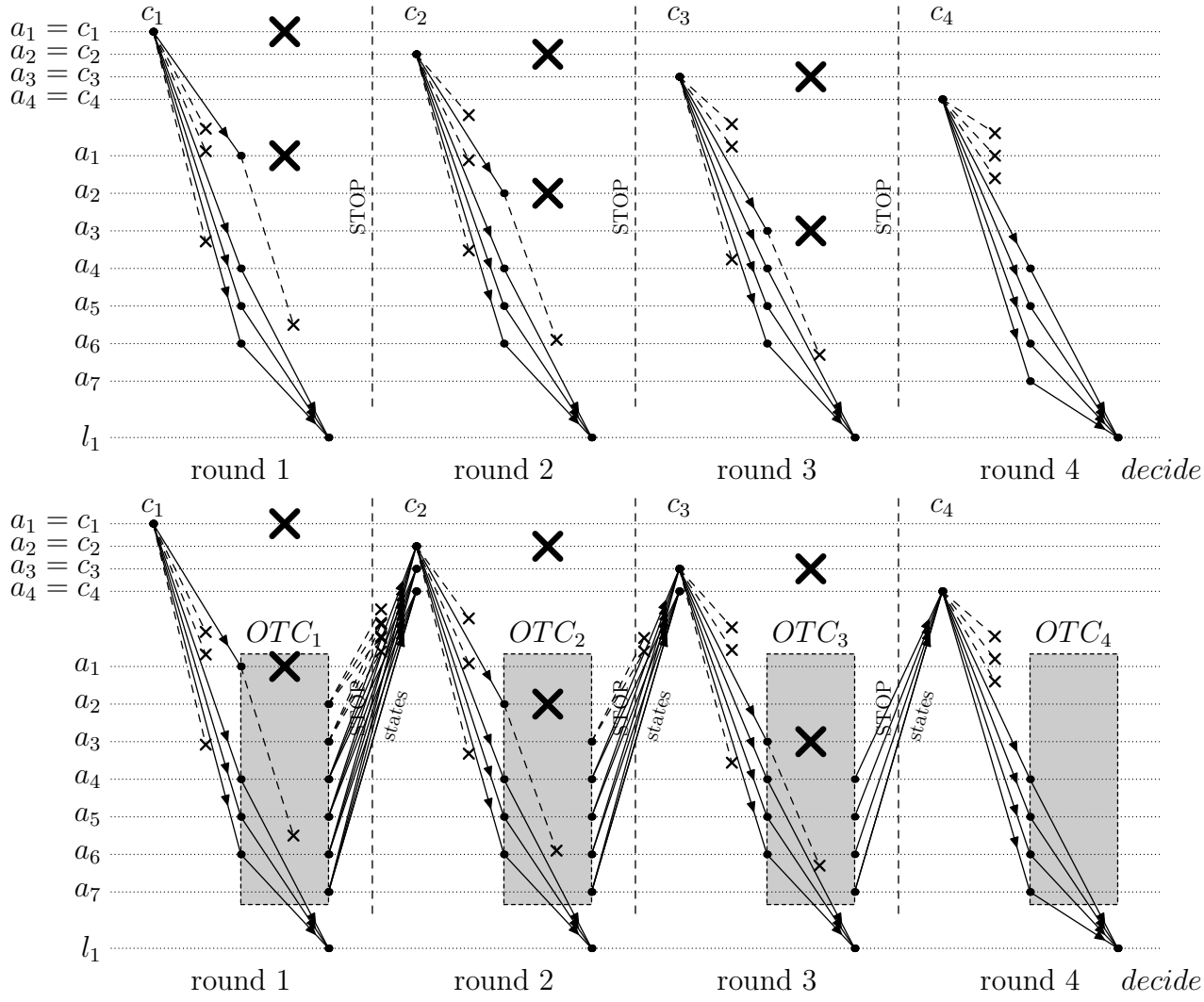


Figure 2.3: Using multiple instances of OTC to solve Consensus.

The main differences between Consensus algorithms are located inside the grey boxes, which are actually instances of the abstraction we call *Optimistically Terminating Consensus* (OTC). The phrase "optimistically terminating" refers to the fact that we require it to decide only in "optimistic runs", in which all correct acceptors propose the same value. In contrast, Consensus requires a decision in *all* runs.

In order to prevent different instances of OTC from making different decisions, we forbid proposing to an OTC instance a value that can differ from a possible decision of some previous OTC. To this end, OTC provides stronger versions of the Agreement and Validity properties, which allow us to reason about such possible decisions.

OTC Interface

The interface of the OTC abstraction is summarized in Figure 2.4. It provides every acceptor with two actions: *propose*(x) and *stop*, and every learner with three predicates: *decision*(x), *possible*(x), and *valid*(x).

Name	Process	Type	Meaning
propose(x)	acceptor	action	propose x
stop	acceptor	action	stop processing
decision(x)	learner	predicate	if true, then x is the decision
possible(x)	learner	predicate	if any learner ever decides on x, then true
valid(x)	learner	predicate	if true, then an honest acceptor proposed x

Figure 2.4: Summary of the primitives provided by OTC.

Acceptors use the action *propose*(x) to propose their proposal x. We assume that each honest acceptor proposes at most one value. Executing *stop* stops all processing and results in the acceptor entering a final unchangeable state.

Learners are equipped with three predicates: *decision*(x), *possible*(x), and *valid*(x). These predicates are functions that operate on the learner's state and return immediately without affecting the state. Predicate *decision*(x) specifies whether the learner can decide on x. We assume that a learner decides on a value x as soon as its predicate *decision*(x) becomes true. This predicate is *stable*, that is, once it is true, it will remain true forever.

Predicates *valid*(x) and *possible*(x) are used by coordinators to learn about proposed values and possible decisions in previous rounds. The stable predicate *valid*(x) is true only if an honest acceptor proposed x. The predicate *possible*(x) describes which values x are still possible decisions. Formally, if any learner ever decides on x, then *possible*(x) must hold at all times. Predicate *possible*(x) is *anti-stable*, that is, once it becomes false, it remains false forever. In other words, a once impossible decision x cannot become possible again. Before any processing starts, predicates *decision*(x) and *valid*(x) are false for any x, because no acceptor has proposed anything yet and no decision has been made. Predicate *possible*(x) starts as true, because at this stage any x can become a decision. Formally, the following properties hold:

Integrity. If *valid*(x), then an honest acceptor proposed x.

Possibility. If *decision*(x), then *possible*(x) holds at all learners, at all times.

Optimistic Termination

As opposed to Consensus, the OTC abstraction is required to decide only in "favourable runs". These are runs in which there are few faulty acceptors, all correct acceptors propose the same value, and none of them executes *stop*. Formally,

Optimistic Termination (q, k)**.** If at most q out of n acceptors are faulty, all correct acceptors propose x, and none of them executes *stop*, then *decision*(x) will hold at all correct learners in k communication steps.

Here, the maximum number q of faulty acceptors and the number k of communication steps are parameters of the Optimistic Termination property. In this case, we say that the algorithm satisfies Optimistic Termination (q, k). An OTC algorithm can satisfy more than one Optimistic Termination property; for example, satisfying Termination $(0, 1)$ and $(f, 2)$ means that the algorithm decides in one step if there are no failures and in two steps otherwise. A special symbol "$\bullet$" means "any", for example, Optimistic Termination $(0, \bullet)$ requires all correct learners to eventually decide if all acceptors are correct. (Both examples assume that all correct acceptors propose the same value and none of them executes stop.)

Permanent properties

We introduce two classes of validity and agreement properties: *standard* and *permanent*. Let us start with the standard case:

> **Standard Validity.** If $decision(x)$ holds at some learner, then an honest acceptor proposed x.
>
> **Standard Agreement.** There is at most one x for which $decision(x)$ holds at some learner.

These properties are identical to those of Consensus, except that here Standard Validity does not assume all acceptors to be honest. The requirement to decide in all runs, even if every acceptor proposes a different value, makes it impossible for Consensus algorithms to satisfy Standard Validity without this assumption. On the other hand, OTC must decide only if all correct acceptors propose the same x, which allows us to discard the honesty assumption.

We say that (the state of) a learner is *complete* if all *correct* acceptors have executed *stop* and the learner has received all messages sent by these acceptors before or by their (first) *stop* action. Therefore, if all correct acceptors execute *stop*, then all correct learners will eventually be complete. Note that a learner does not know which acceptors are correct, so it does not know whether its state is complete or not.

OTC satisfies Permanent Validity and Permanent Agreement, which are defined as:

> **Permanent Validity.** For any complete learner, $possible(x) \implies valid(x)$ for all x.
>
> **Permanent Agreement.** For any complete learner, $possible(x)$ holds for at most one x.

A state is *semi-complete* if it satisfies both of these properties, that is, if $possible(x) \implies valid(x)$ for all x and $possible(x)$ holds for at most one x. A learner can easily check

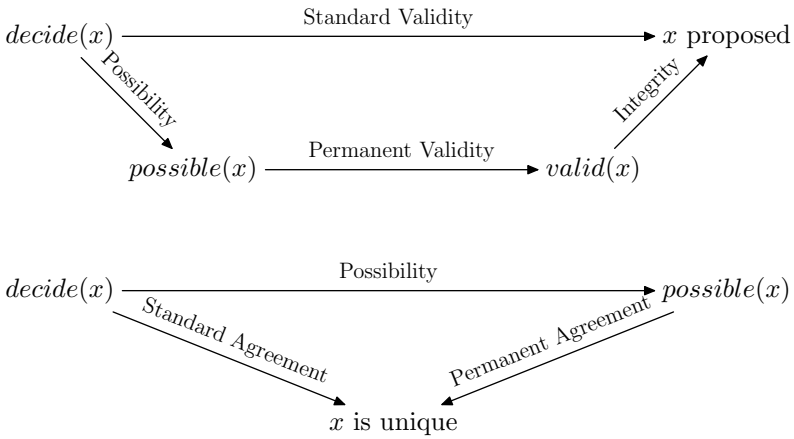


Figure 2.5: Relationships between various properties discussed in Section 2.2. Each property is an implication $p \implies q$, where p and q are predicates, and " $\implies$ " is represented as an arrow from p to q.

whether its state is semi-complete or not. In a nutshell, Permanent Validity and Permanent Agreement imply that every complete state is also semi-complete.

Standard Agreement and Validity state that there should be at most one decision, which has been proposed by an honest acceptor. If no decision has been made, these properties are always satisfied. The permanent properties are stronger; they require that, from the point of view of any correct learner, there will eventually be at most one *possible* decision, which has been *provably* proposed by an honest acceptor (assuming all correct acceptors stop). Theorems A.3.1 and A.3.2 show that any algorithm that satisfies the permanent variant of a property also satisfies the standard variant, provided that properties Integrity and Possibility hold. This is graphically shown in Figure 2.5.

OTC specification summary

As shown in Figure 2.4, the OTC abstraction is defined in terms of actions $propose(x)$ and *stop* available to acceptors, and predicates $valid(x)$, $possible(x)$, and $decision(x)$ available to learners. These primitives must satisfy the following properties:

Integrity. If $valid(x)$, then an honest acceptor proposed x.

Possibility. If $decision(x)$, then $possible(x)$ holds at all learners, at all times.

Permanent Validity. For any complete learner, $possible(x) \implies valid(x)$ for all x.

Permanent Agreement. For any complete learner, $possible(x)$ holds for at most one x.

Optimistic Termination (q, k)**.** If at most q out of n acceptors are faulty, all correct acceptors propose x, and none of them executes *stop*, then *decision*(x) will hold at all correct learners after k communication steps.

The pair (q, k) is a parameter of the OTC abstraction.

Action *stop*

Earlier in this section, we said that acceptors stop all processing (halt) after executing *stop*. This semantics is not part of the specification; OTC algorithms can continue operating after executing *stop*. However, most OTC implementations adhere to this semantics because, as we will show now, halting after executing *stop* cannot violate the correctness of an OTC algorithm.

We will restrict our attention to correct acceptors; by definition, faulty acceptors can halt at any moment without violating the specification. Any correct acceptor executing *stop* satisfies all Optimistic Termination properties. Other OTC properties are time-independent safety properties, so they cannot be violated by not performing actions. The definition of a "complete state", used in Permanent Validity and Permanent Agreement, does not change because it depends only on actions performed by the acceptor until it executed *stop* for the first time.

2.2.1 Implementing Consensus

Our OTC-based Consensus algorithm progresses in a sequence of *rounds*. Initially, the first round tries to decide on some value. If the first round does not seem to make progress, it is stopped, and the second round takes over. If the decision has not been made by the second round, it is stopped as well, and the third rounds starts, etc. Each round i has a special proposer c_i called the *round coordinator*, and the corresponding OTC instance OTC_i. Coordinator c_i broadcasts its proposal to the acceptors, who propose it to the instance OTC_i. Decisions made by OTC instances become final decisions. In this section, we will only briefly explain how OTC properties are useful to implement this idea; the full details will be provided in Chapter 4.

As shown in Figure 2.6, the first coordinator c_1 sends its proposal to all acceptors, who propose it to the first OTC instance OTC_1. If all correct learners decide in the first round, the algorithm can terminate. Otherwise, correct acceptors stop the first round, and coordinator c_2 starts the next one. Coordinator c_2 behaves analogously to c_1: it sends a proposal to all acceptors, who propose it to the second round OTC instance OTC_2. If OTC_2 does not seem to make progress, it is stopped, and c_3 starts the third round, and so on.

Coordinator c_1 always sends its own proposal to the acceptors. This might not be true with other coordinators; they have to make sure that the proposals they send to the

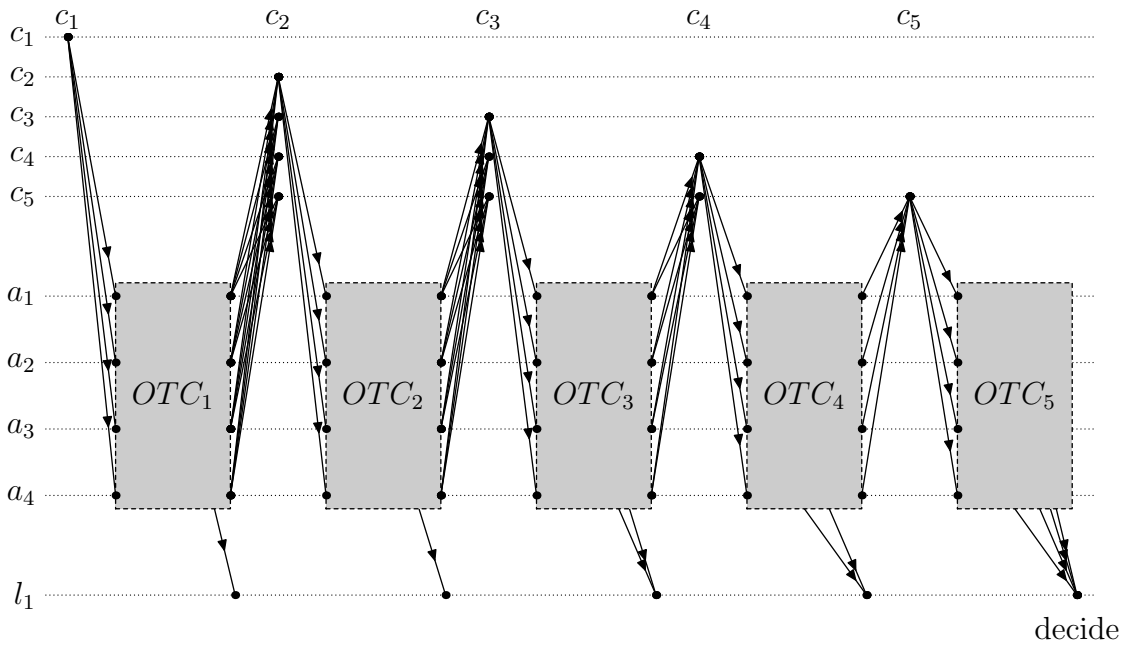


Figure 2.6: Using multiple OTC instances to implement Consensus.

acceptors do not differ from any decision made by the previous rounds. For example, coordinator c_2 can propose its own value only if it is sure that no decision was made in the first round. Otherwise, it must re-propose the value of that possible decision.

To choose its proposal, coordinator c_2 uses the *possible* and *valid* predicates of OTC_1. When all correct acceptors have stopped the first round, Permanent Agreement guarantees that eventually either:

1. Predicate $possible(x)$ does not hold for any x. This means that no decision was made in OTC_1, so coordinator c_2 can issue its own proposal.

2. Predicate $possible(x)$ holds for exactly one x. In this case, Permanent Validity implies $valid(x)$ so an honest acceptor proposed x to OTC_1. Therefore, if c_1 is honest, it must have proposed x, so coordinator c_2 can reissue this proposal without violating Validity.

Coordinators c_3, c_4, ... choose their proposals using a slightly more complicated, but similar, reasoning. See Chapter 4 for details.

In each round i, acceptors propose to OTC_i the value received from the coordinator c_i. A malicious coordinator can make acceptors propose different values, however, this is not a problem because the OTC abstraction tolerates different proposals. Agreement can be violated only if c_i deliberately proposes a value different from some previous round decision. Chapter 4 will explain how digital signatures can be used to prevent this. For the moment, notice that digital signatures are only necessary for the second and later rounds; coordinator c_1 cannot issue a proposal different from a decision made by

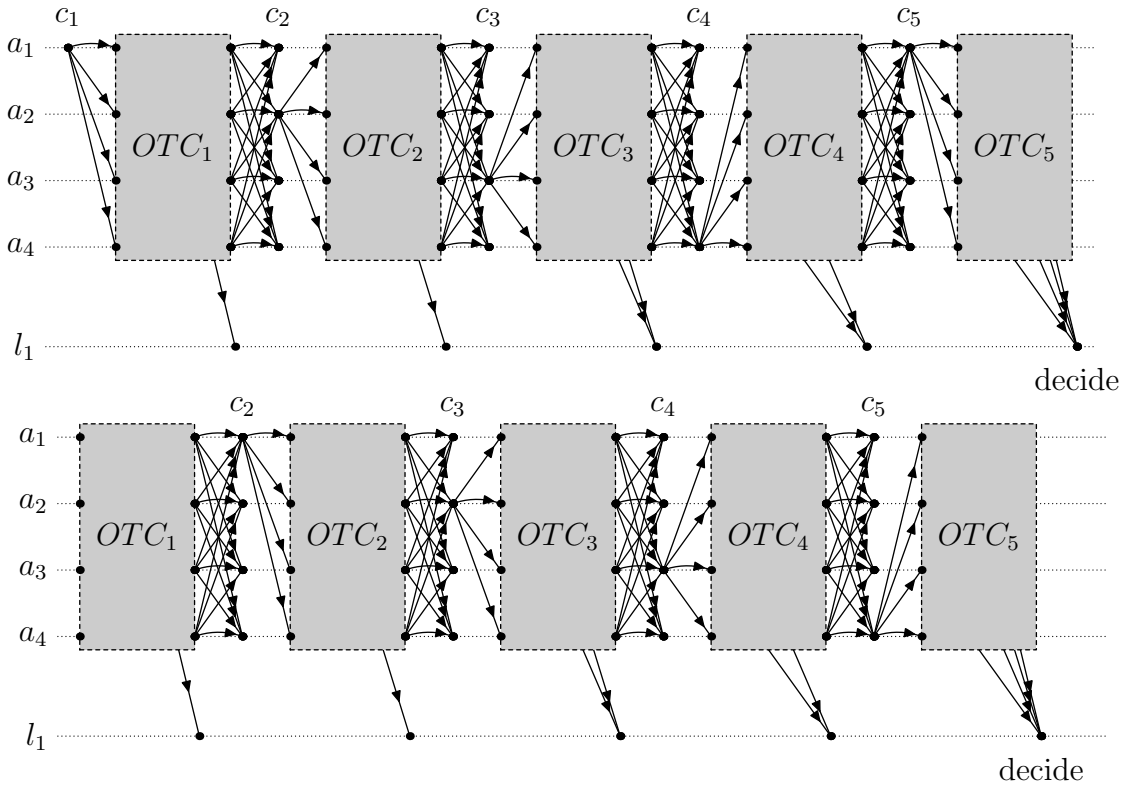


Figure 2.7: OTC-based Consensus with coordinators played by the acceptors using the rotating coordinator strategy. In the lower diagram, the first round coordinator is virtual.

previous rounds because so such rounds exist. Therefore, if the first round decides, digital signatures are not used.

Acceptors stop rounds if they suspect the coordinator (the failure detector model) or the timeout has expired (eventual synchrony). Assume that all OTC instances satisfy Optimistic Termination $(f, \bullet)$, that is, they decide if all correct acceptors proposed the same value and none of them executed *stop*. Therefore, any round i with a correct coordinator will decide, provided that none of the correct acceptors stops it before. Chapter 4 will explain how unreliable failure detectors or eventual synchrony can be used to eventually prevent acceptors from stopping such rounds, thereby ensuring Termination.

In a typical Consensus implementation, the coordinators are played by acceptors, using the *rotating coordinator* paradigm (Figure 2.7). To improve the latency, acceptors can issue their proposals to the first round OTC directly, instead of waiting for the coordinator's proposal. This corresponds to the first round having a *virtual coordinator*, a possibly malicious coordinator that sends to each acceptor this acceptor's proposal. In runs where all acceptors propose the same value, using a virtual coordinator reduces the latency by one step. If the proposals are different, the decision will be made in the second round, with a real (non-virtual) coordinator.

The construction of the Consensus algorithm shows that, in "favourable" runs, the decision is made by the first OTC. Therefore, the latency of a Consensus algorithm in such

```
1   when acceptor a_i executes propose(x) do
2     onecast x using onecast_i
3   when acceptor a_i executes stop do
4     onecast ⊤ using onecast_i
5   predicate decision(x) at a learner is
6     at least n − q instances onecast_i delivered x
7   predicate possible(x) at a learner is
8     at most q + m instances of onecast_i delivered a non-x
9   predicate valid(x) at a learner is
10    more than m instances of onecast_i delivered x
```

Figure 2.8: Generic Agreement.

runs is entirely determined by the latency of the first round OTC. For this reason, in this chapter, we can concentrate on OTC implementations and the latency of the Consensus algorithms using them, while deferring a detailed discussion of Consensus implementations to Chapter 4. To compute the latency of a Consensus algorithm, one communication step must be added to the latency of the first round OTC, to allow the coordinator's proposal to reach the acceptors (unless a virtual coordinator is used).

2.3 Implementing OTC in one communication step

Consider a system composed of n acceptors, out of which at most f are faulty, out of which at most m are malicious. In this section, we will be interested in OTC algorithms that satisfy

> **Optimistic Termination** $(q, 1)$**.** If at most q acceptors are faulty, all correct acceptors propose x, and none of them executes *stop*, then *decision*(x) will hold at all correct learners after one step.

The implementation shown in Figure 2.8 uses n instances of onecast (Section 2.1). Each acceptor a_i owns one instance $onecast_i$, and uses it to onecast its proposal (lines 1–2). Similarly, a_i executes *stop* by onecasting a special symbol $\top$ using the same instance $onecast_i$. Note that symbols $\perp$ (used internally by onecast), $\top$, and possible proposals x are all different.

As shown in Figure 2.8, predicates *valid*(x), *decision*(x), and *possible*(x) are determined by the values delivered by onecast instances $onecast_i$. Predicate *valid*(x) is true if more than m instances $onecast_i$ delivered x. At least one of them must belong to an honest acceptor a_i, which must have proposed x (Integrity). Predicate *decision*(x) holds if at least $n - q$ onecast instances delivered x. This means that if all $n - q$ correct acceptors propose x and do not execute *stop*, all correct learners will decide in one

communication step (Optimistic Termination). Finally, predicate *possible*(x) is true if at most $q + m$ onecast instances delivered a value different from x. If *decision*(x) holds at some learner, then at least $n - q - m$ instances $onecast_i$ owned by *honest* acceptors a_i must have onedelivered x. The onecast Agreement property forbids these instances to deliver values different from x. As a consequence, non-x values can be delivered only by the other $q + m$ onecast instances, which makes predicate *possible*(x) true at all learners, at all times (Possibility).

2.3.1 Validity and Agreement

In order to satisfy Permanent Validity and Permanent Agreement, the Generic Agreement algorithm in Figure 2.8 requires additional assumptions on n. First, we will prove that $n > f + 2m + q$ is sufficient to guarantee Permanent Validity. This property states that if all correct acceptors have executed *stop*, and a learner has received all messages sent by these acceptors before the *stop* actions finished, then *possible*$(x) \implies$ *valid*(x) for every x. Every execution of *stop* involves onecasting, so the assumption implies all $n - f$ onecast instances owned by correct acceptors have executed *onedeliver*. If *possible*(x) holds, then at most $q + m$ onecast instances onedelivered a non-x. This means that at least $n - f - q - m > m$ instances owned by correct acceptors onedelivered x, which implies *valid*(x).

Permanent Agreement requires $n > f + 2m + 2q$. Assume that *possible*(x) and *possible*(y) hold some values x and y. This means that at most $2q + 2m$ instances have onedelivered either a non-x or a non-y. The previous paragraph explained why all $n - f > 2q + 2m$ instances $onecast_i$ corresponding to correct a_i onedeliver something. Therefore, at least one instance onedelivered a value which is neither non-x nor non-y, that is, which equals x and y at the same time. This is only possible if $x = y$, so Permanent Agreement holds.

Note that the only way the *stop* action was used in the proofs above was as a trigger performing a onecast. Since onecast is also performed by *propose*(x), these two actions are indistinguishable from the point of view of Permanent Validity and Permanent Agreement. In other words, once an acceptor has proposed, it can behave as if it has already executed *stop*, and cease operating. This equivalence of *stop* and *propose*(x) is specific to this particular OTC algorithm and does not hold for others presented in this chapter.

2.3.2 Single-value OTC

Although the OTC abstraction allows different acceptors to propose different values, in most cases all proposals are the same. For example, in the Consensus implementation sketched in Section 2.2.1, honest acceptors always propose the value received from the coordinator. If the model does not allow the coordinator to be malicious and send different

```
1  when acceptor executes propose(x) do
2    if x = x_0 then
3      propose_S(x)
4    else
5      stop_S
```

Figure 2.9: Implementing privileged-value OTC using an instance S of single-value OTC.

proposals to different acceptors, the first round OTC implementation can assume all proposals from honest acceptors to be the same. Some honest acceptors might not propose anything at all; this can happen if the coordinator crashes and fails to send its proposal to some or all of the acceptors.

Explicitly assuming that all *honest* acceptors propose the same value might enable us to design OTC implementations that require fewer acceptors than OTC implementations tolerating different proposals. For this reason, we distinguish between these two kinds of OTC, calling the former *single-value OTC*, and the latter *multi-value OTC*.

Single-value OTC differs from multi-value OTC in that it does not have to explicitly satisfy Permanent Agreement, which in that case follows automatically from Permanent Validity. Indeed, assume that $possible(x) \implies valid(x)$ for all x. If $possible(x)$ holds for two different x, then $valid(x)$ holds for two different x, which implies that two different values have been proposed by honest acceptors. This contradicts the assumption all honest acceptors propose the same value.

As mentioned above, single-value OTC can be used in systems with honest coordinators, thereby reducing the required number n of acceptors. However, this does not apply to rounds with virtual coordinators, in which acceptors propose their proposals to the OTC directly. Since these proposals may be different even with honest acceptors, a multi-value OTC must be used.

2.3.3 Privileged-value OTC

Privileged-value OTC is a form of OTC that differs from the original one in that the Optimistic Termination properties hold only if all correct processes propose the privileged value x_0. It can be used to construct Consensus algorithms that are particularly fast in deciding on x_0, at the expense of being slow for other values. This can be useful in systems when one particular value has a significantly higher probability of being proposed than the others. Such Consensus algorithms should have the first round with a virtual coordinator, in which all acceptors propose their proposals directly to the privileged value OTC. If all correct acceptors propose x_0, then the algorithm will decide in the first round. If not, the decision will be taken by one of the other rounds, which use normal OTCs.

Similarly to single-value OTCs, privileged-value OTCs require fewer acceptors than multi-value OTCs. In fact, a privileged value OTC can be easily implemented using single-value OTC, as shown in Figure 2.9. To propose x_0, an acceptor passes it to the underlying instance S of single-value OTC. For other values, the acceptor stops the instance. All other primitives (*stop*, *decision*, *possible*, *valid*) are identical to those of the OTC instance S.

2.4 Implementing OTC in two communication steps

The Generic Agreement algorithm from Figure 2.8 implements multi-value OTC provided that $n > f+2m+2q$. In particular, when all faulty acceptors can be malicious ($m = f$) and the decision should be reached regardless of their actual number ($q = f$), this requirement becomes $n > 5f$. On the other hand, only $n > 3f$ is required to solve Consensus in Byzantine settings [15, 97]. Can we design a multi-value OTC algorithm that would require only $n > 3f$? The answer is "yes". However, as opposed to Generic Agreement, such an implementation requires more than one communication step to decide.

In this section, we will consider OTC algorithms implemented as *chains* of Generic Agreement instances

$$A_1 \quad \rightarrow \quad A_2 \quad \rightarrow \quad \cdots \quad \rightarrow \quad A_k.$$

Acceptors propose their value to the first instance A_1. Then, decisions are propagated along the chain: if an acceptor reaches a decision x in instance A_i, it immediately proposes it to the next instance A_{i+1}. The decision of the last instance A_k becomes the final decision. Stopping the algorithm involves stopping all instances $A_1, \ldots, A_k$.

The predicate *valid* is taken from the first instance, whereas predicates *possible* and *decision* come from the last instance:

$$valid(x) \stackrel{\text{def}}{=} valid_{A_1}(x), \qquad possible(x) \stackrel{\text{def}}{=} possible_{A_k}(x), \qquad decision(x) \stackrel{\text{def}}{=} decision_{A_k}(x).$$

Properties Integrity and Possibility of $A_1 \rightarrow \cdots \rightarrow A_k$ follow immediately from analogous properties of instances A_1 and A_k, respectively. For Optimistic Termination, assume that at most q acceptors are faulty and none of them executes *stop*. Each instance A_i satisfies Optimistic Termination $(q, 1)$: if all correct acceptors propose x to A_i, then all correct learners will decide on x in one step. Since all acceptors are learners, they will all propose x to A_{i+1}. By simple induction, we see that $A_1 \rightarrow \cdots \rightarrow A_k$ satisfies Optimistic Termination (q, k).

Theorems A.5.2 and A.5.3 prove that for $k \geq 2$, the chain $A_1 \rightarrow \cdots \rightarrow A_k$ satisfies Permanent Validity and Permanent Agreement provided that $n > f + m + q$. Since the required number n of acceptors is the same for all $k \geq 2$, we will focus our attention on the two-step OTC algorithm $A_1 \rightarrow A_2$, which satisfies

Variant	General	All honest	All malicious	$q = 0$	$q = f$
one-step OTC					
single-value	$n > f + 2m + q$	$n > f + q$	$n > 3f + q$	$n > f + 2m$	$n > 2f + 2m$
multi-value	$n > f + 2m + 2q$	$n > f + 2q$	$n > 3f + 2q$	$n > f + 2m$	$n > 3f + 2m$
two-step OTC					
multi-value	$n > f + \ m + q$	$n > f + q$	$n > 2f + q$	$n > f + m$	$n > 2f + m$

Figure 2.10: Requirements of one-step and two-step OTC implementations.

Optimistic Termination $(q, 2)$**.** If at most q acceptors are faulty, all correct acceptors propose x, and none of them executes *stop*, then *decision*(x) will hold at all correct learners after two communication steps.

2.5 Consolidation

The previous sections discussed one-step OTC implemented by a single instance of Generic Agreement (Section 2.3) and two-step OTC implemented by two Generic Agreement instances (Section 2.4). We were considering two kinds of one-step OTC: the *single-value* variant, in which all acceptors must propose the same value, and the *multi-value* variant, without this restriction. As explained in Section 2.3.3, privileged-value OTC requires the same number of acceptors as single-value OTC. For two-step OTC, we consider only the multi-value variant because the single-value variant requires the same number of acceptors.

For each of these variants, Figure 2.10 presents the required number n of acceptors in general, as well as in four common situations: when all acceptors are honest ($m = 0$), when all faulty acceptors are malicious ($m = f$), when Optimistic Termination requires all acceptors to be correct ($q = 0$), and when Optimistic Termination holds regardless of the number of faulty acceptors ($q = f$).

The conditions on n presented in Figure 2.10 are optimal and cannot be improved. Theorem 2.8.1 states that any one-step single-value OTC algorithm requires $n > f + 2m + q$. Similarly (Theorem 2.8.2), any one-step multi-value OTC algorithm requires $n > f + 2m + 2q$. Theorem 2.8.3 proves that $n > f + m + q$ is necessary for any, even single-value, OTC algorithm, regardless of the number of communication steps.

Consequences for implementing Consensus

As explained in Section 2.2.1, the latency of a Consensus algorithm in "favourable" runs depends solely on the OTC implementation used in the first round. Since the later rounds are used only in "non-favourable" runs, their OTCs are optimized for resilience rather than latency. For this reason, non-first round OTCs should guarantee Optimistic

Termination $(q, \bullet)$ regardless of the number of faulty acceptors ($q = f$). Theorem 2.8.3 proves that any such OTC algorithm requires $n > f + m + q = 2f + m$. This lower bound is tight; Figure 2.10 shows that this condition is sufficient for a two-step multi-value OTC implementation. Moreover, in systems with honest coordinators and acceptors, this condition becomes $n > f + q = 2f$, so we can use the one-step single-value OTC implementation.

To compute the number n of acceptors required by an OTC-based Consensus algorithm, one should take the maximum over the requirements of all individual rounds. The previous paragraph explained that any such Consensus algorithm requires $n > 2f + m$. Lamport [76] shows that this is true for *any* Consensus algorithm, so the OTC framework causes no overhead in this respect.

The requirement $n > 2f + m$ implies that optimizing the first round OTC to allow $n \leq 2f + m$ makes little sense. For example, in the crash-stop model ($m = 0$), we can use one-step single-value OTC for the first round, which requires $n > f + q$. Setting $q = f$ gives $n > 2f$; using $q < f$ makes no sense, because non-first round OTCs require $n > 2f$ anyway. Similarly, if we use the two-step OTC, which requires $n > f + m + q$, for the first round, we can assume $q = f$, because $n > 2f + m$ is necessary for other rounds.

Single-value OTCs have the same requirement $n > f + q$, regardless of how many steps they require. For this reason, there is no point in using two-step OTCs in the crash-stop model, unless with a virtual coordinator.

Consensus algorithms

Section 2.2.1 explained how to use the OTC algorithms from Sections 2.3 and 2.4 to implement Consensus. In this way, we can construct a variety of Consensus protocols by changing a number of parameters: the number n of acceptors, the number f of faulty acceptors, the number m of malicious acceptors, the number k of communication steps in which a decision will be made, the maximum number q of faulty acceptors with which that decision is guaranteed, honest vs. malicious coordinators, the first round deciding on any value vs. only on the privileged one, real vs. virtual coordinator of the first round. By choosing appropriate parameters, we can match the latency of most known Consensus algorithms (see the list below). All statements about the latency of Consensus algorithms assume timely runs, that is, no correct acceptors suspected if failure detectors are used, or sufficiently fast messages in the eventual synchrony model.

- **Two-step Consensus.** The algorithms proposed by Chandra et al. [18], Hurfin and Raynal [63], Lamport [73], Schiper [112] all assume honest processes. They require $n > 2f$ and guarantee decision in two steps provided that the first round coordinator is correct.

To achieve the same properties in the OTC framework, note that the assumed honesty of the coordinators means that we can use one-step single-value OTCs for all rounds. Setting $q = f$ and $m = 0$ (no malicious acceptors), gives a Consensus algorithm that requires $n > 2f$ and decides in two steps if the first round coordinator is correct. Algorithms [18, 63, 73, 112] have the same properties.

- **One-step Consensus.** The algorithm by Brasileiro et al. [13] assumes honest processes. It requires $n > 3f$ and guarantees decision in one step provided that all correct acceptors propose the same value.

  In the OTC framework, we use a virtual coordinator for the first round, implemented as one-step multi-value OTC, which requires $n > f + 2q$. Other rounds have real, honest coordinators with one-step single-value OTCs having $q = f$, which requires $n > 2f$. In total, our approach requires $n > f + \max\{f, 2q\}$. This is better than the $n > 3f$ required by [13] for all $q < f$, and the same for $q = f$.

- **Byzantine Paxos.** The Byzantine Paxos algorithm by Castro and Liskov [15] tolerates malicious processes and assumes all faulty acceptors to be malicious ($f = m$). It requires $n > 3f$ and guarantees a decision in three communication steps, provided that the coordinator of the first round is correct.

  In the OTC framework, we use real coordinators and two-step multi-value OTC instances for all rounds. This requires $n > 2f + m = 3f$, the same as in [15].

- **Optimistic Byzantine Agreement.** Optimistic Asynchronous Byzantine Agreement by Kursawe [71] tolerates malicious processes and assumes all faulty acceptors to be malicious ($f = m$). It requires $n > 3f$ and, in the absence of failures, it guarantees decision in two communication steps.

  In the OTC framework, we use real coordinators for all rounds. The first round uses one-step multi-value OTC ($n > 3f + 2q$), all the others use two-step multi-value OTC ($n > 3f$). The requirement $n > 3f + 2q$ of the first round is dominant. This, given the assumption $q = 0$, is equivalent to the $n > 3f$ required in [71].

- **Fast Byzantine Paxos.** The Fast Byzantine Paxos algorithm by Martin and Alvisi [87] tolerates malicious processes and assumes all faulty acceptors to be malicious ($f = m$). It requires $n > 5f$, and guarantees a decision in two communication steps, provided that the coordinator of the first round is correct.

  In the OTC framework, we use the same parameters as in the previous point, which requires $n > 3f + 2q$. This, given the assumption $q = f$, is equivalent to the $n > 5f$ required in [87].

In addition to reconstructing existing algorithms, the OTC abstraction can be used to design new ones. Below, we list three new one-step Consensus algorithms. All of them use a virtual coordinator for the first round and real coordinators for other rounds.

- **One-step privileged-value Consensus.** This algorithm assumes all acceptors to be honest, and decides in one step if all correct acceptors proposed the privileged value x_0. For the first round, it uses virtually coordinated one-step privileged-value OTC with $q = f$, which requires $n > f + q = 2f$. For other rounds, we use single-value one-step OTC with $q = f$, which also requires $n > f + q = 2f$. In comparison to the one-step Consensus algorithm by Brasileiro et al. [13], this algorithm guarantees one-step decision only for the privileged value x_0. On the other hand, it requires $n > 2f$, as opposed to the $n > 3f$ required by Brasileiro et al. [13].

- **One-step Byzantine Consensus.** This algorithm tolerates malicious processes. It decides in one step if at most q acceptors are faulty and all correct acceptors propose the same value. For the first round, it uses virtually coordinated one-step multi-value OTC, which requires $n > f + 2m + 2q$. For other rounds, we use two-step multi-value OTCs with $q = f$, which require $n > 2f + m$. This gives the final requirement of $n > f + m + \max\{f, m + 2q\}$.

- **One-step privileged-value Byzantine Consensus.** This algorithm tolerates malicious processes. It decides in one step if at most q acceptors are faulty and all correct acceptors proposed the privileged value x_0. For the first round, it uses virtually coordinated one-step privileged-value OTC, which requires $n > f + 2m + q$. For other rounds, we use two-step multi-value OTCs with $q = f$, which require $n > 2f + m$. This gives the final requirement of $n > f + m + \max\{f, m + q\}$.

2.6 Combining one-step OTC with two-step OTC

The previous section showed a trade-off between OTC implementations: one-step OTCs are fast but require many acceptors, whereas two-step OTCs are slower but can work with fewer acceptors. The choice between these two implementations is far from trivial, because the number of currently faulty acceptors is not known. Ideally, we would like to use both implementations simultaneously.

In this section, we will present the multi-step OTC algorithm that satisfies the three Optimistic Termination conditions $(q_1, 1)$, $(q_2, 2)$, and $(q_3, 3)$ at the same time. We assume $q_1 \leq q_2 \leq q_3$ because Optimistic Termination $(q_1, 1)$ implies $(q_1, 2)$, and Optimistic Termination $(q_2, 2)$ implies $(q_2, 3)$.

The multi-step OTC algorithm consists of three OTC chains from Section 2.4 executed in parallel:

$$\begin{array}{llllll} A_1 & & & & & \text{with } q = q_1, \\ B_1 & \to & B_2 & & & \text{with } q = q_2, \\ C_1 & \to & C_2 & \to & C_3 & \text{with } q = q_3. \end{array}$$

Instances A_1, B_1, and C_1 share onecast instances; each proposed value is proposed to all three chains at the same time. In other words, $\mathit{propose}(x)$ consists of $\mathit{propose}_{A_1}(x)$, $\mathit{propose}_{B_1}(x)$, and $\mathit{propose}_{C_1}(x)$. Stopping the algorithm involves stopping all six Generic Agreement instances.

Predicates $\mathit{decision}(x)$, $\mathit{possible}(x)$, and $\mathit{valid}(x)$, are defined as

$$\begin{aligned} \mathit{valid}(x) &\stackrel{\text{def}}{=} \mathit{valid}_{A_1}(x) \vee \mathit{valid}_{B_1}(x) \vee \mathit{valid}_{C_1}(x) \\ \mathit{decision}(x) &\stackrel{\text{def}}{=} \mathit{decision}_{A_1}(x) \vee \mathit{decision}_{B_2}(x) \vee \mathit{decision}_{C_3}(x) \\ \mathit{possible}(x) &\stackrel{\text{def}}{=} \big(\mathit{possible}_{A_1}(x) \wedge \neg\exists\, x' \neq x : \mathit{valid}_{C_2}(x')\big) \vee \mathit{possible}_{B_2}(x) \vee \mathit{possible}_{C_3}(x) \end{aligned}$$

In other words, the global predicate $\mathit{valid}(x)$ is true if it holds for at least one of the instances A_1, B_1, C_1. Similarly, $\mathit{decision}(x)$ holds if it holds for at least one of A_1, B_2, C_3. Predicate $\mathit{possible}(x)$ is an improved version of the more natural definition

$$\mathit{possible}(x) \stackrel{\text{def}}{=} \mathit{possible}_{A_1}(x) \vee \mathit{possible}_{B_2}(x) \vee \mathit{possible}_{C_3}(x).$$

It states that x is a possible decision of A_1 only if $\mathit{valid}_{C_2}(x')$ holds for no $x' \neq x$. Indeed, if some honest acceptor proposed x' to C_2, then some learner decided on x' in C_1. Instances A_1 and C_1 share the same proposals and onecast instances, so they cannot reach different decisions x and x' (Lemma A.4.3). Using this observation in the definition of $\mathit{possible}(x)$ reduces the minimum number of acceptors required in some cases.

The system of three chains A_1, $B_1 \to B_2$, $C_1 \to C_2 \to C_3$ implements OTC. Properties Integrity, Possibility, and Optimistic Termination (q_i, i) for $i = 1, 2, 3$ follow easily from the analogous properties of the individual chains. The same applies to Permanent Validity, which therefore requires

$$n > f + 2m + q_1, \qquad n > f + m + q_2, \qquad n > f + m + q_3.$$

Since we assume $q_3 \geq q_2$, the third requirement implies the second.

Theorem A.6.1 shows that Permanent Agreement additionally requires

$$n > f + 2m + 2q_1 \quad \text{and} \quad n > f + m + q_2 + \min\{m, q_1\}.$$

Proposals	General	All benign	All malicious
single-value	$n > f + 2m + q_1$		$n > 3f + \ q_1$
	$n > f + \ m + q_3$	$n > f + q_3$	
multi-value	$n > f + 2m + 2q_1$	$n > f + 2q_1$	$n > 3f + 2q_1$
	$n > f + \ m + q_2 + \min\{m, q_1\}$		
	$n > f + \ m + q_3$	$n > f + q_3$	

Figure 2.11: Requirements of the multi-step OTC implementation.

The former requirement ensures Permanent Agreement of instance A_1. The other is necessary to ensure Permanent Agreement between decisions made by A_1 and the two other chains.

Discussion

The requirements of multi-step OTC are summarized in Figure 2.11. The single-value version can be obtained by ignoring all Permanent Agreement requirements. Both conditions $n > f + 2m + q_1$ and $n > m + q_3$ match their respective lower bounds set by Theorems 2.8.1 and 2.8.3. As a result, the single-value multi-step OTC algorithm is strictly better than its one-step and two-step counterparts from Sections 2.3 and 2.4, respectively.

Implementing multi-value multi-step OTC requires two additional conditions: $n > f + 2m + 2q_1$ and $n > f + m + q_2 + \min\{m, q_1\}$, both of which match their respective lower bounds set by Theorems 2.8.2 and 2.8.4. The second condition is stronger than the analogous condition for the two-step OTC ($n > f + m + q_2$), at least for non-zero m and q_1. This discrepancy was first noticed by Dutta et al. [36], who proved that any Consensus implementation requires $n > f + m + q_2 + \min\{m, q_1\}$ assuming $q_2 = f$.

The special case $q_3 = f$ is important for Consensus implementations. It requires $n > 2f + m$, which is required by any such implementation anyway [76]. In exchange, it guarantees

> **Optimistic Termination** $(f, 3)$**.** If all correct acceptors proposed the same value x, and none of them executes *stop*, then *decision*(x) will hold at all correct learners after three communication steps.

Algorithms

The multi-step OTC described in this section allows us to reconstruct two Byzantine Consensus algorithms. In both of them we use multi-value multi-step OTC for the first round, and two-step multi-value OTC with $q = f$ for the other rounds.

- **Paxos at war.** My "Paxos at war" algorithm [121] assumes that all faulty acceptors are malicious and requires $n > 3f + 2q_1$. In timely runs with a correct first round

coordinator, it decides in two steps if at most q_1 acceptors are faulty, and three steps otherwise. The OTC framework version with $m = f$ and $q_2 = q_3 = f$ has the same properties.

- **DGV.** The restriction $m = f$ assumed in [121] was removed in the DGV algorithm by Dutta et al. [36]. Their algorithm requires $n > f + 2m + 2q_1$ and $n > 2f + m + \min\{m, q_1\}$, which corresponds to the requirements of the OTC approach with $q_2 = q_3 = f$.

A new algorithm can be obtained:

- **Ultimate Paxos.** The algorithm tolerates malicious processes. In timely runs with a correct first round coordinator, it decides in one step if at most q_1 acceptors are faulty, in two steps if at most q_2 are faulty, and in three otherwise. For the first round, the algorithm uses multi-step OTC with $q_3 = f$, which requires $n > f + 2m + 2q_1$, $n > f + m + q_2 + \min\{m, q_1\}$, and $n > 2f + m$. In the special case $q_2 = f$, it has the same properties as DGV [36].

2.7 Cheap OTC

In Section 2.5, we argued that since implementing Consensus requires $n > 2f + m$, it does not make sense to optimize OTC implementations to behave correctly for $n \leq 2f + m$. Consider crash-stop settings as an example. Although one-step single-value OTC used for the first round requires only $n > f + q_1$, it does not make sense to consider $q_1 < f$ because the other rounds require $n > 2f$ anyway.

Lamport and Massa [80] realized that this reasoning can lead to a waste of resources. In their Cheap Paxos algorithm, they suggested dividing the set of acceptors into primary ones and auxiliary ones. Primary acceptors are used continuously during the algorithm, whereas the auxiliary ones participate only when failures happen. In the OTC framework, this corresponds to the first round having access only to the primary acceptors. The other rounds, which are executed only when the first round does not decide, have access to both primary and auxiliary acceptors.

Consider the crash-stop example again. We can use the single-value one-step OTC with $q_1 < f$ for the first round. This will require $n > f + q_1$, where n is the number of primary acceptors. As before, the other rounds require $n' > 2f$, where n' is the total number of acceptors. In this section, we will construct *cheap OTC* algorithms that minimize the number n of primary acceptors.

Decision in one communication step

How many acceptors do we need to implement one-step OTC? Theorem 2.8.2 states that any such algorithm requires $n > f + 2m + q_1$. To minimize the number n of acceptors, we assume $q_1 = 0$, getting $n > f + 2m$. Theorem 2.8.3 states that any OTC algorithm satisfying Optimistic Termination $(q_\bullet, \bullet)$ requires $n > f + m + q_\bullet$. Therefore, to maintain the $n > f + 2m$ requirement, we must assume $q_\bullet \leq m$.

It follows that the strongest Optimistic Termination condition we can achieve under the assumption $n > f + 2m$ is

> **Optimistic Termination** $(0, 1)$ **and** $(m, 2)$**.** If all correct acceptors propose x and none of them executes *stop*, then
>
> 1. If all acceptors are correct, then all correct learners decide on x in one step.
> 2. If at most m acceptors are faulty, then all correct learners decide on x in two steps.

This property is satisfied by the multi-step multi-value OTC implementation from Section 2.6 with $q_1 = 0$ and $q_2 = q_3 = m$, which requires $n > f + 2m$.

Decision in two communication steps

Theorem 2.8.3 states that any OTC algorithm requires $n > f + m + q_\bullet$. To minimize the number n of acceptors, we assume $q_\bullet = 0$, getting $n > f + m$. It follows that the strongest Optimistic Termination condition we can achieve under the assumption $n > f + m$ is

> **Optimistic Termination** $(0, 2)$**.** If all acceptors are correct, propose x, and none of them executes *stop*, then *decision*(x) will hold at all correct learners after two communication steps.

This property is satisfied by the two-step multi-value OTC implementation from Section 2.4 with $q_2 = 0$, which requires $n > f + m$.

Summary

In this section, we have presented two multi-value cheap OTC algorithms with the following parameters

Algorithm	Condition	q_1	q_2
one-step	$n > f + 2m$	0	m
two-step	$n > f + m$	—	0

In both cases, we justified their optimality, even as single-value OTC algorithms. They can be used to reconstruct the following algorithm:

- **Cheap Paxos.** The Cheap Paxos algorithm by Lamport and Massa [80] assumes no malicious processes. It requires only $n > f$ primary acceptors, and $n' > 2f$ acceptors in total. It decides in two steps if all acceptors are correct.

  In the OTC framework, we use one-step cheap OTC with $m = 0$ for the first round, and one-step single-value OTC with $q = f$ for the others. These two OTC implementations require $n > f$ and $n' > 2f$, respectively, the same as Cheap Paxos [80].

The following new Consensus algorithms can be constructed:

- **Cheap Byzantine Paxos.** This algorithm can be thought of as a Byzantine generalization of Cheap Paxos. It requires $n > f+2m$ primary acceptors, and $n' > 2f+m$ acceptors in total. It decides in two steps if all acceptors are correct, and in three steps if at most m of them are faulty. We use one-step cheap OTC for the first round, and multi-value two-step OTC for the others.

- **Supercheap Byzantine Paxos.** This algorithm requires $n > f + m$ primary acceptors and $n' > 2f + m$ acceptors in total. If all acceptors are correct, it decides in three communication steps. We use two-step cheap OTC for the first round and two-step multi-value OTC for the others.

2.8 Lower bounds

In this section, we will prove four theorems, which show that the number of acceptors required by the OTC implementations from this chapter cannot be improved. The table below presents a brief summary of the results:

Theorem	Opt. Termination	Proposals	Necessary condition
Theorem 2.8.1	$(q_1, 1)$	single-value	$n > f + \ q_1 + 2m$
Theorem 2.8.2	$(q_1, 1)$	multi-value	$n > f + 2q_1 + 2m$
Theorem 2.8.3	$(q_\bullet, \bullet)$	single-value	$n > f + \ q_\bullet + \ m$
Theorem 2.8.4	$(q_1, 1)$ and $(q_2, 2)$	multi-value	$n > f + \ q_2 + \ m + \min\{q_1, m\}$

All the proofs share a similar structure. We assume there is an OTC algorithm that does not require the given condition. Then, we construct a sequence of runs, such that in at least one of them the algorithm behaves incorrectly. All runs are illustrated with standard diagrams, using the following symbols:

Symbol	Meaning
	process crashes
	process freezes for a period of time
	process is malicious
	process executes *stop*

When a process *freezes* for a period of time, all outgoing messages, except those explicitly mentioned, are blocked at that process until this period finishes. For clarity, the diagrams show only those messages that are important to understand the proofs.

As required by the definition of the number of communication steps from Section 1.4, the Optimistic Termination properties do not assume any particular time metric. Therefore, to show that a given number of communication steps cannot be achieved under given conditions, it is sufficient to show the impossibility for a single time metric. We consider real time with all messages having the same latency d, unless stated otherwise. We assume that the system contains at least three learners (l_1, l_2, l_3).

To provide stronger results, the proofs in this section assume a weaker version of the Optimistic Termination conditions which additionally assumes that no honest process proposes anything other than x.

Theorem 2.8.1. *Any single-value OTC algorithm satisfying Opt. Termination* $(q_1, 1)$ *requires* $n > f + q_1 + 2m$.

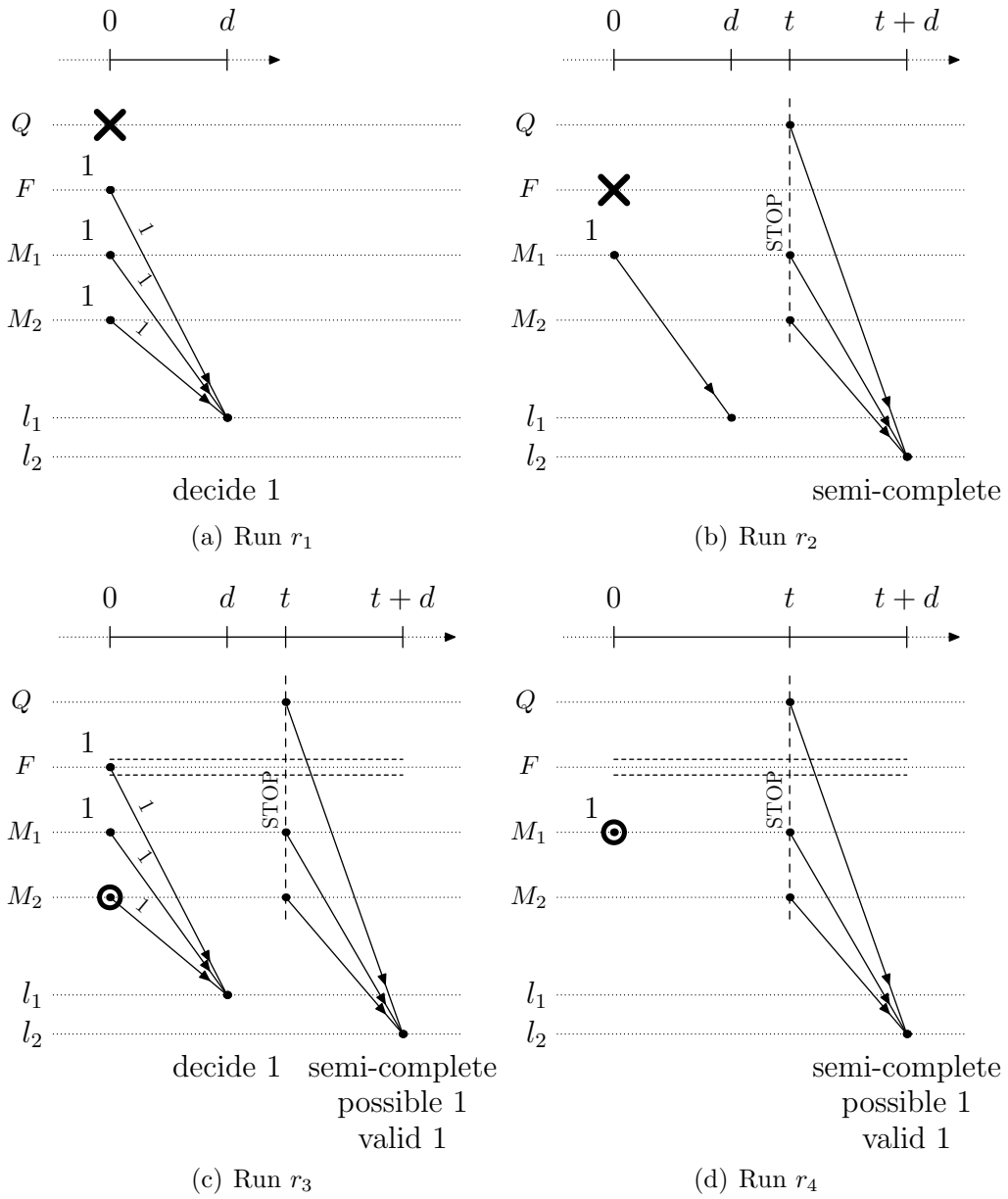
Proof. To obtain contradiction, consider a one-step single-value OTC algorithm with $n \leq f + q_1 + 2m$. Figure 2.12 shows four runs of this algorithm. Acceptors have been divided into four groups: Q, F, M_1, M_2, with sizes of at most q_1, f, m, m, respectively. In all runs, all acceptors from the same group behave identically.

In run r_1, acceptors in Q crash at time 0, and all the other acceptors are correct and propose 1. Since at most q_1 acceptors failed, Optimistic Termination $(q_1, 1)$ requires learner l_1 to decide in one communication step (by time d).

In run r_2, all acceptors are correct, except for those in F, which crash at the beginning. Only acceptors in group M_1 propose 1, the others do not propose anything. At some time $t > d$, all correct acceptors execute *stop*. At time $t + d$, learner l_2 has received all messages sent by correct acceptors at time t or before. Permanent Validity and Permanent Agreement imply that l_2 is semi-complete (Section 2.2).

Run r_3 is identical to r_2, except for two changes. Firstly, acceptors in F are correct, propose 1, send a message to l_1, and immediately freeze until time $t + d$. Acceptors in M_2 are malicious and send a message to l_1 claiming that they proposed 1, whereas in fact they did not propose anything. Apart from that, acceptors M_2 behave correctly.

At time d, learner l_1 cannot distinguish r_3 from r_1, so it decides on 1. Consider the state of learner l_2 at time $t + d$. Predicate *possible*(1) holds because learner l_1 decided



(a) Run r_1 (b) Run r_2

(c) Run r_3 (d) Run r_4

Figure 2.12: Runs examined in the proof of Theorem 2.8.1

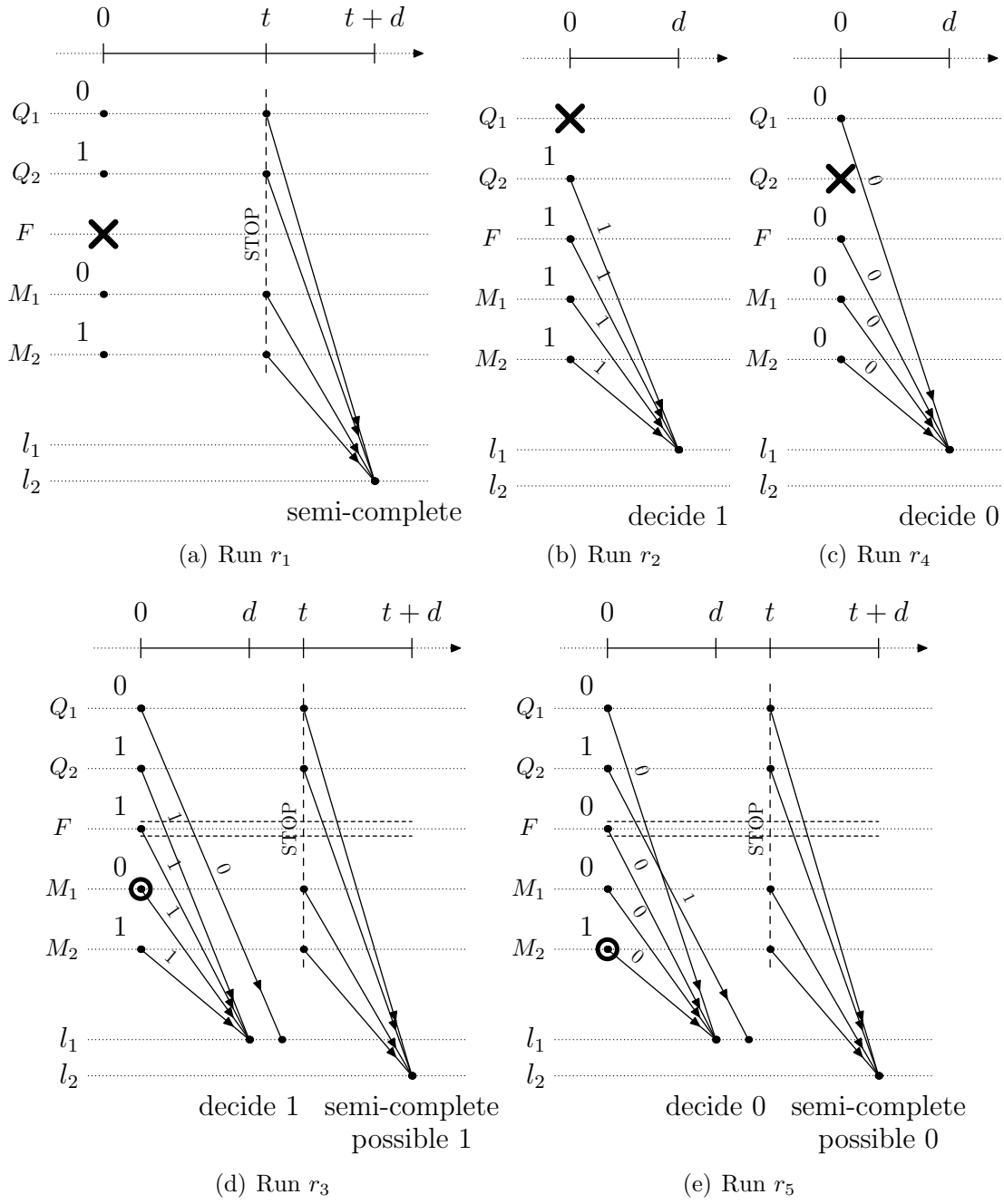


Figure 2.13: Runs examined in the proof of Theorem 2.8.2

on 1 (Possibility). Learner l_2 cannot distinguish r_3 from r_2, so its state is semi-complete. This implies that $possible(1) \implies valid(1)$, so $valid(1)$ holds as well.

Finally, in run r_4, all acceptors are correct except for those in group M_1. No acceptor proposes anything, but acceptors in M_1 are malicious and they behave as if they had proposed 1. Acceptors in F freeze from time 0 to $t + d$, and all other acceptors execute *stop* at time t. At time $t + d$, learner l_2 cannot distinguish runs r_4 and r_3, so $valid(1)$ holds. This violates Integrity, because no (honest) acceptor proposed 1 in this run. □

Theorem 2.8.2. *Any multi-value OTC algorithm satisfying Opt. Termination* $(q_1, 1)$ *requires* $n > f + 2q_1 + 2m$.

Proof. To obtain contradiction, consider a one-step multi-value OTC algorithm with $n \leq f + 2q_1 + 2m$. Figure 2.12 shows five runs of the algorithm. Acceptors have been divided into five groups: Q_1, Q_2, F, M_1 and M_2 with sizes of at most q_1, q_1, f, m, and m, respectively. In all runs, all acceptors from the same group behave identically.

In run r_1, all acceptors are correct, except for those in group F, which crash at time 0. Acceptors Q_1 and M_1 propose 0, whereas acceptors in Q_2 and M_2 propose 1. At some time $t > d$, all correct acceptors execute *stop*. At time $t + d$, learner l_2 has received all messages sent by correct acceptors at time t or before. Permanent Validity and Permanent Agreement imply that its state is semi-complete (Section 2.2).

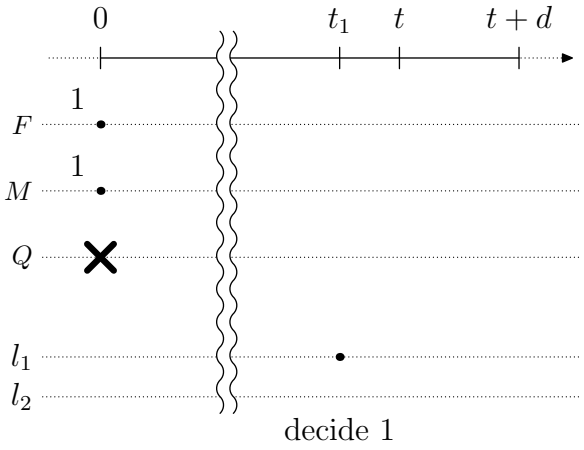
In run r_2, all acceptors are correct and propose 1, except for those in group Q_1, which crash at time 0 without proposing anything. Optimistic Termination $(q_1, 1)$ requires learner l_1 to decide on 1 in one communication step, that is, by time d.

In run r_3, all acceptors are correct, except for those in M_1, which are malicious. Acceptors in Q_1 and M_1 propose 0, whereas the other acceptors propose 1. The message from Q_1 to l_1 is delayed and arrives at l_1 just after time d. Immediately after proposing, acceptors in F send a message to l_1 and freeze until time $t + d$. Malicious acceptors M_1 send a message to l_1 claiming they had proposed 1, otherwise they behave correctly. At time t, all acceptors execute *stop*, except for those in group F. At time d, learner l_1 cannot distinguish run r_3 from r_2, so it decides on 1. At time $t + d$, learner l_2 cannot distinguish run r_3 from r_1, so it enters a semi-complete state. Predicate *possible*(1) holds because learner l_1 decided on 1 (Possibility).

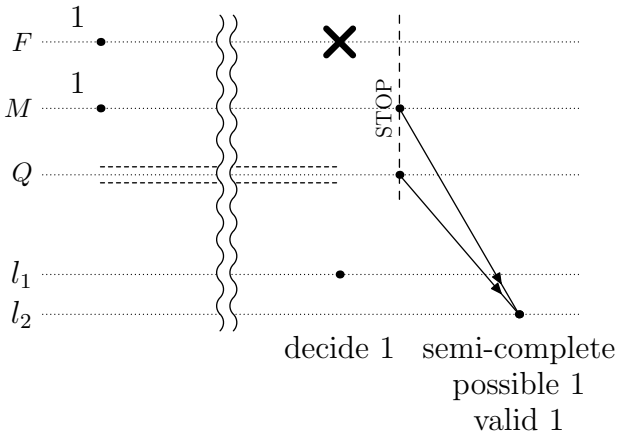
In run r_4, all acceptors are correct and propose 0, except for those in group Q_2, which crash at time 0 without proposing anything. Optimistic Termination $(q_1, 1)$ requires learner l_1 to decide on 0 by time d.

In run r_5, all acceptors are correct, except for those in M_2, which are malicious. Acceptors in Q_2 and M_2 propose 1, whereas the other acceptors propose 0. Message from Q_2 to l_1 is delayed and arrives at l_1 just after time d. Immediately after proposing, acceptors in F send a message to l_1 and freeze until time $t + d$. Malicious acceptors M_2 send a message to l_1 claiming they had proposed 0, otherwise they behave correctly. At time t, all acceptors execute *stop*, except for those in group F. At time d, learner l_1 cannot distinguish run r_5 from r_4, so it decides on 0. At time $t + d$, learner l_2 cannot distinguish run r_5 from r_1, so it enters a semi-complete state. Predicate *possible*(0) holds because learner l_1 decided on 0.

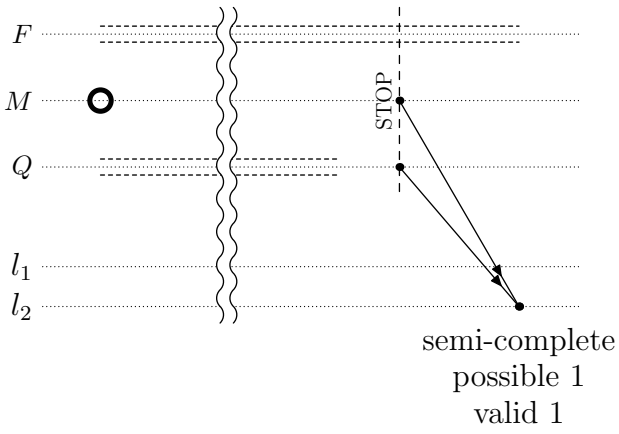
At time $t + d$ learner l_2 cannot distinguish runs r_4 from r_5, so in both cases it is in a semi-complete state with both *possible*(0) and *possible*(1) holding. This violates the definition of semi-completeness. □



(a) Run r_1.



(b) Run r_2.



(c) Run r_3.

Figure 2.14: Runs examined in the proof of Theorem 2.8.3

Theorem 2.8.3. *Any single-value OTC algorithm satisfying Opt. Termination* $(q_\bullet, \bullet)$ *requires* $n > f + q_\bullet + m$.

Proof. To obtain contradiction, consider a single-value OTC algorithm with $n \leq f+m+q_\bullet$. Figure 2.14 shows three runs of this algorithm. Acceptors have been divided into three

groups: F, M, Q with sizes of at most f, m, $q_\bullet$, respectively. In all runs, all acceptors from the same group behave identically.

In run r_1, all acceptors are correct and propose 1, except for those in group Q, which crash at time 0 and propose nothing. Optimistic Termination $(q_\bullet, \bullet)$ requires that learner l_1 eventually decides on 1, say at time t_1.

In run r_2, acceptors F and M propose 1. Acceptors in group Q freeze from time 0 to t_1 without proposing anything. Acceptors in F crash at time t_1; all their messages to processes Q and l_2 are lost. At time t_1, learner l_1 cannot distinguish r_2 from r_1, so it decides on 1. At some time $t > t_1$, acceptors in M and Q execute *stop*. At time $t + d$, learner l_2 has received all messages sent by correct acceptors at time t or before. Permanent Validity and Permanent Agreement imply that its state is semi-complete. Predicate *possible*(1) holds at l_2 because l_1 decided on 1, and semi-completeness implies that *valid*(1) holds as well.

In run r_3, acceptors in M are malicious and all the others are correct. No acceptors propose anything. Acceptors in F freeze from time 0 to $t + d$, and those in Q from time 0 to t_1. Malicious acceptors M behave as if they had proposed 1 and received exactly the same messages from acceptors F as in run r_2; otherwise they are correct. At time t, acceptors M and Q execute *stop*.

At time $t + d$, learner l_2 cannot distinguish runs r_3 from r_2, so *valid*(1) holds. This violates Integrity, as no acceptor proposed anything in this run. □

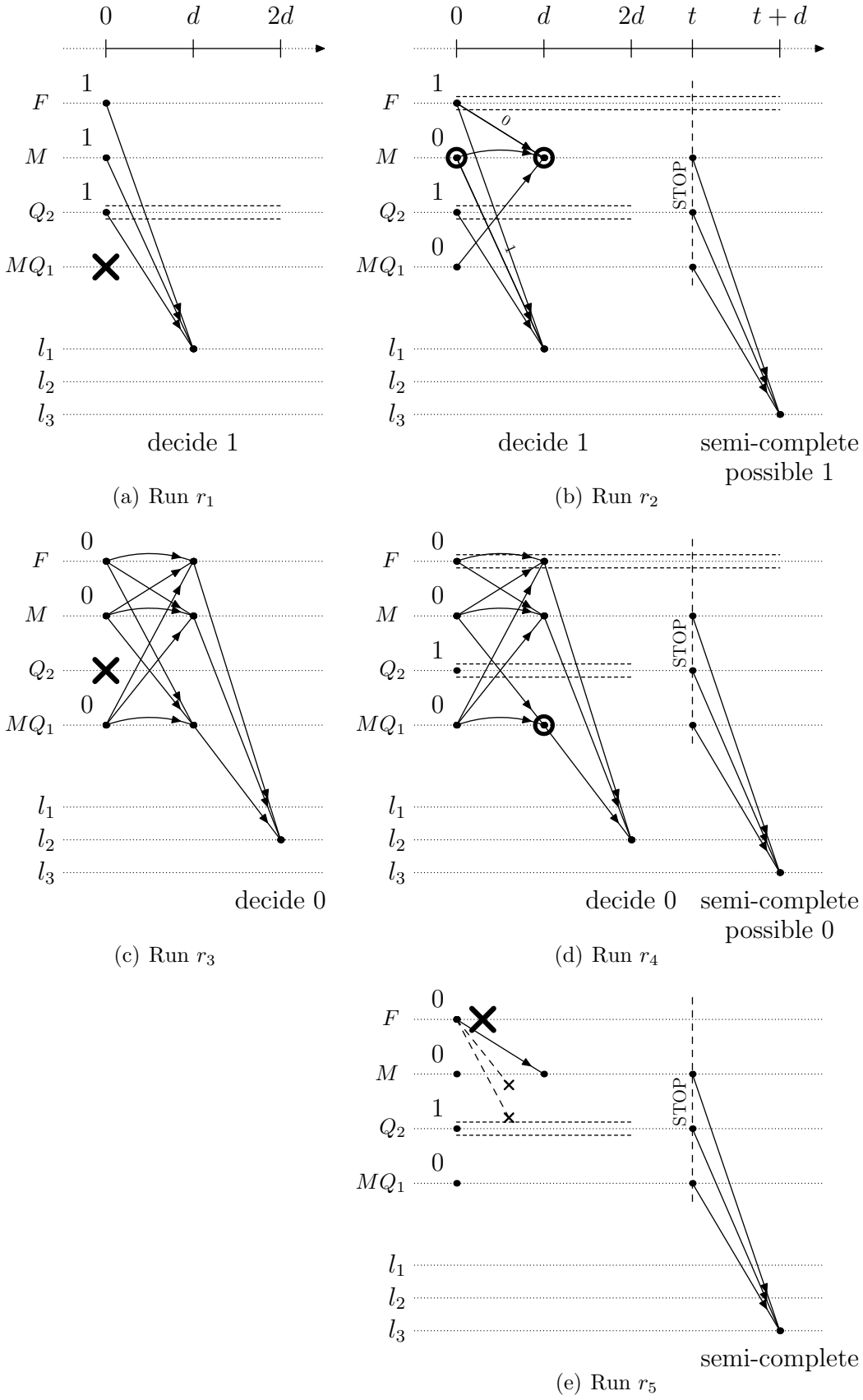
Theorem 2.8.4. *Any OTC algorithm satisfying Opt. Termination* $(q_1, 1)$ *and* $(q_2, 2)$ *requires* $n > f + m + q_2 + \min\{q_1, m\}$.

Proof. To obtain contradiction, consider an OTC algorithm satisfying Optimistic Termination $(q_1, 1)$ and $(q_2, 2)$ with $n \leq f + m + q_2 + \min\{q_1, m\}$. Figure 2.15 shows five runs of this algorithm. Acceptors have been divided into four groups: F, M, Q_2 and MQ_1 with sizes of at most f, m, q_1, and $\min\{m, q_1\}$, respectively. In all runs, all acceptors from the same group behave identically.

In run r_5, all acceptors are correct, except those in F who crash immediately after sending a message to acceptors M. All acceptors propose 0 except those in group Q_2, who propose 1 and immediately freeze until time $2d$. At some time $t > 2d$, all correct acceptors execute *stop*. As a result, Permanent Validity and Permanent Agreement imply that the state of learner l_3 at time $t + d$ is semi-complete.

In run r_1, all acceptors are correct and propose 1, except for those in MQ_1, who crash at time 0 without proposing anything. Acceptors Q_2 send a message to learner l_1 then freeze from time 0 to $2d$. Optimistic Termination $(q_1, 1)$ makes l_1 decide on 1 by time d.

In run r_2, acceptors F and Q_2 propose 1, send a message to l_1 and freeze until times $t + d$ and $2d$, respectively. Other acceptors propose 0. Acceptors M are malicious; to



(a) Run r_1

(b) Run r_2

(c) Run r_3

(d) Run r_4

(e) Run r_5

Figure 2.15: Runs examined in the proof of Theorem 2.8.4

l_1 they pretend to have proposed 1, and they behave as if they had received 0 from acceptors in F. Otherwise they behave correctly. At time t, all acceptors except for F execute *stop*. By time d, learner l_1 cannot distinguish runs r_1 and r_2, so it decides on 1 in both of them. By time $t + d$, learner l_3 cannot distinguish r_2 and r_5, so it enters a semi-complete state. Predicate *possible*(1) holds because l_1 decided on 1 (Possibility).

In run r_3, acceptors in Q crash at time 0. All other acceptors are correct and propose 0. Optimistic Termination $(q_2, 2)$ makes learner l_2 decide on 0 by time $2d$.

Run r_4 is similar to r_2, except that acceptors in Q are correct, propose 1, and freeze until time $2d$. Acceptors in F freeze from time 0 to $t + d$, however, they send a message to M at time 0, and a message to l_2 at time 1. Acceptors in MQ_1 are malicious and pretend to l_2 that at time 1 they had received a message from F, as in run r_3. At time t, all acceptors except for F execute *stop*. At time $2d$, learner l_2 cannot distinguish r_3 from r_4, and decides on 0 in both of them. At time $t + d$, learner l_3 cannot distinguish runs r_4 and r_5, so it is in a semi-complete state. Predicate *possible*(0) holds because learner l_2 decided on 0.

Learner l_3 cannot distinguish runs r_2, r_4, and r_5. Therefore, in all of them, it is in a semi-complete state with both *possible*(0) and *possible*(1) holding, which violates the definition of semi-completeness. □

2.9 Conclusion

This chapter introduced Optimistically Terminating Consensus (OTC), an abstraction that represents an individual round of a Consensus protocol. A sequence of OTC instances can be executed one after another to implement Consensus (see Chapter 4 for details). Choosing different implementations of OTC leads to Consensus algorithms with different properties. Since in "favourable" runs Consensus decides in the first round, the latency of a Consensus protocol in such runs is fully determined by that of the first round OTC.

The OTC abstraction can be thought of as a variant of Consensus that is required to decide only if all correct acceptors proposed the same value, which resembles condition-based Consensus [91]. Unlike that abstraction, OTC instances are designed to be combined into full Consensus protocols. For this reason, they guarantee Permanent Validity and Permanent Agreement, which are stronger than their standard counterparts, because they operate on "possible decisions" and "provable proposals" rather than just on real decisions and real proposals.

OTC is easy to implement, even with malicious acceptors; the learners decide on a given value if a sufficient number of acceptors report to have proposed it (Section 2.3). This simple one-step Generic Agreement implementation is sufficient to match the latency and the required number of acceptors of a large number of Consensus algorithms for the

Algorithm	q_1	q_2	q_3	Condition	All benign	All malicious
one-step						
single-value	q	—	—	$n > f + 2m + q$	$n > f + q$	$n > 3f$
multi-value	q	—	—	$n > f + 2m + 2q$	$n > f + 2q$	$n > 3f + 2q$
two-step						
multi-value	—	f	—	$n > 2f + m$	$n > 2f$	$n > 3f$
multi-step						
single-value	q_1	q_2	f	$n > 2f + m$ $n > f + 2m + q_1$	$n > 2f$	$n > 3f + q_1$
multi-value	q_1	q_2	f	$n > f + m + q_2 + \min\{m, q_1\}$ $n > f + 2m + 2q_1$ $n > 2f + m$	$n > 2f$ $n > f + 2q_1$	$n > 3f + 2q$
cheap						
multi-value	0	m	—	$n > f + 2m$	$n > f$	$n > 3f$
multi-value	—	0	—	$n > f + m$	$n > f$	$n > 2f$

Figure 2.16: Summary of OTC algorithms presented in this chapter.

crash-stop model [13, 16, 63, 73, 80, 112] as well as the Byzantine model [41, 71, 87]. Combining several instances of one-step Generic Agreement leads to new OTC implementations, which match the latency and acceptor requirements of other Consensus algorithms [15, 36, 121]. New algorithms can be obtained, such as Ultimate Paxos, one-step Byzantine Consensus and two variants of Cheap Byzantine Paxos. Figure 2.16 summarizes the requirements of various OTC implementations presented in this chapter. The theorems presented in Section 2.8 prove that all these requirements are optimal.

Consensus algorithms, especially those for the Byzantine model, are notoriously difficult to design, understand, and prove correct. I believe that the OTC abstraction makes this task much easier. The full comparison between OTC and other agreement frameworks will be given in Chapter 4. For the moment, note the three characteristics that distinguish OTC from similar abstractions [10, 11, 12, 51, 65, 92]: (i) tolerating malicious processes, (ii) full self-containment, and (iii) implementability in purely asynchronous settings. These three properties make the OTC abstraction more modular and allow us to implement a much wider range of agreement protocols than any of the previous approaches.

Chapter 3

Automatic discovery of OTC protocols

Chapter 2 introduced the concept of Optimistically Terminating Consensus and briefly explained how to use it to implement Consensus. This modular way of implementing Consensus and agreement abstractions (Chapter 4) has several advantages over constructing such algorithms from scratch. Firstly, the same OTC algorithm can be used in various agreement protocols (reusability). Secondly, OTC algorithms are conceptually simpler than those implementing Consensus, and as a result, proving their correctness is also easier. In this chapter, we will show how to mechanically verify the correctness of OTC algorithms. By searching the space of OTC algorithms and filtering incorrect ones out, we will be able to discover new algorithms automatically.

Automatic correctness testing of individual OTC algorithms is also helpful in designing OTC algorithms manually. Our tool not only quickly verifies the correctness of candidate algorithms, but also shows the scenarios in which incorrect algorithms fail. This allows us to actually understand why a given algorithm is incorrect. Such understanding can lead to new impossibility results, such as those from Chapter 2, which have been obtained with the aid of our OTC verification tool.

The verification method presented in this chapter assumes a particular structure of OTC algorithms. As a result, some correct OTC algorithms might not be expressible in our model. We believe, however, that the simplifications we make do not exclude any "sensible" OTC algorithms. In particular, all OTC algorithms presented in Chapter 2 are expressible in our model.

Our method requires the number of acceptors to be known in advance. In other words, it can verify an algorithm for, say, four acceptors, but is unable to verify a general solution for n acceptors. Nevertheless, the method presented in this chapter is still useful for discovering general OTC algorithms. It can first be applied to generate OTC algorithms for consecutive numbers of acceptors, such as 3, 4, and 5. Then, we found out, it is usually not difficult for a human to spot a pattern and generalize this sequence of algorithms for

fixed n's into a general OTC algorithm that takes n as a parameter. The same technique can be used to obtain lower bounds. First, we use the tool to understand why given requirements cannot be met for a specific n, and then generalize our observations into a lower bound theorem.

This chapter is structured in the following way. Section 3.1 introduces our execution model and provides precise definitions of two fundamental concepts: *events* and *states*. To reason about them, Section 3.2 develops a set-theoretic formalism, which is then used in Section 3.3 to define predicates $valid(x)$, $possible(x)$, and $decision(x)$ that satisfy OTC properties Integrity, Possibility, and Optimistic Termination. Section 3.4 describes a method of checking whether these predicates satisfy the other two OTC properties: Permanent Validity and Permanent Agreement, thereby verifying correctness of a given OTC algorithm. Section 3.5 uses this method to search for new OTC algorithms.

Related work

Although automatic reasoning about protocols is common in security [14, 22, 84, 88, 96], not much related work has been done in the area of agreement algorithms. Paxos [73] seems to be the only asynchronous agreement protocol to have undergone a significant amount of formal analysis. The algorithm itself has been specified in TLA$^+$ [75] and in the General Timed Automaton model [106]. TLA$^+$ has also been used to give formal specifications of Disk Paxos [42] and Paxos Commit [45]. Win and Ernst [118] and later Win et al. [119] used the Larch theorem prover [43] to formally show the correctness of the Paxos algorithm. Kellomaki [68] obtained the same result with PVS [94].

Bar-David and Taubenfeld [8] used a combination of model checking and program generation to automatically discover new mutual exclusion algorithms [85]. Apart from this, we are not aware of any previous attempt at automatic discovery of distributed algorithms.

3.1 Execution model

We assume the same system model as in the previous chapters: a network of processes communicating using asynchronous reliable channels. This means that channels do not create or modify messages. All messages between correct processes are eventually delivered, but there are no bounds on message transmission times.

We assume that the system consists of an unlimited number of honest learners and a fixed number n of possibly faulty acceptors $a_1, \ldots, a_n$. In the OTC abstraction (Section 2.2), acceptors issue proposals and collaborate in order for the learners to decide on one of these proposals. Acceptors can perform two kinds of actions: issue a proposal by

executing *propose*(x), and stop the algorithm by executing *stop*. Learners have access to three predicates: *decision*(x), *possible*(x), and *valid*(x).

The number of possible OTC algorithms is large, and in order to be able to represent them efficiently, we need to put some restrictions on the algorithms we will be considering. At the same time, we have to make sure that during this process we will not omit any "sensible" algorithms.

3.1.1 Messages

We consider only full-information protocols, in which each message contains the entire state of the sender. This assumption involves no loss of generality because the information present in such messages allows the recipient to reconstruct any other messages that could have been sent by the sender. Sending entire states might increase the size of the messages, but does not affect latency. If message sizes are of importance, then the algorithm automatically found by our method can be later manually modified so that only relevant information is sent.

For example, assume that an acceptor's state consists of two variables x and y. The list of possible messages that can be sent by the acceptor includes:

$$\langle x \rangle, \quad \langle y \rangle, \quad \langle x + y \rangle, \quad \langle x * y, x + y, x - y \rangle, \quad \langle x, y \rangle.$$

Note that the information present in any of these messages can be deduced from the last message $\langle x, y \rangle$ that carries the entire state of the acceptor. However, if the recipients are interested only in $x + y$, sending $\langle x + y \rangle$ instead will reduce the size of the message.

In our model, we assume no bounds on process speeds or message transmission times, so sending the same state twice does not provide any new information. Therefore, we assume that acceptors broadcast their states only if they change. This can happen because of events such as receiving a message or executing an action such as *propose*(x) or *stop*. In fact, these kinds of events are the only ones possible in our model; we explicitly rule out non-determinism and real time clocks. In other words, we employ the diffusing computation model [30]; acceptors broadcast their states only immediately after receiving a message, stopping, or proposing a value, and remain idle otherwise.

3.1.2 States

We have already explained when a state of a process changes, but have not yet defined what a state is. Since a state changes only on events, it is natural to define it as a sequence of events that occurred at a given process, that is, received messages, and executions of *propose*(x) and *stop*.

For the reasons described in the next paragraph, we assume that the order of events does not matter. The only exception is the *stop* action; the notion of a "complete state" depends on messages sent by a correct acceptor *before or during* its first execution of *stop*. For the moment, we will ignore this problem by considering only runs in which no acceptor executes *stop*. We will return to this problem in Section 3.1.5.

Each Optimistic Termination property assumes that all correct acceptors propose the same value, and requires correct learners to decide on it. In other words, in the cases covered by Optimistic Termination properties, the value of the decision is uniquely determined by the proposals, and must be reached regardless of the order in which various events, such as message deliveries, occur. For this reason, we assume that the state of a process does not depend on the order of events it experienced. In other words, we assume that the state of a process is a *set* of events that occurred at that process, rather than a sequence.

Note that the above argument does not apply to the Consensus problem, where the Termination condition does not specify what decision should be made. In some cases, the decision cannot be determined from the proposals alone, and depends on the order of events. In fact, the existence of such cases is the very reason for Consensus impossibility in purely asynchronous systems [40].

3.1.3 Events

The observation that the order of events does not matter allows us to further restrict the space of OTC algorithms to consider. Instead of broadcasting the whole state whenever it changes, we will assume that acceptors broadcast the event that caused the change. Since state changes are deterministic and the order of events does not matter, broadcasting only events results in no loss of information in comparision to broadcasting entire states. On the other hand, it has the advantage of being simpler to model and the messages being more compact.

Since we ignore *stop* in this section, an *event* is either a *propose* action or a message reception. The previous paragraph established that messages are actually events that caused the state change. Therefore, *message reception* consists of the received event and the acceptor at which this event occurred. In other words,

$$\begin{aligned} \mathit{event} &= \mathit{message\ reception} \quad \text{or} \quad \mathit{propose}(x) \\ \mathit{message\ reception} &= \mathit{event} : \mathit{acceptor} \end{aligned}$$

Unfolding this recursive definition yields:

$$\mathit{event} = \langle x : e_1 e_2 \dots e_k \rangle,$$

```
1   state ← ∅
2   when an acceptor executes propose(x) do
3     incorporate ⟨x : ε⟩ into the state
4   when a process receives ⟨x : e1e2 ... ek−1⟩ from acceptor ek do
5     incorporate ⟨x : e1e2 ... ek⟩ into the state
6   action incorporate ⟨x : e1e2 ... ek⟩ into the state do
7     if ⟨y : e1e2 ... ek⟩ ∉ state for any y then
8       insert ⟨x : e1e2 ... ek⟩ into state
9       if the current process p is an acceptor do
10        broadcast ⟨x : e1e2 ... ek⟩
11  when an acceptor executes stop do
12    for all sequences e1e2 ... ek do          { including the empty sequence ε }
13      incorporate ⟨⊤ : e1e2 ... ek⟩ into the state
```

Figure 3.1: Algorithm describing the evolution of states.

where x is a proposed value and $e_1e_2 \dots e_k$ is a list of acceptors. The event $\langle x : \varepsilon \rangle$, where ε is the empty list of acceptors, corresponds to the action *propose*(x). The event $\langle x : e_1 \rangle$ corresponds to receiving a message from acceptor e_1 claiming that e_1 proposed x. Event $\langle x : e_1e_2 \rangle$ corresponds to receiving a message from e_2 that claims that it has received a message from e_1 claiming that e_1 proposed x. In general, event $\langle x : e_1e_2 \dots e_k \rangle$ corresponds to receiving a message from e_k claiming that event $\langle x : e_1e_2 \dots e_{k-1} \rangle$ occurred at e_k.

3.1.4 Evolution of states

We have already established that the state of a process is a set of events of the form $\langle x : e_1e_2 \dots e_k \rangle$. Figure 3.1 gives a detailed description of the evolution of states of honest processes. Each process starts with the empty set *state*. In lines 2–3, an acceptor proposing some value x incorporates the event $\langle x : \varepsilon \rangle$ into its state. Similarly, when a message containing the event $\langle x : e_1e_2 \dots e_{k-1} \rangle$ arrives from acceptor e_k, the process incorporates the event $\langle x : e_1e_2 \dots e_k \rangle$ into its state. We say that a particular event $\langle x : e_1e_2 \dots e_k \rangle$ *occurred* at a process if it belongs to its *state*.

A process *incorporates* an event into its state by adding it to *state*. In addition, acceptors broadcast the event to all processes, including themselves. This produces a never-ending exchange of events $\langle x : e_1e_2 \dots e_k \rangle$, with an arbitrarily large k. In practice, all events $\langle x : e_1e_2 \dots e_k \rangle$ with k larger than those in the Optimistic Termination requirements are ignored. For example, when verifying an OTC algorithm for Optimistic Termination $(q_1, 1)$ and $(q_2, 2)$, we ignore all events $\langle x : e_1e_2 \dots e_k \rangle$ with $k > 2$.

The **if** statement in line 7 ignores $\langle x : e_1e_2 \dots e_k \rangle$ if another event $\langle y : e_1e_2 \dots e_k \rangle$ is already in the state. We argue that ignoring such events does not limit the generality

of our algorithm. A process might try to incorporate two events $\langle x : e_1 e_2 \dots e_k \rangle$ and $\langle y : e_1 e_2 \dots e_k \rangle$ with $x \neq y$ in two cases. In the first case, the list of the acceptors in both events is empty, that is, the events are $\langle x : \varepsilon \rangle$ and $\langle y : \varepsilon \rangle$. This means that the process is an acceptor that executed both *propose*(x) and *propose*(y), which contradicts the assumption that honest acceptors issue at most one proposal.

If the events $\langle x : e_1 e_2 \dots e_k \rangle$ and $\langle y : e_1 e_2 \dots e_k \rangle$ have a non-empty sequence of acceptors $e_1 e_2 \dots e_k$, then the process must have received messages $\langle x : e_1 e_2 \dots e_{k-1} \rangle$ and $\langle y : e_1 e_2 \dots e_{k-1} \rangle$ from acceptor e_k. This means that e_k has incorporated both events into its state, which is precisely what the **if** instruction in line 7 prevents. Therefore, e_k must be a malicious acceptor; its messages convey no useful information and can safely be ignored.

3.1.5 Action *stop*

Until now, we have assumed that no acceptor executes *stop*. As explained in Section 2.2, we can assume that executing *stop* leaves an acceptor in a final state, a state in which no further event can occur. In our model, this means that the value of *state* after executing *stop* must contain an event of the form $\langle x : e_1 e_2 \dots e_k \rangle$ for every sequence $e_1 e_2 \dots e_k$; otherwise, the event $\langle x : e_1 e_2 \dots e_k \rangle$ could still occur in the future.

Modelling the action *stop* as a special kind of event would complicate our set-theoretic model of states. Instead, lines 6–10 emulate *stop* in our current model by trying to incorporate events $\langle \top : e_1 e_2 \dots e_k \rangle$ for all possible sequences $e_1 e_2 \dots e_k$, where $\top$ is a symbol outside the set of possible proposals. This adds to *state* all events $\langle \top : e_1 e_2 \dots e_k \rangle$ for which no event of the form $\langle x : e_1 e_2 \dots e_k \rangle$ belongs to *state*. After this operation, *state* is final because it contains an event of the form $\langle x : e_1 e_2 \dots e_k \rangle$ for every sequence $e_1 e_2 \dots e_k$.

3.1.6 Summary

In this section, we have shown how general assumptions about the OTC protocols we are interested in lead to a specific algorithm describing the evolution of process states. We concluded that the behaviour of any OTC implementation can be described by the algorithm in Figure 3.1. In brief, acceptors broadcast their proposals and relay messages from other acceptors, adding their own identifiers and suppressing clearly malicious messages. Our model supports acceptors' actions *propose*(x) and *stop*. In Section 3.3, we will show how to define learners' predicates *decision*, *possible*, and *valid*.

3.2 State formalism

In this section, we will consolidate our knowledge of events and process states, and introduce a set-theoretic formalism for dealing with these.

3.2.1 Events

Recall that an event is a pair $\langle x : \alpha \rangle$, where x is a value and α is a sequence of acceptors. The meaning of $\langle x : \alpha \rangle$ is defined recursively. The event $\langle x : \varepsilon \rangle$ occurring at some process means that process proposed x. The event $\langle x : e_1 e_2 \ldots e_k \rangle$ means that the process received a message from acceptor e_k, which claims that the event $\langle x : e_1 e_2 \ldots e_{k-1} \rangle$ occurred at e_k.

Two events *conflict* if they have different proposal values and the same sequence of acceptors. In other words, events $\langle x : \alpha \rangle$ and $\langle y : \beta \rangle$ conflict iff $x \neq y$ and $\alpha = \beta$. For example, events $\langle 1 : a_1 a_3 \rangle$ and $\langle 2 : a_1 a_3 \rangle$ conflict, whereas $\langle 1 : a_1 a_3 \rangle$ and $\langle 2 : a_1 a_2 \rangle$ do not.

3.2.2 States

We define a state to be *any* set of events of the form $\langle x : \alpha \rangle$. It does not have to correspond to the state of any particular process and can contain conflicting events. States that do not contain any conflicting events are called *pure*. In Section 3.1, we explained why states of individual honest processes are always pure.

For example,

$$S_1 = \{\langle 1 : a_1 a_2 \rangle, \langle 2 : a_2 \rangle\}, \quad S_2 = \{\langle 2 : a_1 a_2 \rangle, \langle 2 : a_2 \rangle\}.$$

are both pure states because none of them contains conflicting events. However, the state

$$S_1 \cup S_2 = \{\langle 1 : a_1 a_2 \rangle, \langle 2 : a_1 a_2 \rangle, \langle 2 : a_2 \rangle\}$$

is not pure because events $\langle 1 : a_1 a_2 \rangle$ and $\langle 2 : a_1 a_2 \rangle$ conflict.

The *conflict* operator

For any state S, we define $\mathit{conflict}(S)$ to be the set of sequences α, for which some events $\langle z : \alpha \rangle \in S$ conflict:

$$\mathit{conflict}(S) \stackrel{\text{def}}{=} \{\, \alpha \mid \exists\, x \neq y : \langle x : \alpha \rangle \in S \wedge \langle y : \alpha \rangle \in S \,\}.$$

In our example,

$$\mathit{conflict}(S_1) = \mathit{conflict}(S_2) = \emptyset \quad \text{and} \quad \mathit{conflict}(S_1 \cup S_2) = \{a_1 a_2\}.$$

Using the $\mathit{conflict}(S)$ operator, we can give an alternative definition of a pure state:

$$\text{state } S \text{ is pure} \quad \Longleftrightarrow \quad \mathit{conflict}(S) = \emptyset.$$

From sequences to states

To make specifying states easier, we will extend the event notation $\langle x : \alpha \rangle$ to allow the parameter α to be a *set* of sequences. For any set of sequences X, the symbol $\langle x : X \rangle$ denotes a state consisting of all events $\langle x : \alpha \rangle$ with $\alpha \in X$. Formally,

$$\langle x : X \rangle = \{ \langle x : \alpha \rangle \mid \alpha \in X \}.$$

For example, state S_2 can be expressed as

$$S_2 = \{\langle 2 : a_1 a_2 \rangle, \langle 2 : a_2 \rangle\} = \langle 2 : \{a_1 a_2, a_2\} \rangle.$$

Note that states of the form $\langle x : X \rangle$ are always pure because events with the same x cannot conflict. In other words,

$$\mathit{conflict}(\langle x : X \rangle) = \emptyset.$$

From states to sequences

In many cases, we will be interested in dealing not with entire states, but with their subsets consisting only of events with a particular proposal value. For this reason, for any state S, we define $S(x)$ to be the set of sequences α such that the event $\langle x : \alpha \rangle$ belongs to S:

$$S(x) = \{ \alpha \mid \langle x : \alpha \rangle \in S \}.$$

For example, if

$$S = \{\langle 1 : a_1 a_3 \rangle, \langle 1 : a_1 a_2 \rangle, \langle 2 : a_2 \rangle, \langle 2 : a_2 a_2 \rangle, \langle 2 : a_2 a_3 \rangle\},$$

then

$$S(1) = \{a_1 a_3, a_1 a_2\} \quad \text{and} \quad S(2) = \{a_2, a_2 a_2, a_2 a_3\}$$

The notation $S(x)$ can be seen as the opposite of the $\langle x : X \rangle$ notation used to define states. In particular, we can write

$$S = \bigcup_x \langle x : S(x) \rangle.$$

Similarly, we can redefine $\mathit{conflict}(S)$ as

$$\mathit{conflict}(S) = \bigcup_{x \neq y} S(x) \cap S(y).$$

3.2.3 Inferring events

For several reasons, such as slow messages or malicious acceptors, a given process might not directly witness all the events that happened in the system. However, many such events can be *inferred* by the process. As an example, consider a learner with the following state

$$S = \{\langle 1 : a_1\rangle, \langle 1 : a_3a_2\rangle, \langle 2 : a_2\rangle, \langle 2 : a_2a_2\rangle, \langle 2 : a_2a_3\rangle\},$$

The event $\langle 1 : a_3a_2\rangle \in S$ means that acceptor a_2 sent a message claiming that the event $\langle 1 : a_3\rangle$ occurred at a_2. Therefore, assuming that a_2 is not malicious, we can conclude that the event $\langle 1 : a_3\rangle$ indeed occurred at a_2. From this, we can conclude that the $\langle 1 : \varepsilon\rangle$ occurred at a_3 (acceptor a_3 proposed 1), provided that neither a_2 nor a_3 are malicious. Similarly, $\langle 1 : a_1\rangle \in S$ implies that acceptor a_1 proposed 1, provided that a_1 is not malicious.

The goal of this section is to define an operator $\mathit{infer}(S, M)$ that takes a state S and a set M of malicious acceptors, and returns the set of events whose occurrence in the system can be inferred from S. The resulting set of events (a state) can contain conflicting events, so $\mathit{infer}(S, M)$ is not necessarily pure.

For example, assuming no malicious acceptors ($M = \emptyset$), we have

$$\mathit{infer}(S, \emptyset) = \{\langle 1 : \varepsilon\rangle, \langle 1 : a_1\rangle, \langle 1 : a_3\rangle, \langle 1 : a_3a_2\rangle, \langle 2 : \varepsilon\rangle, \langle 2 : a_2\rangle, \langle 2 : a_2a_2\rangle, \langle 2 : a_2a_3\rangle\}.$$

On the other hand, if a_2 is malicious ($M = \{a_2\}$), then event $\langle 2 : \varepsilon\rangle$ might not have occurred:

$$\mathit{infer}(S, \{a_2\}) = \{\langle 1 : \varepsilon\rangle, \langle 1 : a_1\rangle, \langle 1 : a_3\rangle, \langle 1 : a_3a_2\rangle, \langle 2 : a_2\rangle, \langle 2 : a_2a_2\rangle, \langle 2 : a_2a_3\rangle\}.$$

Operator *prefixes*

In order to define $\mathit{infer}(S, M)$, we will first define the operator $\mathit{prefixes}(\alpha, M)$, which takes a sequence of acceptors $\alpha = e_1e_2 \dots e_k$ and the set M of malicious acceptors, and returns a set of sequences:

$$\mathit{prefixes}(e_1e_2 \dots e_k, M) \stackrel{\text{def}}{=} \{\, e_1e_2 \dots e_i \mid e_{i+1}, e_{i+2}, \dots, e_k \notin M \,\}$$

In other words, $\mathit{prefixes}(e_1e_2 \dots e_k, M)$ produces a set of sequences that can be obtained by removing a sequence of honest acceptors from the end of $e_1e_2 \dots e_k$.

The purpose of the operator $\mathit{prefixes}(\alpha, M)$ is the following. Assume that an event $\langle x : \alpha\rangle$ with $x \neq \top$ occurred at some process. Then, we can infer that for every sequence $\beta \in \mathit{prefixes}(\alpha, M)$, the event $\langle x : \beta\rangle$ occurred in the system. For example, if event

$\langle 1 : a_1 a_2 \rangle$ occurred at some process, and $M = \emptyset$, then

$$\mathit{prefixes}(a_1 a_2, \emptyset) = \{a_1 a_2, a_1, \varepsilon\}$$

implies that events $\langle 1 : a_1 \rangle$ and $\langle 1 : \varepsilon \rangle$ occurred somewhere in the system.

To prove this, assume the event $\langle x : e_1 e_2 \ldots e_k \rangle$ occurred at some process, and that $e_1 e_2 \ldots e_i$ is a prefix of $e_1 e_2 \ldots e_k$ such that e_{i+1}, e_{i+2}, …, e_k are honest. Then, event $\langle x : e_1 e_2 \ldots e_i \rangle$ occurred at some process because the event $\langle x : e_1 e_2 \ldots e_k \rangle$ is nothing else than that acceptor e_k claims that acceptor e_{k-1} claims that … that acceptor e_{i+1} claims that the event $\langle x : e_1 e_2 \ldots e_i \rangle$ occurred at e_{i+1}. Since all acceptors e_{i+1}, e_{i+2}, …, e_k are honest, all these claims are true.

Operator *infer*

We can extend the definition of $\mathit{prefixes}(\alpha, M)$ to sets of sequences X in the obvious way:

$$\mathit{prefixes}(X, M) \stackrel{\text{def}}{=} \bigcup_{\alpha \in X} \mathit{prefixes}(\alpha, M).$$

Now, we can define $\mathit{infer}(S, M)$ using the $S(x)$ notation:

$$\mathit{infer}(S, M) \stackrel{\text{def}}{=} \hat{S} \quad \text{such that} \quad \hat{S}(x) = \mathit{prefixes}(S(x), M) \quad \text{for any } x \neq \top.$$

The case $x = \top$ requires special treatment. The state-propagation algorithm in Figure 3.1 shows that the event $\langle \top : e_1 e_2 \ldots e_k \rangle$ can occur for two reasons: either because of acceptor e_k claiming that the event $\langle \top : e_1 e_2 \ldots e_{k-1} \rangle$ occurred at e_k or because of the *stop* action. Since the latter reason is always a possibility, the occurrence of $\langle \top : e_1 e_2 \ldots e_k \rangle$ does not imply the occurrence of $\langle \top : e_1 e_2 \ldots e_{k-1} \rangle$ anywhere in the system, even if e_k is honest. For this reason, we define the operator $\mathit{infer}(S, M)$ as

$$\mathit{infer}(S, M) = \hat{S} \quad \text{such that} \quad \hat{S}(x) = \begin{cases} S(x) & \text{for } x = \top, \\ \mathit{prefixes}(S(x), M) & \text{otherwise.} \end{cases}$$

3.2.4 Special sets of sequences

In preparation for introducing the concepts of correctness and consistency of states, we will first introduce two important sets of sequences: αM and αF. Set αM consists of all sequences ending with a malicious acceptor, whereas set αF consists of all sequences ending with a faulty acceptor. For reasons discussed below, both of these sets contain the

empty sequence ε:

$$\alpha M = \{\, e_1 e_2 \dots e_k \mid e_k \in M \,\} \cup \{\varepsilon\},$$
$$\alpha F = \{\, e_1 e_2 \dots e_k \mid e_k \in F \,\} \cup \{\varepsilon\}.$$

We also define complementary sets $\alpha\overline{M}$ and $\alpha\overline{F}$, consisting of sequences that end with a non-malicious and a non-faulty acceptor, respectively:

$$\alpha\overline{M} = \{\, e_1 e_2 \dots e_k \mid e_k \notin M \,\},$$
$$\alpha\overline{F} = \{\, e_1 e_2 \dots e_k \mid e_k \notin F \,\}.$$

Intuitively, set αM contains all sequences α such that conflicting events of the form $\langle x : \alpha \rangle$ can occur in the system. For obvious reasons, this set contains all sequences ending with a malicious acceptor. Since $\langle x : \varepsilon \rangle$ corresponds to an acceptor proposing x, and different acceptors can propose different values, the empty sequence ε must belong to αM as well.

Assuming $\varepsilon \in \alpha F$ is convenient for the definition of state completeness in Section 3.4.1.

3.2.5 Correctness of states

Let M be the set of malicious acceptors. Recall that two events conflict if they have different proposal values and the same sequence of acceptors. We say that a state S is *M-correct* if there are no conflicts among events reported by honest acceptors. In other words,

$$S \text{ is } M\text{-correct} \quad \stackrel{\text{def}}{\iff} \quad \mathit{conflict}(S) \subseteq \alpha M.$$

For example, for $M = \{a_1\}$, out of the following three states S:

$$\{\langle 1 : a_3 a_1 \rangle, \langle 2 : a_3 a_1 \rangle\},$$
$$\{\langle 1 : a_3 a_2 \rangle, \langle 2 : a_3 a_2 \rangle\},$$
$$\{\langle 1 : a_3 a_2 \rangle, \langle 2 : a_2 a_2 \rangle\},$$

only the second one is *not* M-correct.

3.2.6 Consistency of states

A state S is *M-consistent* if the set of events inferred from S is M-correct. In other words,

$$S \text{ is } M\text{-consistent} \stackrel{\text{def}}{\iff} \mathit{infer}(S, M) \text{ is } M\text{-correct} \iff \mathit{conflict}(\mathit{infer}(S, M)) \subseteq \alpha M.$$

For example, for $M = \{a_1\}$, the M-correct state

$$S = \{\langle 1 : a_2a_1\rangle, \langle 2 : a_2a_2\rangle, \langle 3 : a_2a_3\rangle\}$$

is not M-consistent, because in the inferred state

$$\mathit{infer}(S, M) = \{\langle 1 : a_2a_1\rangle, \langle 2 : a_2a_2\rangle, \langle 2 : a_2\rangle, \langle 2 : \varepsilon\rangle, \langle 3 : a_2a_3\rangle, \langle 3 : a_2\rangle, \langle 3 : \varepsilon\rangle\},$$

events $\langle 2 : a_2\rangle$ and $\langle 3 : a_2\rangle$ conflict and $a_2 \notin \alpha M$. Conflicting events $\langle 2 : \varepsilon\rangle$ and $\langle 3 : \varepsilon\rangle$ are not a reason for this state not being M-consistent, because $\varepsilon \in \alpha M$.

The notion of M-consistency of states is important because the set of all events that occurred in a single run must be M-consistent. For example, assume $M = \{a_1\}$ and consider the following states

$$\begin{aligned}
S_1 &= \{\langle 1 : a_2a_1\rangle\}, & \mathit{infer}(S_1, M) &= \{\langle 1 : a_2a_1\rangle\},\\
S_2 &= \{\langle 2 : a_2a_2\rangle\}, & \mathit{infer}(S_2, M) &= \{\langle 2 : \varepsilon\rangle, \langle 2 : a_2\rangle, \langle 2 : a_2a_2\rangle\},\\
S_3 &= \{\langle 3 : a_2a_3\rangle\}, & \mathit{infer}(S_3, M) &= \{\langle 3 : \varepsilon\rangle, \langle 3 : a_2\rangle, \langle 3 : a_2a_3\rangle\}.
\end{aligned}$$

State $S_1 \cup S_2$ is M-consistent because the inferred state

$$\mathit{infer}(S_1 \cup S_2, M) = \mathit{infer}(S_1, M) \cup \mathit{infer}(S_2, M) = \{\langle 2 : \varepsilon\rangle, \langle 2 : a_2\rangle, \langle 2 : a_2a_2\rangle, \langle 1 : a_2a_1\rangle\}$$

is M-correct. State $S_1 \cup S_3$ is M-consistent for the same reason. On the other hand, state $S_2 \cup S_3$ is not M-consistent, because the state

$$\mathit{infer}(S_2, M) \cup \mathit{infer}(S_3, M) = \{\langle 2 : \varepsilon\rangle, \langle 3 : \varepsilon\rangle, \langle 2 : a_2\rangle, \langle 3 : a_2\rangle, \langle 2 : a_2a_2\rangle, \langle 3 : a_2a_3\rangle\}$$

contains conflicting events $\langle 2 : a_2\rangle$ and $\langle 3 : a_2\rangle$, and $a_2 \notin \alpha M$. As a result, states S_2 and S_3 cannot occur in the same run.

3.3 Predicates

Section 2.2 specified OTC in terms of actions $\mathit{propose}(x)$ and *stop* available to acceptors, and predicates $\mathit{decision}(x)$, $\mathit{possible}(x)$, and $\mathit{valid}(x)$ available to learners. These primitives satisfy:

Integrity. If $\mathit{valid}(x)$, then an honest acceptor proposed x.

Possibility. If $\mathit{decision}(x)$, then $\mathit{possible}(x)$ holds at all learners, at all times.

Permanent Validity. For any complete learner, $\mathit{possible}(x) \implies \mathit{valid}(x)$ for all x.

Permanent Agreement. For any complete learner, *possible*(x) holds for at most one x.

Optimistic Termination (q, k)**.** If at most q out of n acceptors are faulty, all correct acceptors propose x, and none of them executes *stop*, then *decision*(x) will hold at all correct learners after k communication steps.

Predicates *decision*(x) and *valid*(x) are stable, that is, once they are true, they remain true forever. Predicate *possible*(x) is anti-stable: once false, it remains false forever.

Treating predicates as Boolean functions with TRUE $>$ FALSE enables us to reformulate some definitions in terms of arithmetic instead of logic. For example, a predicate is (anti-) stable if it is an increasing (decreasing) function of time. Here, a function $f(t)$ is increasing if $t < t' \implies f(t) \leq f(t')$, and decreasing if $t < t' \implies f(t) \geq f(t')$. We say that a predicate A is stronger (weaker) than B if $A \geq B$ ($A \leq B$). Note that for Boolean values $p \leq q$ is equivalent to $p \implies q$. In this case, p and q can be entire logic expressions; for example, Integrity is a decreasing function of *valid*(x).

3.3.1 Overview of correctness testing

Section 3.1 gave a description of the execution of an OTC algorithm (Figure 3.1), including actions *propose*(x) and *stop* made by acceptors. In Section 3.2, we formalized the notion of a state and introduced some state-related operations, such as *infer*(S, M). We are now ready to give precise definitions of the predicates provided by any OTC algorithm: *valid*, *decision*, and *possible*. In Section 3.4, we will use them to present an automatic method of checking the correctness of OTC algorithms.

In brief, our method of testing the correctness of OTC algorithms works as follows. We specify an OTC algorithm by listing the Optimistic Termination properties that we require. Section 3.3.4, determines the weakest possible predicate *decision*(x) that satisfies these properties. Section 3.3.5 uses this *decision*(x) and the Possibility property of OTC to determine the weakest predicate *possible*(x). Section 3.3.6 uses the Integrity property to determine the strongest possible predicate *valid*(x).

Section 3.4, we will test Permanent Validity and Permanent Agreement using the strongest *valid*(x) and weakest *possible*(x) determined in the previous paragraph. Both properties are increasing functions of *valid*(x) and decreasing functions of *possible*(x), so if they do not hold for the strongest *valid*(x) and the weakest *possible*(x), they cannot hold for any predicates *valid*(x) and *possible*(x) satisfying the other properties. On the other hand, if Permanent Validity and Permanent Agreement do hold, then we have found an algorithm satisfying all properties required by OTC, including our Optimistic Termination conditions.

3.3.2 Extended failure model

Let $A = \{a_1, \ldots, a_n\}$ be the set of all acceptors. In the previous chapters, we assumed that at most f of them can be faulty, out of which at most m malicious. In other words, we assumed that the set $F \subseteq A$ of faulty acceptors contains at most f elements, and that the set $M \subseteq A$ of malicious acceptors contains at most m elements.

The drawback of this method of restricting the possible sets of faulty acceptors is that it implicitly assumes that all acceptors are the same and they fail independently. Consider an example with three acceptors a_1, a_2, and a_3. We can require that at most two acceptors are faulty by setting $f = 2$. However, it is impossible to express the assumption that a_1 and a_2 cannot fail at the same time. To be able to express such requirements, we will generalize our method of specifying sets F and M. In this chapter only, we will use two families of sets of acceptors: $\mathcal{F}$ and $\mathcal{M}$. Family $\mathcal{F}$ contains all possible sets F of faulty acceptors, whereas $\mathcal{M}$ contains all possible sets M of malicious ones. In other words, we require

$$F \in \mathcal{F}, \qquad M \in \mathcal{M}, \qquad M \subseteq F.$$

In our three-acceptor example, these sets can be defined as follows

$$\begin{aligned} \mathcal{F} &= \{\emptyset, \{a_1\}, \{a_2\}, \{a_3\}, \{a_1, a_3\}, \{a_2, a_3\}\}, \\ \mathcal{M} &= \{\emptyset, \{a_1\}, \{a_2\}, \{a_3\}\}. \end{aligned}$$

This allows at most two acceptors to be faulty, out of which at most one malicious, and forbids a_1 and a_2 to be both faulty. For example, we can have

$$F = \{a_1, a_3\}, \qquad M = \{a_3\}$$

but not

$$F = \{a_1, a_3\}, \qquad M = \{a_2\}.$$

because although $F \in \mathcal{F}$ and $M \in \mathcal{M}$, the requirement $M \subseteq F$ does not hold.

This method of specifying allowed sets F and M is more general than the one that just limits their sizes by f and m, respectively. In fact, any restrictions given by the latter model can be transformed into the one used in this chapter by setting

$$\begin{aligned} \mathcal{F} &= \{\, F \subseteq A \mid |F| \leq f \,\}, \\ \mathcal{M} &= \{\, M \subseteq A \mid |M| \leq m \,\}. \end{aligned}$$

3.3.3 Termination rules

The extended failure model from Section 3.3.2 prompts for introducing a more general variant of the Optimistic Termination condition. Each such condition is parameterized by $\langle V, C, k\rangle$, where $V \subseteq C \subseteq A$ are sets of acceptors and k is a positive integer.

> **Optimistic Termination $\langle V, C, k\rangle$.** If all acceptors in V propose the same value x, all acceptors in C are correct, and no correct acceptor executes *stop*, then *decision*(x) will hold at all correct learners in k communication steps.

We call tuples $\langle V, C, k\rangle$ *termination rules*.

Take a system with three honest acceptors a_1, a_2, a_3 as an example. Consider an OTC algorithm with the following Termination condition. First, if all acceptors are correct and propose the same value, then all correct learners decide in one step. Second, if a_1 and one other acceptor are correct, then the algorithm decides in two steps on the value proposed by a_1. This condition can be expressed by the following set of three termination rules:

$$\mathcal{T} = \left\{ \begin{array}{llll} \langle & V = \{a_1, a_2, a_3\}, & C = \{a_1, a_2, a_3\}, & k = 1 \ \rangle \\ \langle & V = \{a_1\}, & C = \{a_1, a_2\}, & k = 2 \ \rangle \\ \langle & V = \{a_1\}, & C = \{a_1, a_3\}, & k = 2 \ \rangle \end{array} \right\}.$$

Observe that the standard Optimistic Termination (q, k) corresponds to

$$\mathcal{T} = \{\, \langle X, X, k\rangle \mid X \subseteq A, |X| = q \,\}.$$

3.3.4 Predicate $decision_S(x)$

From now on, we will use a subscript S in all state predicates to indicate which state S they operate on. This section will show how to determine the weakest predicate $decision_S(x)$ that satisfies Optimistic Termination T, where $T = \langle V, C, k\rangle$. Assume that all acceptors in V proposed the same value x, all acceptors in C are correct, and none of them executed *stop*. After k communication steps, the set of events that must have occurred at every correct learner is $\langle x : rule(T)\rangle$, where

$$rule(T) = \{\, e_1 e_2 \ldots e_j \mid e_1 \in V \text{ and } e_i \in C \text{ for all } i \leq j \leq k \,\}.$$

For example,

$$rule(\langle\{a_1, a_2\}, \{a_1, a_2, a_3\}, 2\rangle) = \{a_1, a_2, a_1a_1, a_1a_2, a_1a_3, a_2a_1, a_2a_2, a_2a_3\}.$$

For any termination rule $T = \langle V, C, k \rangle$, the set $D = \mathit{rule}(T)$ is the corresponding *decision rule*. If all acceptors in V proposed the same value x, all acceptors in C are correct, and none of them executed *stop*, then all events in $\langle x : D \rangle$ must occur at every correct learner within k communication steps. For that reason, a learner cannot violate Optimistic Termination T by delaying the decision until all events from $\langle x : D \rangle$ have occurred. On the other hand, if acceptors outside V do not propose anything, and acceptors outside C crash at the beginning, the $\langle x : D \rangle$ are the only events that will occur at the learners within k communication steps. As a consequence, a learner must decide on x as soon as all events from $\langle x : D \rangle$ occurred. In other words, the weakest predicate $\mathit{decision}_S(x)$ can be defined as

$$\mathit{decision}_S(x) \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad S(x) \supseteq D.$$

The above definition assumes a single decision rule $D = \mathit{rule}(T)$. For sets $\mathcal{T}$ consisting of many termination rules T, we must consider a family $\mathcal{D}$ of decision rules, defined as

$$\mathcal{D} = \mathit{rule}(\mathcal{T}) = \{\, \mathit{rule}(T) \mid T \in \mathcal{T} \,\}.$$

For example, $\mathcal{T}$ from Section 3.3.3 leads to:

$$\mathit{rule}(\mathcal{T}) = \left\{ \begin{array}{c} \{a_1, a_2, a_3\} \\ \{a_1, a_1a_1, a_1a_2\} \\ \{a_1, a_1a_1, a_1a_3\} \end{array} \right\}.$$

For multiple decision rules, the definition of $\mathit{decision}_S(x)$ generalizes to:

$$\mathit{decision}_S(x) \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad S(x) \supseteq D \text{ for some } D \in \mathcal{D}.$$

This section showed that predicate $\mathit{decision}_S(x)$ is uniquely determined by the set $\mathcal{T}$ of termination rules. Section 3.3.5 will show that predicate $\mathit{possible}_S(x)$ is uniquely determined by $\mathit{decision}_S(x)$. Actions $\mathit{propose}(x)$ and *stop* (Section 3.1) and predicate $\mathit{valid}_S(x)$ (Section 3.3.6) are defined in an algorithm-independent way. As a result, an OTC algorithm is uniquely determined by $\mathcal{T}$. For this reason, we will sometimes refer to it as "algorithm $\mathcal{T}$".

3.3.5 Predicate $\mathit{possible}_S(x)$

This section will show how to determine the weakest $\mathit{possible}_S(x)$ allowed by the OTC property Possibility. This property requires that if $\mathit{decision}(x)$ holds at some learner, then $\mathit{possible}(x)$ holds at all learners at all times. In other words, if $\mathit{decision}_{\hat{S}}(x)$ holds for some state $\hat{S}$, then $\mathit{possible}_S(x)$ must hold for all states S that can occur in the same run as $\hat{S}$. By inverting this implication, we conclude that $\mathit{possible}_S(x)$ must be true if

there is a state $\hat{S}$ with the following properties:

1. State $\hat{S}$ decides on x, that is, $decision_{\hat{S}}(x)$ holds:

$$\hat{S}(x) \supseteq D \quad \text{for some } D \in \mathcal{D}.$$

2. States S and $\hat{S}$ can occur in the same execution, that is, $S \cup \hat{S}$ is M-consistent for some $M \in \mathcal{M}$:

$$conflict(infer(S \cup \hat{S}, M)) \subseteq \alpha M \quad \text{for some } M \in \mathcal{M},$$

The weakest predicate $possible_S(x)$ that satisfies the above conditions holds exactly when both of the above conditions hold. To simplify the definition, note that the second condition is a decreasing function of $\hat{S}$. Therefore, we can restrict ourselves to checking only minimal states $\hat{S}$ allowed by the first property, that is, sets $\hat{S} = \langle x : D \rangle$ with $D \in \mathcal{D}$. We can now define $possible_S(x)$ as

$$possible_S(x) \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad conflict(infer(S \cup \langle x : D \rangle, M)) \subseteq \alpha M \quad \text{for some } D \in \mathcal{D} \text{ and } M \in \mathcal{M}.$$

Example 1

Consider an OTC algorithm in which a learner decides in one step if all three acceptors are correct and propose the same value:

$$\mathcal{D} = \{\{a_1, a_2, a_3\}\}.$$

Assume that at most one acceptor is maliciously faulty and consider the state

$$S = \{\langle 1 : a_1 \rangle, \langle \top : a_1 a_2 \rangle, \langle \top : a_3 \rangle\}.$$

Predicate $possible_S(1)$ holds in this state because it is possible that all three acceptors proposed 1, and sent their states to some learner, which decided on 1. The messages from a_2 may be very slow, a_2 might have executed *stop* before receiving a_1's proposal, and a_3 could be maliciously reporting $\top$ despite proposing 1.

Formally, $possible_S(1)$ holds because state S and

$$\hat{S} = \{\langle 1 : a_1 \rangle, \langle 1 : a_2 \rangle, \langle 1 : a_3 \rangle\},$$

for which $decision_{\hat{S}}(1)$ holds, can occur in the same execution. This is because for $M = \{a_3\}$ we have

$$infer(S \cup \hat{S}, M) = \{\langle 1 : \varepsilon \rangle, \langle 1 : a_1 \rangle, \langle 1 : a_2 \rangle, \langle 1 : a_3 \rangle, \langle \top : a_1 a_2 \rangle, \langle \top : a_3 \rangle\}.$$

Hence,

$$\mathit{conflict}(\mathit{infer}(S \cup \hat{S}, M)) = \{a_3\} \subseteq \alpha M.$$

Example 2

Consider

$$S = \{\langle 1 : a_1\rangle, \langle 2 : a_1 a_2\rangle, \langle \top : a_3\rangle\}.$$

Here, a_1 or a_2 must be malicious, so acceptor a_3 is honest. As before, a_3 reports executing stop before proposing anything, but this time we can trust its reports and conclude that a_3 has not proposed anything. Therefore, $\mathit{possible}_S(1)$ does not hold.

For example, for $M = \{a_3\}$ we have

$$\mathit{infer}(S \cup \hat{S}, M) = \{\langle 1 : \varepsilon\rangle, \langle 1 : a_1\rangle, \langle 1 : a_2\rangle, \langle 1 : a_3\rangle, \langle 2 : \varepsilon\rangle, \langle 2 : a_1\rangle, \langle 2 : a_1 a_2\rangle, \langle \top : a_3\rangle\}.$$

Here, conflicting events $\langle x : \alpha\rangle$ have $\alpha \in \{\varepsilon, a_1, a_3\}$. Since $a_1 \notin \alpha M$, state $S \cup \hat{S}$ is not M-consistent for $M = \{a_3\}$. Similarly we can show that $S \cup \hat{S}$ is not M-consistent for any other set M containing at most one acceptor. As a result, $\mathit{possible}_S(1)$ does not hold.

3.3.6 Predicate $\mathit{valid}_S(x)$

In this section, we will determine the strongest predicate $\mathit{valid}_S(x)$ allowed by Integrity. It requires that if $\mathit{valid}_S(x)$ holds at a learner in state S, then an honest acceptor proposed x. In other words, if $\mathit{valid}_S(x)$ holds, then it can be inferred from S that the event $\langle x : \varepsilon\rangle$ occurred at some honest acceptor. This statement must be true for every possible set $M \in \mathcal{M}$ of malicious acceptors for which state S can be attained. Since learners do not propose values themselves and events that happened at malicious acceptors cannot be inferred, we can define $\mathit{valid}_S(x)$ as

$$\mathit{valid}_S(x) \quad \stackrel{\text{def}}{\iff} \quad S \text{ is } M\text{-consistent} \implies \langle x : \varepsilon\rangle \in \mathit{infer}(S, M) \quad \text{for all } M \in \mathcal{M}.$$

We can assume that $x \neq \top$ because $\top$ is disjoint from the set of possible proposals x (Section 3.1.5). Thus, we can use the definition of $\mathit{infer}(S, M)$ to give an equivalent definition of $\mathit{valid}_S(x)$:

$$\mathit{valid}_S(x) \quad \stackrel{\text{def}}{\iff} \quad S \text{ is } M\text{-consistent} \implies \varepsilon \in \mathit{prefixes}(S(x), M) \quad \text{for all } M \in \mathcal{M}.$$

Note that this definition of $\mathit{valid}_S(x)$ is common to all OTC algorithms.

Example

Consider a system of three acceptors a_1, a_2, a_3 with at most two of them malicious. Consider the state

$$S = \{\langle 1 : a_1\rangle, \langle 2 : a_1a_1\rangle, \langle 3 : a_2\rangle, \langle 3 : a_3\rangle\}.$$

It is obvious that a_1 is malicious because it reports (indirectly) to have proposed to different values. Therefore, at most one of a_2 and a_3 is malicious. Since they both report to have proposed 3, at least one of them is honest and proposed 3, which implies $valid_S(3)$. Formally,

$$a_1 \notin M \implies \{\langle 1 : a_1\rangle, \langle 2 : a_1\rangle\} \subseteq \mathit{infer}(S, M) \implies \text{S in not } M\text{-consistent},$$
$$a_1 \in M \implies S(3) = \{a_2, a_3\} \not\subseteq M \implies \varepsilon \in \mathit{prefixes}(S(3), M).$$

Therefore, "S is M-consistent $\implies$ $\varepsilon \in \mathit{prefixes}(S(3), M)$" holds for all $M \in \mathcal{M}$, which implies $valid_S(3)$.

The situation is different if $\langle 2 : a_1a_1\rangle \notin S$. For $M = \{a_2, a_3\} \in \mathcal{M}$ we have

$$\mathit{infer}(S, M) = S = \{\langle 1 : \varepsilon\rangle, \langle 1 : a_1\rangle, \langle 3 : a_2\rangle, \langle 3 : a_3\rangle\}.$$

State S is M-consistent and $\varepsilon \notin \mathit{prefixes}(S(3), M) = \{a_2, a_3\}$, so $valid_S(3)$ does not hold.

3.4 Testing correctness of OTC algorithms

In the previous sections, we showed that an OTC algorithm can be uniquely determined by the following parameters: the set A of acceptors, the family $\mathcal{F}$ of possible sets of faulty acceptors, the family $\mathcal{M}$ of possible sets of malicious acceptors, and the set $\mathcal{T}$ of termination rules. We proved this by constructing the strongest $valid_S(x)$ and the weakest $decision_S(x)$ and $possible_S(x)$ allowed by the OTC properties Integrity, Optimistic Termination, and Possibility, respectively:

$$valid_S(x) \stackrel{\text{def}}{\iff} S \text{ is } M\text{-consistent} \implies \varepsilon \in \mathit{prefixes}(S(x), M), \text{ for all } M \in \mathcal{M},$$
$$decision_S(x) \stackrel{\text{def}}{\iff} S \supseteq D \text{ for some } D \in \mathcal{D} = \mathit{rule}(\mathcal{T}),$$
$$possible_S(x) \stackrel{\text{def}}{\iff} \mathit{conflict}(\mathit{infer}(S \cup \langle x : D\rangle, M)) \subseteq \alpha M, \text{ for some } D \in \mathcal{D},\ M \in \mathcal{M}.$$

In this section, we will use these definitions to test the other two OTC properties: Permanent Validity and Permanent Agreement.

3.4.1 Completeness of states

The Permanent Validity and Permanent Agreement properties both assume that the learner's state is complete. This means (Section 2.2) that all *correct* acceptors have executed *stop* and the learner has received all messages sent by these acceptors before or during their (first) *stop* action.

Recall from Section 3.1.5 that executing *stop* leaves a correct acceptor e_k in a state S with some $\langle x : e_1 e_2 \dots e_{k-1} \rangle \in S$ for every $e_1 e_2 \dots e_{k-1}$. As a result, before finishing executing *stop*, this acceptor has broadcast some event $\langle x : e_1 e_2 \dots e_{k-1} \rangle$ for every $e_1 e_2 \dots e_{k-1}$. Therefore, at any complete learner, at least one event $\langle x : e_1 e_2 \dots e_k \rangle$ for each $e_1 e_2 \dots e_k$ with $e_k \notin F$ must have occurred. For this reason, we define

$$S \text{ is } F\text{-complete} \quad \stackrel{\text{def}}{\iff} \quad \alpha\overline{F} \subseteq \bigcup_x S(x).$$

Example

Consider a system consisting of two acceptors a_1 and a_2, with a_1 being faulty ($F = \{a_1\}$). For simplicity, let us restrict our attention to sequences $e_1 e_2 \dots e_k$ with $k \leq 2$, in which case $\alpha\overline{F} = \{a_2, a_1 a_2, a_2 a_2\}$. The state

$$S = \{\langle 2 : a_1 \rangle, \langle 1 : a_2 \rangle, \langle 1 : a_2 a_2 \rangle\}$$

is not F-complete, because $a_1 a_2 \in \alpha\overline{F}$ and S does not contain any event of the form $\langle x : a_1 a_2 \rangle$. On the other hand, state

$$S = \{\langle 2 : a_1 \rangle, \langle \top : a_1 a_2 \rangle, \langle 1 : a_2 \rangle, \langle 1 : a_2 a_2 \rangle\}$$

is F-complete.

3.4.2 Permanent Validity

Permanent Validity requires that $possible_S(x) \implies valid_S(x)$ for any complete state S. This section presents an algorithm that checks whether a given OTC algorithm satisfies this property, by trying to find a complete state S that violates it. In other words, we will be looking for a run in which some learners arrive at a complete state S for which there exists an x such that $possible_S(x)$ holds but $valid_S(x)$ does not. If such a state can be found, its existence proves that the OTC algorithm does not satisfy Permanent Validity. If such a state does not exist, then the Permanent Validity property holds.

Formally, we are looking for sets $F \in \mathcal{F}$ and $M \in \mathcal{M}$, with $M \subseteq F$, and a state S such that

V1: State S can occur (is M-consistent):

$$\mathit{conflict}(\mathit{infer}(S, M)) \subseteq \alpha M.$$

V2: State S is F-complete:

$$\alpha\overline{F} \subseteq \bigcup_x S(x).$$

V3: Predicate $\mathit{possible}_S(x)$ holds:

$$\mathit{conflict}(\mathit{infer}(S \cup \langle x : D_x\rangle, M_x)) \subseteq \alpha M_x \quad \text{for some } M_x \in \mathcal{M} \text{ and } D_x \in \mathcal{D}.$$

V4: Predicate $\mathit{valid}_S(x)$ does not hold:

$$\mathit{conflict}(\mathit{infer}(S, M_v)) \subseteq \alpha M_v \text{ and } \varepsilon \notin \mathit{prefixes}(S(x), M_v) \quad \text{for some } M_v \in \mathcal{M}.$$

We will be iterating over all possible values of (F, M, M_x, D_x, M_v). For each possible (F, M, M_x, D_x, M_v), we will try to find a state S that satisfies Properties **V**. If we succeed for at least one (F, M, M_x, D_x, M_v), then Permanent Validity is not met. Otherwise, Permanent Validity holds.

For the rest of the section, assume that the values of F, M, M_x, D_x, M_v are fixed. How do we find a state S that satisfies the above properties? Trying all possible states S is prohibitive; even assuming that all events in S are of the form $\langle x : e_1 e_2 \dots e_i\rangle$ with the same x and $i \leq k$ for some k, the number of such states is $2^{1+n+\dots+n^k}$. We will need a better method of finding S.

Without loss of generality we can assume that the state S consists only of events of the form $\langle x : \alpha\rangle$ or $\langle \top : \alpha\rangle$. This is because all events $\langle y : \alpha\rangle \in S$ with $y \notin \{x, \top\}$ can be replaced by $\langle \top : \alpha\rangle$ without invalidating any of the above four properties. Therefore, we can assume that $S = \langle x : S_x\rangle \cup \langle \top : S_\top\rangle$, for some disjoint sets of sequences S_x and $S_\top$.

Given this assumption, Properties **V** can be rewritten as

V1: State S is M-consistent:

$$\mathit{prefixes}(S_x, M) \cap S_\top \subseteq \alpha M.$$

V2: State S is F-complete:

$$\alpha\overline{F} \subseteq S_x \cup S_\top.$$

```
1  function PermanentValidity(A, F, M, T)
2    for all D_x ∈ rule(T) do
3      for all F ∈ F do
4        for all M, M_x, M_v ∈ M do
5          if M ⊆ F then
6            iteratively compute the least fixed point S_x of (3.1)
7            if computed S_x satisfies (3.2) then
8              return FALSE
9    return TRUE
```

Figure 3.2: The algorithm for testing Permanent Validity.

V3: Predicate $possible_S(x)$ holds:

$$prefixes(S_x, M_x) \cap S_\top \subseteq \alpha M_x$$
$$prefixes(D_x, M_x) \cap S_\top \subseteq \alpha M_x$$

V4: Predicate $valid_S(x)$ does not hold:

$$prefixes(S_x, M_v) \cap S_\top \subseteq \alpha M_v, \tag{a}$$
$$\varepsilon \notin prefixes(S_x, M_v). \tag{b}$$

Property **V2** is increasing with respect to $S_\top$; all the other properties are decreasing. For this reason, we can assume that $S_\top = \alpha\overline{F} \setminus S_x$. This is the smallest $S_\top$ allowed by Property **V2**, making it automatically satisfied.

To eliminate $S_\top$ from the other properties notice that for any set X

$$X \cap S_\top \subseteq \alpha M \quad \iff \quad X \cap \alpha\overline{F} \cap \overline{S_x} \cap \alpha\overline{M} = \emptyset \quad \iff \quad X \cap \alpha\overline{F} \cap \alpha\overline{M} \subseteq S_x.$$

Properties **V1**, **V3**, and **V4**(a) can thus be rewritten as

$$\begin{aligned} &S_x \supseteq prefixes(S_x, M_*) \cap \alpha\overline{F} \cap \alpha\overline{M_*} \quad \text{for all } M_* \in \{M, M_x, M_v\}, \\ &S_x \supseteq prefixes(D_x, M_x) \cap \alpha\overline{F} \cap \alpha\overline{M_x}. \end{aligned} \tag{3.1}$$

Function *prefixes* is increasing with respect to its first argument, so the right-hand side of each of these inequalities is an increasing function of S_x. As shown below, this allows us to iteratively compute the smallest set S_x that satisfies (3.1). Having computed S_x, it is then sufficient to check Property **V4**(b):

$$\varepsilon \notin prefixes(S_x, M_v). \tag{3.2}$$

If it holds, then we have found a state $S = \langle x : S_x \rangle \cup \langle \top : S_\top \rangle$ for which Permanent Validity does not hold. If not, then the above statement will be false for all supersets of S_x because *prefixes* is increasing with respect to its first argument. As a result, for a given (F, M, M_x, D_x, M_v), there is no state S violating Permanent Validity. The complete Permanent Validity testing algorithm is shown in Figure 3.2.

Computing S_x as the least fixed point

Inequalities (3.1) can be rewritten as $S_x \supseteq \phi(S_x)$, where

$$\phi(X) \quad = \quad \mathit{prefixes}(D_x, M_x) \cap \alpha\overline{F} \cap \alpha\overline{M_x} \quad \cup \bigcup_{M_* \in \{M, M_x, M_v\}} \mathit{prefixes}(X, M_*) \cap \alpha\overline{F} \cap \alpha\overline{M_*}$$

is an increasing function of X. This allows us to use the iterative fixpoint algorithm by Tarski [115] to find the smallest X satisfying $X \supseteq \phi(X)$, that is, the smallest S_x satisfying inequalities (3.1).

Tarski's method constructs an increasing sequence $X_0 \subseteq X_1 \subseteq \cdots$ defined as $X_0 = \emptyset$ and $X_{i+1} = \phi(X_i)$. The first $X_i = X_{i+1} = \phi(X_i)$ encountered is the least fixed point of ϕ. In the sequence $X_0 \subset \cdots \subset X_i$, each set has at least one element more than its predecessor, so the number i of iterations does not exceed the maximum size of X_i. In our case, the number of iterations does not exceed the number of sequences $e_1 e_2 \ldots e_i$. Assuming $i \leq k$, this number is $1 + \cdots + n^k$, not $2^{1+\cdots+n^k}$ as in the direct search for state S.

Example 1

Consider a four-acceptor system, with at most two faulty acceptors, one of which is malicious. Consider an OTC algorithm that decides in one communication step if all correct acceptors propose the same value:

$$\mathcal{D} = \left\{ \begin{matrix} \{a_1, a_2\}, \{a_1, a_3\}, \{a_1, a_4\} \\ \{a_2, a_3\}, \{a_2, a_4\}, \{a_3, a_4\} \end{matrix} \right\}.$$

We will use the algorithm in Figure 3.2 to find a state S that violates Permanent Validity. Consider the following parameters:

$$M = \emptyset, \quad F = \{a_1, a_2\}, \quad M_x = \{a_4\}, \quad M_v = \{a_3\}, \quad D_x = \{a_3, a_4\} \in \mathcal{D}.$$

Since we are only interested in one-step decision, we can limit our attention to events $\langle x : \alpha \rangle$ with the sequence α containing at most one acceptor, which results in $\alpha\overline{F} = \{a_3, a_4\}$.

To find the state $S = \langle x : S_x \rangle \cup \langle \top : S_\top \rangle$ that violates Permanent Validity, we will first find S_x using inequalities (3.1):

$$S_x \supseteq \mathit{prefixes}(D_x, M_x) \cap \alpha\overline{F} \cap \alpha\overline{M_x} = \{\varepsilon, a_3, a_4\} \cap \alpha\overline{F} \cap \alpha\overline{M_x} = \{a_3\},$$
$$S_x \supseteq \mathit{prefixes}(S_x, M) \cap \alpha\overline{F} \cap \alpha\overline{M} = \{\varepsilon, a_3\} \cap \alpha\overline{F} \cap \alpha\overline{M} = \{a_3\},$$
$$S_x \supseteq \mathit{prefixes}(S_x, M_x) \cap \alpha\overline{F} \cap \alpha\overline{M_x} = \{\varepsilon, a_3\} \cap \alpha\overline{F} \cap \alpha\overline{M_x} = \{a_3\},$$
$$S_x \supseteq \mathit{prefixes}(S_x, M_v) \cap \alpha\overline{F} \cap \alpha\overline{M_v} = \{a_3\} \cap \alpha\overline{F} \cap \alpha\overline{M_v} = \emptyset.$$

This means that $\{a_3\}$ is the smallest S_x set satisfying the above inequalities. This leads to $S_\top = \alpha\overline{F} \setminus S_x = \{a_4\}$, which results in $S = \{\langle x : a_3 \rangle, \langle \top : a_4 \rangle\}$. Formula (3.2)

$$\varepsilon \notin \mathit{prefixes}(S_x, M_v) = \{a_3\},$$

proves that S does not satisfy Permanent Validity.

Let us now present the above example in natural language. Consider a run with faulty acceptors a_1 and a_2, in which only a_3 proposes anything, say x. Assume that all correct acceptors have executed *stop*, and consider a learner in a complete state $S = \{\langle x : a_3 \rangle, \langle \top : a_4 \rangle\}$. The learner does not know which acceptors are faulty. It is possible that acceptors a_1 and a_2 are correct but slow and their messages have not reached the learner yet. In this case, if a_3 lies about proposing x, then no honest acceptor proposed x, so $\mathit{valid}_S(x)$ must be false. On the other hand, if a_4 lies about not proposing x, another learner might be in state $\hat{S} = \{\langle x : a_3 \rangle, \langle x : a_4 \rangle\}$. Since $\hat{S}(x) = \{a_3, a_4\} \in \mathcal{D}$, predicate $\mathit{decision}_{\hat{S}}(x)$ holds, so predicate $\mathit{possible}_S(x)$ must hold at the original learner. To sum up, state S is complete, $\mathit{possible}_x(S)$ holds but $\mathit{valid}_x(S)$ does not, which contradicts Permanent Validity.

Example 2

Consider a four-acceptor system, with at most one faulty acceptor, possibly malicious. Consider an OTC algorithm that decides in one communication step if all correct acceptors propose the same value:

$$\mathcal{D} = \{\{a_1, a_2, a_3\}, \{a_1, a_2, a_4\}, \{a_1, a_3, a_4\}, \{a_2, a_3, a_4\}\}.$$

Consider

$$M = \{a_1\}, \quad F = \{a_1\}, \quad M_x = \{a_4\}, \quad M_v = \{a_3\}, \quad D_x = \{a_2, a_3, a_4\} \in \mathcal{D}.$$

Other cases of (M, F, M_x, M_v, D_x) can be checked in a similar way, but this case is probably most interesting. As in Example 1, we do not consider sequences containing

more than one acceptor.

First, we use inequalities (3.1) to compute S_x:

$$S_x \supseteq \mathit{prefixes}(D_x, M_x) \cap \alpha\overline{F} \cap \alpha\overline{M_x} = \{\varepsilon, a_2, a_3, a_4\} \cap \alpha\overline{F} \cap \alpha\overline{M_x} = \{a_2, a_3\},$$
$$S_x \supseteq \mathit{prefixes}(S_x, M) \cap \alpha\overline{F} \cap \alpha\overline{M} = \{\varepsilon, a_2, a_3\} \cap \alpha\overline{F} \cap \alpha\overline{M} = \{a_2, a_3\},$$
$$S_x \supseteq \mathit{prefixes}(S_x, M_x) \cap \alpha\overline{F} \cap \alpha\overline{M_x} = \{\varepsilon, a_2, a_3\} \cap \alpha\overline{F} \cap \alpha\overline{M_x} = \{a_2, a_3\},$$
$$S_x \supseteq \mathit{prefixes}(S_x, M_v) \cap \alpha\overline{F} \cap \alpha\overline{M_v} = \{\varepsilon, a_2, a_3\} \cap \alpha\overline{F} \cap \alpha\overline{M_v} = \{a_2, a_3\},$$

which gives us $S_x = \{a_2, a_3\}$ and $S = \{\langle x : a_2\rangle, \langle x : a_3\rangle, \langle \top : a_4\rangle\}$. Now, we use formula (3.2) to see that state S does not violate Permanent Validity:

$$\varepsilon \in \mathit{prefixes}(S_x, M) = \{\varepsilon, a_2, a_3\}.$$

3.4.3 Permanent Agreement

Permanent Agreement requires that for any complete state S, predicate $\mathit{possible}_S(x)$ holds for at most one x. This section presents an algorithm, similar to that from Section 3.4.2, that checks whether a given OTC algorithm satisfies this property. We will be looking for a run in which some learners can arrive at a complete state S for which $\mathit{possible}_S(z)$ holds for two different $z \in \{x, y\}$. If such a state can be found, its existence will prove that the algorithm does not satisfy Permanent Agreement. If such a state does not exist, then the Permanent Agreement property is met.

Formally, we are looking for sets $F \in \mathcal{F}$ and $M \in \mathcal{M}$, with $M \subseteq F$, and a state S such that

A1: State S can occur (is M-consistent):

$$\mathit{conflict}(\mathit{infer}(S, M)) \subseteq \alpha M.$$

A2: State S is F-complete:

$$\alpha\overline{F} \subseteq \bigcup_x S(x).$$

A3: Predicate $\mathit{possible}_S(z)$ holds for $z \in \{x, y\}$, where $x \neq y$:

$$\mathit{conflict}(\mathit{infer}(S \cup \langle z : D_z\rangle, M_z)) \subseteq \alpha M_z \quad \text{for some } M_z \in \mathcal{M} \text{ and } D_z \in \mathcal{D}.$$

Similarly to Permanent Validity, we will be iterating over all possible values of F, M, M_x, D_x, M_y, D_y. For each $(F, M, M_x, D_x, M_y, D_y)$, we will try to find a state S that

satisfies all the above properties. If we succeed for at least one $(F, M, M_x, D_x, M_y, D_y)$, then Permanent Agreement is not met. Otherwise, Permanent Agreement holds.

Without loss of generality, we can assume that the state S consists only of events of the form $\langle x : \alpha \rangle$, $\langle y : \alpha \rangle$, and $\langle \top : \alpha \rangle$. This is because all events $\langle z : \alpha \rangle \in S$ with $z \notin \{x, y, \top\}$ can be replaced by $\langle \top : \alpha \rangle$ without invalidating any of the above three properties. For this reason, we can assume that $S = \langle x : S_x \rangle \cup \langle y : S_y \rangle \cup \langle \top : S_\top \rangle$, for some pairwise disjoint sets of sequences S_x, S_y, and $S_\top$.

Given this assumption the Properties **A** can be rewritten as

A1: State S is M-consistent:

$$\begin{aligned} \mathit{prefixes}(S_x, M) \cap \mathit{prefixes}(S_y, M) &\subseteq \alpha M \qquad \text{(a)} \\ \mathit{prefixes}(S_x, M) \cap S_\top &\subseteq \alpha M \\ \mathit{prefixes}(S_y, M) \cap S_\top &\subseteq \alpha M \end{aligned}$$

Since $X \subseteq \mathit{prefixes}(X, M)$ for any set of sequences X, the first inequality implies $\mathit{prefixes}(S_x, M) \cap S_y \subseteq \alpha M$ and $\mathit{prefixes}(S_y, M) \cap S_x \subseteq \alpha M$. Therefore, we can rewrite the last two inequalities as:

$$\mathit{prefixes}(S_x, M) \cap (S_y \cup S_\top) \subseteq \alpha M, \qquad \text{(b)}$$
$$\mathit{prefixes}(S_y, M) \cap (S_x \cup S_\top) \subseteq \alpha M. \qquad \text{(c)}$$

The reason for this transformation will become clear later.

A2: State S is F-complete:

$$\alpha \overline{F} \subseteq S_x \cup S_y \cup S_\top.$$

A3: Predicate $\mathit{possible}_S(z)$ holds for $z \in \{x, y\}$. Defining $\bar{x} \stackrel{\text{def}}{=} y$ and $\bar{y} \stackrel{\text{def}}{=} x$, and using the same transformations as in Property **A1**, we get:

$$\begin{aligned} \mathit{prefixes}(S_x, M_z) \cap \mathit{prefixes}(S_y, M_z) &\subseteq \alpha M_z \qquad \text{(a)} \\ \mathit{prefixes}(S_x, M_z) \cap (S_y \cup S_\top) &\subseteq \alpha M_z \qquad \text{(b)} \\ \mathit{prefixes}(S_y, M_z) \cap (S_x \cup S_\top) &\subseteq \alpha M_z \qquad \text{(c)} \\ \mathit{prefixes}(D_z, M_z) \cap \mathit{prefixes}(S_{\bar{z}}, M_z) &\subseteq \alpha M_z \qquad \text{(d)} \\ \mathit{prefixes}(D_z, M_z) \cap (S_{\bar{z}} \cup S_\top) &\subseteq \alpha M_z. \qquad \text{(e)} \end{aligned}$$

Property **A2** is increasing with respect to $S_\top$; all the other properties are decreasing. For this reason, we can assume that $S_\top = \alpha \overline{F} \setminus (S_x \cup S_y)$. This is the smallest $S_\top$ allowed by Property **A2**, making it automatically satisfied.

```
1  function PermanentAgreement(A, F, M, T)
2    for all D_x ∈ rule(T) do
3      for all D_y ∈ rule(T) do
4        for all F ∈ F do
5          for all M, M_x, M_y ∈ M do
6            if M ⊆ F then
7              iteratively compute the least fixed point ⟨S_x, S_y⟩ of (3.3)
8              if computed S_x and S_y satisfy (3.4) then
9                return FALSE
10   return TRUE
```

Figure 3.3: The algorithm for testing Permanent Agreement.

To eliminate $S_\top$ from the other properties, notice that for any set X

$$X \cap (S_z \cup S_\top) \subseteq \alpha M \quad \Longleftrightarrow \quad X \cap \alpha\overline{F} \cap \overline{S_{\bar{z}}} \cap \alpha\overline{M} = \emptyset \quad \Longleftrightarrow \quad X \cap \alpha\overline{F} \cap \alpha\overline{M} \subseteq S_{\bar{z}}.$$

Thus, Properties **A1**(bc) and **A3**(bce) can be rewritten as

$$\begin{aligned} &S_z \supseteq \mathit{prefixes}(S_z, M_*) \cap \alpha\overline{F} \cap \alpha\overline{M_*} \quad \text{for all } M_* \in \{M, M_x, M_y\}, \\ &S_z \supseteq \mathit{prefixes}(D_z, M_z) \cap \alpha\overline{F} \cap \alpha\overline{M_z}. \end{aligned} \tag{3.3}$$

The right-hand side of each of these inequalities is an increasing function of S_z. As a result, we can compute the smallest sets S_z that satisfy these inequalities using Tarski's least fixed point algorithm [115]. Then, it is sufficient to check Properties **A1**(a) and **A3**(ad) for the computed S_x and S_y, that is, whether

$$\begin{aligned} &\mathit{prefixes}(S_x, M_*) \cap \mathit{prefixes}(S_y, M_*) \subseteq \alpha M_* \quad \text{for all } M_* \in \{M, M_x, M_y\}, \\ &\mathit{prefixes}(D_z, M_z) \cap \mathit{prefixes}(S_{\bar{z}}, M_z) \subseteq \alpha M_z. \end{aligned} \tag{3.4}$$

If this is the case, then we have found a state S for which Permanent Agreement does not hold. If not, then the above statement will be false for all supersets of S_x and S_y because function *prefixes* is increasing. As a result, for a given $(F, M, M_x, D_x, M_y, D_y)$, there is no state S violating Permanent Agreement, so this property holds. The complete Permanent Agreement testing algorithm is shown in Figure 3.3.

Example 1

Consider a system consisting of five acceptors, out of which at most one is (maliciously) faulty. Consider an OTC protocol that decides in one communication step if all correct

acceptors proposed the same value.

$$\mathcal{D} = \left\{ \begin{matrix} \{a_1, a_2, a_3, a_4\}, \{a_1, a_2, a_3, a_5\}, \{a_1, a_2, a_4, a_5\} \\ \{a_1, a_3, a_4, a_5\}, \{a_2, a_3, a_4, a_5\} \end{matrix} \right\}.$$

We will use the algorithm in Figure 3.3 to find a state S that violates Permanent Agreement. Consider

$$M = F = \{a_3\}, \quad M_x = \{a_4\}, \quad M_y = \{a_2\}, \quad D_x = \{a_1, a_2, a_3, a_4\}, \quad D_y = \{a_2, a_3, a_4, a_5\}.$$

Since we are only interested in one-step decision, we can limit our attention to events $\langle x : \alpha \rangle$ with the sequence α containing at most one acceptor, which results in $\alpha\overline{F} = \{a_1, a_2, a_4, a_5\}$.

To find the state $S = \langle x : S_x \rangle \cup \langle y : S_y \rangle \cup \langle \top : S_\top \rangle$, we will first use (3.3) to compute S_x:

$$\begin{gathered} S_x \supseteq \mathit{prefixes}(D_x, M_x) \cap \alpha\overline{F} \cap \alpha\overline{M_x} = \{\varepsilon, a_1, a_2, a_3, a_4\} \cap \alpha\overline{F} \cap \alpha\overline{M}_x = \{a_1, a_2\}, \\ S_x \supseteq \mathit{prefixes}(S_x, M) \cap \alpha\overline{F} \cap \alpha\overline{M} = \{\varepsilon, a_1, a_2\} \cap \alpha\overline{F} \cap \alpha\overline{M} = \{a_1, a_2\}, \\ S_x \supseteq \mathit{prefixes}(S_x, M_x) \cap \alpha\overline{F} \cap \alpha\overline{M_x} = \{\varepsilon, a_1, a_2\} \cap \alpha\overline{F} \cap \alpha\overline{M_x} = \{a_1, a_2\}, \\ S_x \supseteq \mathit{prefixes}(S_x, M_y) \cap \alpha\overline{F} \cap \alpha\overline{M_y} = \{\varepsilon, a_1, a_2\} \cap \alpha\overline{F} \cap \alpha\overline{M_y} = \{a_1\}, \end{gathered}$$

which means that $S_x = \{a_1, a_2\}$. Similarly, we can show that $S_y = \{a_4, a_5\}$ and $S_\top = \alpha\overline{F} \setminus \big(S_x \cup S_y\big) = \emptyset$, which leads to $S = \{\langle x : a_1 \rangle, \langle x : a_2 \rangle, \langle y : a_4 \rangle, \langle y : a_5 \rangle\}$. To test whether S violates Permanent Agreement, we test inequalities (3.4):

$$\begin{gathered} \mathit{prefixes}(S_x, M) \cap \mathit{prefixes}(S_y, M) = \{\varepsilon, a_1, a_2\} \cap \{\varepsilon, a_4, a_5\} = \{\varepsilon\} \subseteq \alpha M, \\ \mathit{prefixes}(S_x, M_x) \cap \mathit{prefixes}(S_y, M_x) = \{\varepsilon, a_1, a_2\} \cap \{\varepsilon, a_4, a_5\} = \{\varepsilon\} \subseteq \alpha M_x, \\ \mathit{prefixes}(S_x, M_y) \cap \mathit{prefixes}(S_y, M_y) = \{\varepsilon, a_1, a_2\} \cap \{\varepsilon, a_4, a_5\} = \{\varepsilon\} \subseteq \alpha M_y, \\ \mathit{prefixes}(D_x, M_x) \cap \mathit{prefixes}(S_y, M_x) = \{\varepsilon, a_1, a_2, a_3, a_4\} \cap \{\varepsilon, a_4, a_5\} = \{\varepsilon, a_4\} \subseteq \alpha M_x, \\ \mathit{prefixes}(D_y, M_y) \cap \mathit{prefixes}(S_x, M_y) = \{\varepsilon, a_2, a_3, a_4, a_5\} \cap \{\varepsilon, a_1, a_2\} = \{\varepsilon, a_2\} \subseteq \alpha M_y. \end{gathered}$$

All these inequalities hold, therefore, S violates Permanent Agreement.

Let us now present the above example in natural language. Consider a run in which acceptors a_1, a_2 proposed x, acceptors a_4, a_5 proposed y, and the faulty acceptor a_3 does not propose anything. Assume all correct acceptors executed *stop*, and consider a learner in a complete state $S = \{\langle x : a_1 \rangle, \langle x : a_2 \rangle, \langle y : a_4 \rangle, \langle y : a_5 \rangle\}$. The learner does not know which acceptor is faulty. If acceptor a_3 is correct but slow and proposed x, and a_4 is malicious, then some other learner might see all four acceptors a_1, a_2, a_3, a_4 report x, and decide on x. Therefore, $\mathit{possible}_S(x)$ must hold at the original learner. Similarly, if acceptor a_3 was correct but slow and proposed y, and a_2 malicious, then some learner might

see all four acceptors a_2, a_3, a_4, a_5, report y, and decide on y. Therefore, $possible_S(y)$ must hold as well. This violates Permanent agreement because both $possible_S(x)$ and $possible_S(y)$ hold in a complete state S.

Example 2

As in the previous example, consider a system consisting of five acceptors, out of which at most one is (maliciously) faulty. This time, we will investigate Permanent Agreement of an OTC algorithm that decides in two steps on the value proposed by a_3, provided that a_3 is correct. We have,

$$\mathcal{D} = \left\{ \begin{array}{l} \{a_3, a_3a_1, a_3a_2, a_3a_3, a_3a_4\}, \{a_3, a_3a_1, a_3a_2, a_3a_3, a_3a_5\} \\ \{a_3, a_3a_1, a_3a_3, a_3a_4, a_3a_5\}, \{a_3, a_3a_2, a_3a_3, a_3a_4, a_3a_5\} \end{array} \right\}.$$

This algorithm obviously violates any form of Validity because a_3 can lie about its proposal. However, we will see that this algorithm does not violate Permanent Agreement. We will show this only for

$$M = F = \{a_3\}, \quad M_x = \{a_4\}, \quad M_y = \{a_2\},$$
$$D_x = \{a_3, a_3a_1, a_3a_2, a_3a_3, a_3a_4\}, \quad D_y = \{a_3, a_3a_2, a_3a_3, a_3a_4, a_3a_5\}.$$

The other cases of (M, F, M_x, M_y, D_x, D_y) can be checked in a similar way. Since we are only interested in two-step decision, we do not consider sequences containing more than two acceptors.

First, we use inequalities (3.3) to compute S_x:

$$S_x \supseteq \mathit{prefixes}(D_x, M_x) \cap \alpha\overline{F} \cap \alpha\overline{M_x} = \{\varepsilon, a_3, a_3a_1, \ldots, a_3a_4\} \cap \alpha\overline{F} \cap \alpha\overline{M_x} = \{a_3a_1, a_3a_2\},$$
$$S_x \supseteq \mathit{prefixes}(S_x, M) \cap \alpha\overline{F} \cap \alpha\overline{M} = \{a_3, a_3a_1, a_3a_2\} \cap \alpha\overline{F} \cap \alpha\overline{M} = \{a_3a_1, a_3a_2\},$$
$$S_x \supseteq \mathit{prefixes}(S_x, M_x) \cap \alpha\overline{F} \cap \alpha\overline{M_x} = \{\varepsilon, a_3, a_3a_1, a_3a_2\} \cap \alpha\overline{F} \cap \alpha\overline{M_x} = \{a_3a_1, a_3a_2\},$$
$$S_x \supseteq \mathit{prefixes}(S_x, M_y) \cap \alpha\overline{F} \cap \alpha\overline{M_y} = \{\varepsilon, a_3, a_3a_1, a_3a_2\} \cap \alpha\overline{F} \cap \alpha\overline{M_y} = \{a_3a_1\},$$

which means that $S_x = \{a_3a_1, a_3a_2\}$. Similarly, $S_y = \{a_3a_4, a_3a_5\}$. Now, we use inequalities (3.4) to test whether S_x and S_y violate Permanent Agreement.

$$\mathit{prefixes}(S_x, M) \cap \mathit{prefixes}(S_y, M) = \{\varepsilon, a_3, a_3a_1, a_3a_2\} \cap \{\varepsilon, a_3, a_3a_4, a_3a_5\} = \{\varepsilon, a_3\} \subseteq \alpha M,$$
$$\mathit{prefixes}(S_x, M_x) \cap \mathit{prefixes}(S_y, M_x) = \{\varepsilon, a_3, a_3a_1, a_3a_2\} \cap \{\varepsilon, a_3, a_3a_4, a_3a_5\} = \{\varepsilon, a_3\} \not\subseteq \alpha M_x.$$

The second inequality does not hold, so we do not have to check the others; Permanent Agreement is not violated in this state.

Example 3

Consider a system of three acceptors, out of which at most one is (non-maliciously) faulty. Consider an OTC algorithm that decides (i) in one step if all acceptors are correct and propose the same value, and (ii) in two steps if a_1 is correct. We have

$$\mathcal{D} = \{\{a_1, a_2, a_3\}, \{a_1, a_1a_1, a_1a_2\}, \{a_1, a_1a_1, a_1a_3\}\}.$$

Consider

$$F = \{a_1\}, \qquad M = M_x = M_y = \emptyset, \qquad D_x = \{a_1, a_2, a_3\}, \qquad D_y = \{a_1, a_1a_1, a_1a_2\}.$$

The other cases of (M, F, M_x, M_y, D_x, D_y) can be checked in a similar way. Since we are only interested in deciding in at most two steps, we do not consider sequences containing more than two acceptors.

First, we use inequalities (3.3) to compute S_x:

$$S_x \supseteq \mathit{prefixes}(D_x, M_x) \cap \alpha\overline{F} \cap \alpha\overline{M_x} = \{\varepsilon, a_1, a_2, a_3\} \cap \alpha\overline{F} \cap \alpha\overline{M}_x = \{a_2, a_3\},$$
$$S_y \supseteq \mathit{prefixes}(D_y, M_y) \cap \alpha\overline{F} \cap \alpha\overline{M_y} = \{\varepsilon, a_1, a_1a_1, a_1a_2\} \cap \alpha\overline{F} \cap \alpha\overline{M}_x = \{a_1a_2\}.$$

Since no acceptors are malicious, it can be easily checked that $S_x = \{a_2, a_3\}$ and $S_y = \{a_1a_2\}$ satisfy all inequalities (3.3). Now, we use inequalities (3.4) to test whether S_x and S_y violate Permanent Agreement.

$$\mathit{prefixes}(S_x, M_*) \cap \mathit{prefixes}(S_y, M_*) = \{\varepsilon, a_2, a_3\} \cap \{\varepsilon, a_1, a_1a_2\} = \{\varepsilon\} \subseteq \alpha M_*,$$
$$\mathit{prefixes}(D_x, M_x) \cap \mathit{prefixes}(S_y, M_x) = \{\varepsilon, a_1, a_2, a_3\} \cap \{\varepsilon, a_1, a_1a_2\} = \{\varepsilon, a_1\} \nsubseteq \alpha M_x,$$
$$\mathit{prefixes}(D_y, M_y) \cap \mathit{prefixes}(S_x, M_y) = \{\varepsilon, a_1, a_1a_1, a_1a_2\} \cap \{\varepsilon, a_2, a_3\} = \{\varepsilon\} \subseteq \alpha M_y.$$

The second inequality does not hold, so S_x and S_y do not violate Permanent Agreement.

3.5 Discovering new OTC algorithms

Section 3.4 presented algorithms that test whether an OTC algorithm satisfies Permanent Validity and Permanent Agreement. An OTC algorithm is specified by the set $\mathcal{T}$ of termination rules, whereas the system is specified by the set A of acceptors, the family $\mathcal{F}$ of possible sets of faulty acceptors, and the family $\mathcal{M}$ of possible sets of malicious acceptors.

```
1  function OTCSearch(A, F, M, T) is
2    if PermanentValidity(A, F, M, T) and PermanentAgreement(A, F, M, T) then
3      print T
4      for all possible termination rules t do
5        if t ∉ T do
6          OTCSearch(A, F, M, T ∪ {t})
```

Figure 3.4: The basic version of the algorithm for searching the space of OTC protocols.

3.5.1 Basic search

In this section, we will show how to automatically discover new OTC algorithms. We start with an empty set $\mathcal{T}$ of termination rules and keep recursively adding new rules as long as Permanent Validity and Permanent Agreement hold.

This method is implemented by the function *OTCSearch* in Figure 3.4. First, we test whether the OTC algorithm $\mathcal{T}$ is correct in a system specified by A, $\mathcal{F}$, $\mathcal{M}$. If not, then the function *OTCSearch* returns immediately; adding new rules to an incorrect OTC algorithm $\mathcal{T}$ cannot produce a correct one. If the algorithm $\mathcal{T}$ is correct, then the set $\mathcal{T}$ is printed out. Then, we iterate over all possible termination rules t. For each such rule, we invoke *OTCSearch* recursively with the rule t added to $\mathcal{T}$.

The algorithm shown in Figure 3.4 searches for multi-value OTC algorithms, in which different acceptors can propose different values. To look for single-value OTC algorithms, the check for Permanent Agreement should be omitted.

3.5.2 Search optimization

A number of techniques can be applied to improve the speed of the search algorithm in Figure 3.4. In this section, we will briefly discuss some of them.

Rule order

The order of the termination rules in $\mathcal{T}$ does not matter. However, the algorithm in Figure 3.4 adds new rules to $\mathcal{T}$ in a specific order, and as a result the same set of rules is analyzed many times. For example, set $\{t_1, t_2\}$ can be obtained in two ways: either by first adding t_1, and then t_2, or vice versa. Similarly, set $\{t_1, t_2, t_3\}$ can be obtained in six different ways. In general, $\{t_1, \ldots, t_n\}$ will be analyzed $n!$ times, slowing the algorithm down exponentially.

To ensure that each set $\mathcal{T}$ is generated and analysed only once, consider any total order "$<$" on termination rules. We will modify the algorithm in Figure 3.4 and require new elements to be added to $\mathcal{T}$ in an order consistent with "$<$". If we assume, in our example, that $t_1 < t_2 < t_3$, then $\{t_1, t_2, t_3\}$ can be obtained only by adding t_1, t_2, t_3 in this

```
1  function OTCSearch(A, F, M, T) is
2    if PermanentValidity(A, F, M, T) and PermanentAgreement(A, F, M, T) then
3      print T
4      for all possible termination rules t do
5        if t is bigger (">") than all elements of T and
6           t does not dominate any element of T and
7           t is not dominated by any element of T then
8             OTCSearch(A, F, M, T ∪ {t})
```

Figure 3.5: The optimized version of the algorithm for searching the space of OTC protocols.

order. The sequence t_1, t_3, t_2 will not work; after $\mathcal{T} = \{t_1, t_3\}$ has been created, adding t_2 is impossible because $t_2 < t_3 \in \mathcal{T}$.

Rule domination

Some termination rules are stronger than others. For example, consider a three-acceptor system with the following two termination rules:

$$t_1 = \langle\{a_1, a_2\}, \{a_1, a_2\}, 1\rangle \quad \text{and} \quad t_2 = \langle\{a_1, a_2, a_3\}, \{a_1, a_2, a_3\}, 2\rangle.$$

Rule t_1 demands a decision in one communication step if acceptors a_1 and a_2 are correct and proposed the same value. Rule t_2 requires a decision in two communication steps if all acceptors are correct and proposed the same value. It is obvious that every OTC algorithm satisfying rule t_1 also satisfies t_2; we say that rule t_1 *dominates* t_2. Formally,

$$t_1 \text{ dominates } t_2 \quad \stackrel{\text{def}}{\iff} \quad \mathit{rule}(t_1) \subseteq \mathit{rule}(t_2),$$

that is, the domination relation on termination rules reflects the subset relation on corresponding decision rules (Section 3.3.4). Equivalently,

$$\langle V_1, C_1, k_1\rangle \text{ dominates } \langle V_2, C_2, k_2\rangle \quad \iff \quad V_1 \subseteq V_2 \ \wedge \ C_1 \subseteq C_2 \ \wedge \ k_1 \leq k_2.$$

Since, in our example, rule t_1 dominates t_2, OTC algorithms corresponding to the sets $\mathcal{T}_1 = \{t_1\}$ and $\mathcal{T}_{12} = \{t_1, t_2\}$ are the same, so analyzing both of them is a waste of time. To avoid this, we will refrain from analyzing sets $\mathcal{T}$ that contain a pair of rules such that one dominates the other.

Implementation

Figure 3.5 shows the version of *OTCSearch* that employs the optimizations described above. It differs from the basic version in the **if** statement in lines 5–7. In Figure 3.4, this

if statement tests merely whether the new rule t already belongs to $\mathcal{T}$. The **if** statement in Figure 3.5 is stricter; it requires t to be bigger than all elements of $\mathcal{T}$, and not to dominate or be dominated by any of them.

3.5.3 Results

This section presents the results from applying the above algorithms to four-acceptor systems in which one acceptor can fail. We consider two common settings: the crash-stop model and the Byzantine model. For both cases, we present the list of correct OTC implementations computed by *OTCSearch* implemented in C and verified by an independent implementation in Python [107]. We do not list all correct OTC algorithms; we omit those that can be obtained from others by permuting the set of acceptors. Also, we eliminate algorithms that are clearly inferior to others. In other words, we list only those algorithms that are not dominated by other correct OTC algorithms. (A set of rules $\mathcal{T}$ is dominated by another set of rules $\mathcal{T}'$ iff every rule in $\mathcal{T}$ is dominated by some rule in $\mathcal{T}'$.)

Crash-stop model

Consider a system consisting of four honest acceptors, out of which at most one is faulty:

$$A = \{a_1, a_2, a_3, a_4\}, \quad \mathcal{F} = \{\emptyset, \{a_1\}, \{a_2\}, \{a_3\}, \{a_4\}\}, \quad \mathcal{M} = \{\emptyset\}.$$

The one-step multi-value OTC implementation from Section 2.3 guarantees one-step decision if all correct acceptors propose the same value, that is,

$$\mathcal{T} = \left\{ \begin{array}{l} \langle\{a_1, a_2, a_3\}, \{a_1, a_2, a_3\}, 1\rangle \\ \langle\{a_1, a_2, a_4\}, \{a_1, a_2, a_4\}, 1\rangle \\ \langle\{a_1, a_3, a_4\}, \{a_1, a_3, a_4\}, 1\rangle \\ \langle\{a_2, a_3, a_4\}, \{a_2, a_3, a_4\}, 1\rangle \end{array} \right\}.$$

OTCSearch$(A, \mathcal{F}, \mathcal{M}, \emptyset)$ produced six correct OTC implementations for these settings.

$$\mathcal{T}_1 = \left\{ \begin{array}{c} \langle\{a_1, a_2, a_3\}, \{a_1, a_2, a_3\}, 1\rangle \\ \langle\{a_1, a_2, a_4\}, \{a_1, a_2, a_4\}, 1\rangle \\ \langle\{a_1, a_3, a_4\}, \{a_1, a_3, a_4\}, 1\rangle \\ \langle\{a_2, a_3, a_4\}, \{a_2, a_3, a_4\}, 1\rangle \\ \langle\{a_1, a_2\}, \{a_1, a_2\}, 2\rangle \\ \langle\{a_1, a_4\}, \{a_1, a_4\}, 2\rangle \\ \langle\{a_1, a_3\}, \{a_1, a_3\}, 2\rangle \end{array} \right\} \quad \text{and} \quad \mathcal{T}_2 = \left\{ \begin{array}{c} \langle\{a_1, a_2, a_3\}, \{a_1, a_2, a_3\}, 1\rangle \\ \langle\{a_1, a_2, a_4\}, \{a_1, a_2, a_4\}, 1\rangle \\ \langle\{a_1, a_3, a_4\}, \{a_1, a_3, a_4\}, 1\rangle \\ \langle\{a_2, a_3, a_4\}, \{a_2, a_3, a_4\}, 1\rangle \\ \langle\{a_1, a_2\}, \{a_1, a_2\}, 2\rangle \\ \langle\{a_1, a_3\}, \{a_1, a_3\}, 2\rangle \\ \langle\{a_2, a_3\}, \{a_2, a_3\}, 2\rangle \end{array} \right\}.$$

Algorithms $\mathcal{T}_1$ and $\mathcal{T}_2$ both extend $\mathcal{T}$, and guarantee one-step decision if all correct acceptors proposed the same value. In addition, $\mathcal{T}_2$ decides in two steps if any two of the first three acceptors are correct and propose the same value. On the other hand, $\mathcal{T}_1$ decides in two steps if a_1 and one other acceptor are correct and propose the same value.

$$\mathcal{T}_3 = \left\{ \begin{array}{c} \langle\{a_1\},\{a_1,a_2\},2\rangle \\ \langle\{a_1\},\{a_1,a_3\},2\rangle \\ \langle\{a_1\},\{a_1,a_4\},2\rangle \\ \langle\{a_1,a_2\},\{a_1,a_2\},1\rangle \end{array} \right\} \quad \text{and} \quad \mathcal{T}_4 = \left\{ \begin{array}{c} \langle\{a_1\},\{a_1,a_2\},2\rangle \\ \langle\{a_1\},\{a_1,a_3\},2\rangle \\ \langle\{a_1\},\{a_1,a_4\},2\rangle \\ \langle\{a_1,a_2,a_3\},\{a_1,a_2,a_3\},1\rangle \\ \langle\{a_1,a_2,a_4\},\{a_1,a_2,a_4\},1\rangle \\ \langle\{a_1,a_3,a_4\},\{a_1,a_3,a_4\},1\rangle \end{array} \right\},$$

The first three rules ensure that both algorithms $\mathcal{T}_3$ and $\mathcal{T}_4$ decide in two communication steps if a_1 is correct, regardless of the proposals. Besides, $\mathcal{T}_3$ decides in one step if a_1 and a_2 are correct and propose the same value. Algorithm $\mathcal{T}_4$ decides in one step if a_1 and two other acceptors are correct and propose the same value.

The other two OTC algorithms are:

$$\mathcal{T}_5 = \left\{ \begin{array}{c} \langle\{a_1,a_2\},\{a_1,a_2\},1\rangle \\ \langle\{a_1,a_3\},\{a_1,a_3\},2\rangle \\ \langle\{a_2,a_3\},\{a_2,a_3\},2\rangle \end{array} \right\} \quad \text{and} \quad \mathcal{T}_6 = \left\{ \begin{array}{c} \langle\{a_1,a_2\},\{a_1,a_2\},1\rangle \\ \langle\{a_1,a_3\},\{a_1,a_3\},2\rangle \\ \langle\{a_1,a_4\},\{a_1,a_4\},2\rangle \\ \langle\{a_2,a_3,a_4\},\{a_2,a_3,a_4\},2\rangle \end{array} \right\}.$$

Byzantine model

Consider a system consisting of four acceptors, out of which at most one is maliciously faulty:

$$A = \{a_1,a_2,a_3,a_4\}, \quad \mathcal{F} = \mathcal{M} = \{\emptyset,\{a_1\},\{a_2\},\{a_3\},\{a_4\}\}.$$

The multi-step multi-value OTC implementation from Section 2.6 guarantees two-step decision if all correct acceptors propose the same value. If all acceptors are correct and propose the same value, the decision is made in one step:

$$\mathcal{T} = \left\{ \begin{array}{c} \langle\{a_1,a_2,a_3,a_4\},\{a_1,a_2,a_3,a_4\},1\rangle \\ \langle\{a_1,a_2,a_3\},\{a_1,a_2,a_3\},2\rangle \\ \langle\{a_1,a_2,a_4\},\{a_1,a_2,a_4\},2\rangle \\ \langle\{a_1,a_3,a_4\},\{a_1,a_3,a_4\},2\rangle \\ \langle\{a_2,a_3,a_4\},\{a_2,a_3,a_4\},2\rangle \end{array} \right\}.$$

Calling $OTCSearch(A,\mathcal{F},\mathcal{M},\emptyset)$ produced five correct OTC implementations for these

settings.

$$\mathcal{T}_1 = \left\{ \begin{array}{c} \langle\{a_1,a_2,a_3,a_4\},\{a_1,a_2,a_3,a_4\},1\rangle \\ \langle\{a_1,a_2,a_3\},\{a_1,a_2,a_3\},2\rangle \\ \langle\{a_1,a_2,a_4\},\{a_1,a_2,a_4\},2\rangle \\ \langle\{a_1,a_3,a_4\},\{a_1,a_3,a_4\},2\rangle \\ \langle\{a_2,a_3,a_4\},\{a_2,a_3,a_4\},2\rangle \\ \langle\{a_1,a_2\},\{a_1,a_2,a_3,a_4\},2\rangle \\ \langle\{a_1,a_3\},\{a_1,a_2,a_3,a_4\},2\rangle \\ \langle\{a_2,a_3\},\{a_1,a_2,a_3,a_4\},2\rangle \end{array} \right\}, \quad \mathcal{T}_2 = \left\{ \begin{array}{c} \langle\{a_1,a_2,a_3,a_4\},\{a_1,a_2,a_3,a_4\},1\rangle \\ \langle\{a_1,a_2,a_4\},\{a_1,a_2,a_4\},2\rangle \\ \langle\{a_1,a_3,a_4\},\{a_1,a_3,a_4\},2\rangle \\ \langle\{a_2,a_3,a_4\},\{a_2,a_3,a_4\},2\rangle \\ \langle\{a_1,a_2,a_3\},\{a_1,a_2,a_3\},2\rangle \\ \langle\{a_1,a_2\},\{a_1,a_2,a_3,a_4\},2\rangle \\ \langle\{a_1,a_3\},\{a_1,a_2,a_3,a_4\},2\rangle \\ \langle\{a_1,a_4\},\{a_1,a_2,a_3,a_4\},2\rangle \end{array} \right\}.$$

Both $\mathcal{T}_1$ and $\mathcal{T}_2$ extend $\mathcal{T}$; the first five rules in these algorithm are the same. The last three rules in $\mathcal{T}_1$ and $\mathcal{T}_2$ assume that all acceptors are correct. Algorithm $\mathcal{T}_1$ guarantees that if two acceptors from $\{a_1, a_2, a_3\}$ propose the same value, then the decision is made in two steps. Algorithm $\mathcal{T}_2$ makes a two-step decision provided that a_1 and one other acceptor propose the same value.

The other three OTC algorithms are:

$$\left\{ \begin{array}{c} \langle\{a_1,a_2\},\{a_1,a_2,a_3\},2\rangle \\ \langle\{a_1,a_3\},\{a_1,a_3,a_4\},2\rangle \\ \langle\{a_1,a_4\},\{a_1,a_2,a_4\},2\rangle \\ \langle\{a_1,a_2\},\{a_1,a_2,a_4\},3\rangle \\ \langle\{a_1,a_3\},\{a_1,a_2,a_3\},3\rangle \\ \langle\{a_1,a_4\},\{a_1,a_3,a_4\},3\rangle \\ \langle\{a_2,a_3,a_4\},\{a_2,a_3,a_4\},2\rangle \end{array} \right\}, \quad \left\{ \begin{array}{c} \langle\{a_1,a_2\},\{a_1,a_2,a_3\},2\rangle \\ \langle\{a_1,a_2\},\{a_1,a_2,a_4\},2\rangle \\ \langle\{a_1,a_3\},\{a_1,a_2,a_3\},3\rangle \\ \langle\{a_1,a_3\},\{a_1,a_3,a_4\},2\rangle \\ \langle\{a_1,a_4\},\{a_1,a_2,a_4\},3\rangle \\ \langle\{a_1,a_4\},\{a_1,a_3,a_4\},3\rangle \\ \langle\{a_1,a_4\},\{a_1,a_2,a_3,a_4\},2\rangle \\ \langle\{a_2,a_3,a_4\},\{a_2,a_3,a_4\},2\rangle \end{array} \right\},$$

$$\left\{ \begin{array}{c} \langle\{a_1,a_2\},\{a_1,a_2,a_3\},2\rangle \\ \langle\{a_1,a_2\},\{a_1,a_2,a_4\},2\rangle \\ \langle\{a_1,a_3\},\{a_1,a_2,a_3\},3\rangle \\ \langle\{a_1,a_3\},\{a_1,a_3,a_4\},2\rangle \\ \langle\{a_2,a_3\},\{a_1,a_2,a_3\},3\rangle \\ \langle\{a_2,a_3\},\{a_2,a_3,a_4\},2\rangle \end{array} \right\}.$$

3.6 Conclusion and future work

In this chapter, we introduced a method for automatic testing and discovery of OTC algorithms. Automatic testing means that we can check the correctness of any OTC algorithm candidate expressible in our framework. A positive result proves that the given algorithm satisfies all OTC properties from Section 2.2. A negative result shows a state in which one of the OTC properties is violated. Negative results are useful in two cases. Firstly, in the algorithm design process, to understand why a given OTC algorithm is incorrect. Secondly, negative results can often be generalized to impossibility theorems.

Automatic discovery of OTC algorithms allows us to skip the manual algorithm design process altogether. Instead of using automatic correctness testing to verify individual OTC algorithms, a user just specifies a set of requirements. The discovery method presented in this chapter searches the solution space for OTC algorithms that meet the given criteria. Chapter 4 will show how to use both manually and automatically generated OTC algorithms to construct efficient solutions for distributed agreement problems such as Consensus or Atomic Commitment.

Our correctness testing method is built around an execution model based on events of the form $\langle x : e_1 e_2 \dots e_k \rangle$, where x is a proposal and $e_1 e_2 \dots e_k$ is a list of acceptors. We have developed a formalism for reasoning about events and sets of events, which we call states. We used this formalism to define predicates $\mathit{valid}(x)$, $\mathit{possible}(x)$, and $\mathit{decision}(x)$ that satisfy OTC properties Integrity, Possibility, and Optimistic Termination. These predicates can then be used to test the other two OTC properties: Permanent Validity and Permanent Agreement, thereby verifying correctness of a given OTC algorithm candidate. Finally, generating OTC algorithm candidates and using the above correctness-testing method allows us to discover new OTC algorithms.

Our method assumes that termination rules apply to all proposed values equally. From a mathematical standpoint, it is not difficult to waive this assumption and consider a separate set of termination rules for every value x. This approach is not practical, however; it results in a huge, and potentially infinite, number of rules. As a compromise, our C implementation is capable of distinguishing two families of rules: those applying to all proposals and those applying only to the privileged value x_0. The only modification needed is that the algorithm in Figure 3.3 does not check for Permanent Agreement violations caused by decision rules D_x and D_y which both belong to the second category.

Another possible extension of our model is to allow acceptors to digitally sign some of their messages. Again, the modification of our method required in this case is small and requires only a change in the *prefixes* function. Without digital signatures, $e_1 e_2 \dots e_i \in \mathit{prefixes}(e_1 e_2 \dots e_k, M)$ iff acceptors $e_{i+1}, \dots, e_k$ are all honest. With digital signatures, $e_1 e_2 \dots e_i \in \mathit{prefixes}(e_1 e_2 \dots e_k, M)$ also if $e_1 e_2 \dots e_i$ is signed by $e_{i+1} \notin M$.

Chapter 4

Implementing agreement abstractions

In Chapter 1, we observed that asynchronous Consensus algorithms share the same structure; they consist of a sequence of rounds, each starting with a coordinator process broadcasting its proposal to the acceptors. The acceptors then *somehow* try to make the learners decide on it; the exact method depends on the Consensus algorithm.

In Chapter 2, we encapsulated the heart of each round into a new abstraction called Optimistically Terminating Consensus (OTC). We also presented several OTC algorithms, which can be used to match the latency and acceptor requirements of most known Consensus protocols. Chapter 3 extended this work by developing a method for discovering new OTC algorithms automatically. This chapter deals with using sequences of OTC instances to implement agreement abstractions. We will formalize the idea described in Section 2.2.1 and give latency-optimal implementations of several variants of Consensus as well as other agreement abstractions.

This chapter is structured in the following way. Section 4.1 will introduce the Coordinated Consensus abstraction. Sections 4.2 and 4.3 will show how to use OTC instances to implement Coordinated Consensus in the crash-stop model and the Byzantine model, respectively. In Section 4.4, we will show how to use this abstraction to implement several agreement abstractions such as Consensus, Atomic Commitment [48], and Interactive Consistency [97]. Finally, Section 4.5 compares OTC with other frameworks.

4.1 Coordinated Consensus

Before giving a precise description of an OTC-based Consensus algorithm, we must give a precise definition of the problem being solved. In this section, we will introduce a new agreement abstraction called *Coordinated Consensus*, similar to the definition of Consensus given by Lamport [76]. The next sections will show how to use Coordinated

Consensus to implement simple solutions to many common agreement problems such as Consensus, Interactive Consistency, and Atomic Commitment.

Coordinated Consensus consists of a sequence of rounds. Each round i has its coordinator c_i, which issues a proposal and broadcasts it to the acceptors. Acceptors cooperate in reaching a decision and making it known to the learners.

Formally, Coordinated Consensus is defined in terms of two primitives: *propose*(x) and *decision*(x), available to coordinators and learners, respectively. Each coordinator c_i proposes a value x by invoking *propose*(x). We say that a learner decided on x if the predicate *decision*(x) holds at that learner.

Coordinated Consensus is defined by the following properties:

Validity. If all coordinators are honest and *decision*(x) holds at some learner, then some coordinator proposed x.

Agreement. There is at most one x for which *decision*(x) holds at some learner.

Termination. If infinitely many c_i are correct and eventually all of them propose, then all correct learners will eventually decide.

While the Coordinated Consensus is similar to the Consensus problem as defined in [36, 76], there are minor differences. Our Validity condition assumes that all coordinators are honest. Lamport [76] does not make this assumption and requires that any decision must have always been proposed by some coordinator. Dutta et al. [36] observe that such a condition is impossible to enforce, because a malicious coordinator can propose one value and then behave as if it had proposed another. To avoid this problem, they suggest the following condition: "if a learner l learns a value v in run r, then there is a run r' (possibly different from r) such that some coordinator proposes v in r', and l cannot distinguish r from r'". This definition is not satisfactory either because it is satisfied by a trivial algorithm in which processes send no messages and all learners decide on a pre-agreed value, regardless of the actual proposals.

Both of the above problems are avoided by assuming honest coordinators in the Validity condition. Note that this condition limits possible decisions not only in runs with honest coordinators, but also in runs which are indistinguishable from these. For example, consider a good run r with all coordinators correct and c_1 proposing 1. In this run, all learners will decide on 1 in the first round, without starting any of the later rounds. Now consider a similar run r', with a correct c_1 proposing 1 and c_2 being malicious. Formally, the maliciousness of c_2 allows all learners to decide on 2 instead of 1. However, by the time learners decide in run r, they cannot distinguish r from r'. As a result, they will decide on 1 in run r' as well.

The Agreement condition is common to all agreement problems, and requires that no two learners decide on different values, even if all coordinators are malicious. Termination requires all correct learners to decide if there are infinitely many correct coordinators.

The Termination condition implicitly requires all acceptors to actually start participating in the algorithm. This can happen either explicitly, or implicitly when the acceptor executes an action such as $propose(x)$ or receives a message related to the algorithm. We say that the algorithm has started if at least one correct acceptor has started to participate in it.

The eventual synchrony model uses timeouts to determine when to stop rounds. The explicit notion of the "start" of the algorithm is especially important in this model because it allows us to start the first round timer at the right moment. In both models, it "shields" the algorithm from wrong suspicions and delayed messages that happened before the algorithm started.

Recall that a run is timely if no correct process is ever suspected (failure detectors) or the maximum message transmission time d is sufficiently small (eventual synchrony). In the eventual synchrony model, we additionally demand that all processes required to propose by the Termination condition do so within one communication step from the start of the algorithm.

4.2 Coordinated Consensus in the crash-stop model

In this section, we will show how to solve the Coordinated Consensus, assuming that all the processes are honest. This method will be generalized in Section 4.3, where we will present a Coordinated Consensus algorithm that tolerates malicious coordinators and acceptors.

4.2.1 Overview

Our Coordinated Consensus algorithm progresses in a sequence of *rounds*, numbered 1, 2, etc. Initially, the first round tries to decide on some value. If the first round does not seem to make progress, it is stopped, and the second round takes over. If the decision has not been made by the second round, it is stopped as well, and the third rounds starts, etc.

As shown in Figure 4.1, each round i has a coordinator c_i and the corresponding OTC instance OTC_i. Coordinator c_i broadcasts its proposal to the acceptors, who propose it to the instance OTC_i. Since coordinators are honest, all acceptors propose the same value to a given OTC_i, which enables us to use single-value OTC implementations. A decision made by any of these OTC instances becomes the final decision of the Consensus algorithm.

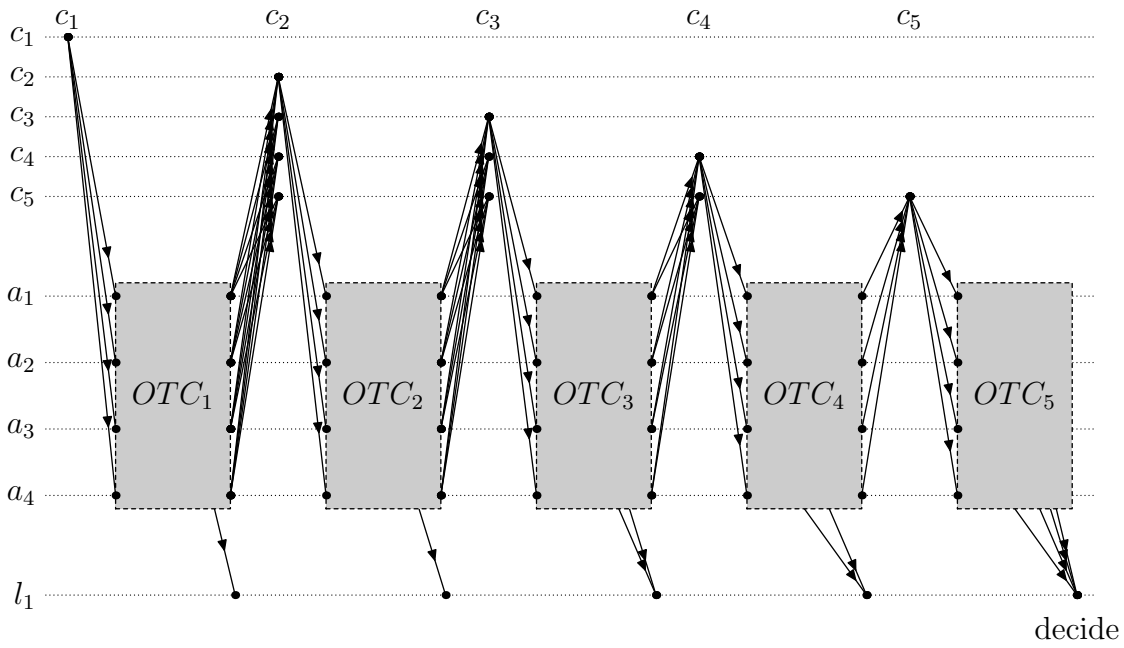


Figure 4.1: Coordinated Consensus.

This method satisfies Validity. If a learner decides on x in some OTC_i, then an honest acceptor proposed x to OTC_i. The acceptor received x from the coordinator c_i, so if c_i is honest, it must have proposed x.

The Agreement property poses a problem, however. Although individual OTC instances satisfy Agreement, the decisions made by different instances might not be the same. For this reason, in each round i, the acceptors always check for possible decisions made in previous rounds. If no decision was made in any of the previous rounds, they take the proposal x_i issued by the round coordinator c_i, and propose it to OTC_i (this is always the case for the first round). Otherwise, if one of the previous rounds could have decided on some value, the acceptors propose this value to OTC_i instead. This method guarantees that decisions made by different rounds will be the same, however, it must be implemented carefully to ensure that Validity still holds.

For Termination, we assume the failure detector model. Acceptors stop a round when they suspect the coordinator. Assuming infinitely many correct coordinators, there will eventually be a round with a correct coordinator not suspected by any of the correct acceptors. In that round, all correct learners will decide.

Traditional failure detectors can only monitor acceptors, not other processes, which forces us to assume that all coordinators are acceptors. If this assumption is not appropriate, one can extend failure detectors to cover processes other than acceptors, or use the eventual synchrony model instead (Section 4.3).

```
1   when coordinator c_i executed propose(x_i) do
2     for all j < i do
3       wait until the state S_j of c_i as a learner in OTC_j is semi-complete
4     broadcast x_i and ⟨S_j⟩_{j<i} to all acceptors
5   when an acceptor receives x_i and ⟨S_j⟩_{j<i} from c_i do
6     x ← choose(⟨S_j⟩_{j<i}, x_i)                          { select the proposal }
7     OTC_i.propose(x)
8   when a learner has OTC_i.decision(x) do                  { alternative 1 }
9     decide(x)
10  when an acceptor stopped all rounds j < i and suspects c_i, or
11        received message "stop round j" from some acceptor do
12    OTC_i.stop
13    broadcast "stop round i"
14  when a learner has OTC_i.decision(x) or received "decide on x" do { alternative 2 }
15    decide(x)
16    if the learner is also an acceptor then
17      broadcast "decide on x"
18    wait until received "decide on x" from more than f acceptors
19    halt
```

Figure 4.2: Coordinated Consensus algorithm for the crash-stop model.

4.2.2 Details

Figure 4.2 shows the details of this algorithm. Each coordinator c_i is a learner in all OTC instances OTC_j with $j < i$, and a proposer in OTC_i. Let us denote by S_j the state of c_i as a learner in OTC_j. When the coordinator c_i issues its proposal x_i, it waits until all states S_j are semi-complete, that is, $valid_{S_j}(x) \implies possible_{S_j}(x)$ for all x, and $possible_{S_j}(x)$ holds for at most one x. Permanent Validity and Permanent Agreement properties of OTC instances ensure that this will eventually happen, provided that all rounds $j < i$ have been stopped. Then, coordinator c_i broadcasts its proposal x_i and the collection of states $\langle S_j \rangle_{j<i}$ to the acceptors.

When an acceptor receives this information, it first tries to find out whether any of the previous rounds might have decided. This reasoning is done by function *choose*, which will be presented in Section 4.2.3. It takes $\langle S_j \rangle_{j<i}$ and x_i as arguments and returns either the value of a possible decision made by some previous round, or x_i if no such decision was made. The acceptor proposes this returned value x to OTC_i. All acceptors receive the same $\langle S_j \rangle_{j<i}$ and x_i, and function *choose* is deterministic, therefore all acceptors propose the same x to OTC_i. If no correct acceptor ever stops round i, and OTC_i satisfies Optimistic Termination $(f, \bullet)$, then all correct learners will eventually decide. This decision becomes the final decision of the Coordinated Consensus algorithm in lines 8–9. Here,

action *decide*(x) makes the predicate *decision*(x) true; formally, *decision*(x) is defined as "*decide*(x) has been called", where *decide*(x) is an empty action without other side-effects.

Agreement and Validity

As we will see in Section 4.2.3, the value $x = choose(\langle S_j \rangle_{j<i}, x_i)$ has two properties. First, if all coordinators are honest, then x has been proposed by one of them, which implies Validity. Second, no decision other than x was made in rounds $j < i$, therefore different rounds cannot make different decisions, which implies Agreement.

Termination

Rounds with faulty coordinators may not make progress, so they must eventually be stopped so that other coordinators can take over and ensure Termination. To detect such rounds, we use the $\Diamond S$ failure detector. Recall, from Section 1.3.2, that $\Diamond S$ is a crash detector with the following properties:

Strong Completeness. Eventually every faulty acceptor is permanently suspected by every correct acceptor.

Eventual Weak Accuracy. Eventually some correct acceptor will never be suspected by any correct acceptor.

Chandra et al. [17] showed that $\Diamond S$ is the weakest failure detector that makes Consensus solvable in asynchronous systems.

To ensure Termination, we make two additional assumptions. Firstly, we assume that every acceptor coordinates infinitely many rounds; for example, in the rotating coordinator method, acceptor a_i coordinates rounds i, $i+n$, $i+2n$, etc. Secondly, we assume that all the OTC instances satisfy Optimistic Termination $(f, \bullet)$. As shown above, this implies that if c_i is correct and no correct acceptor ever stops round i, then all correct learners will eventually decide.

In the algorithm shown in Figure 4.2, an acceptor stops round i, stopped all previous rounds and suspects the coordinator (lines 10–13). In addition, it informs other acceptors by broadcasting a message "stop round i". Every acceptor receiving this message stops round i as well. Therefore, if one correct acceptor stops round i, then all correct acceptors will eventually do so. In other words, a round is stopped by either no or all correct acceptors.

The first step to show Termination is to prove that if it does not hold, then all correct acceptors stop all rounds. To obtain a contradiction, assume this is not true, and let i be the first round which is not stopped by all correct acceptors. The choice of i means that rounds $j < i$ have been stopped by all correct acceptors. Also, the all-or-none property

discussed above, implies that no correct acceptor ever stops round i. If the coordinator c_i is correct, then – as we explained above – the Optimistic Termination $(f, \bullet)$ of OTC_i implies that all correct learners will eventually decide (Termination). On the other hand, if c_i is faulty, then it will eventually become suspected by all correct acceptors (Strong Completeness of $\Diamond S$), who will all execute $OTC_i.stop$. This in turn contradicts the choice of i. Therefore, we have just proved that if Termination does not hold, then all rounds are stopped.

On the other hand, Eventual Weak Accuracy of $\Diamond S$ ensures that at least one correct acceptor a will eventually never be suspected. As all other acceptors, a coordinates infinitely many rounds, so it will eventually coordinate a round in which it will not be suspected, and so that round will never be stopped. The result from the previous paragraph implies Termination.

Halting

So far, we have been assuming that processes never stop executing their code. Halting the algorithm immediately after deciding saves resources but is not always safe. As an example, consider a scenario where the coordinator c_1 and some acceptors are faulty in such a way that only one acceptor decides. If this acceptor halts immediately after deciding, the number of correct acceptors participating in later rounds might not be sufficient to guarantee Termination.

To implement safe halting, we replace lines 8–9 with lines 14–19. Now, acceptors inform other learners about their decisions by broadcasting "decide on x". When a learner receives such a message, it decides on x immediately but waits with halting until it has received more than f of these messages. This ensures that at least one of them comes from a correct acceptor, so it will eventually reach all correct learners. This **wait** instruction can be omitted if the learner does not play any other role in the algorithm, that is, it is neither an acceptor nor a coordinator.

Latency

The Coordinated Consensus algorithm shown in Figure 4.2 satisfies

> **Latency.** If the run is timely, coordinator c_1 is correct, at most q acceptors are faulty, and OTC_1 satisfies Optimistic Termination (q, k), then all correct learners will decide in $k + 1$ communication steps.

Recall that, in the failure detector model, a run is timely if no correct acceptors are ever suspected. Therefore, given the above assumptions, coordinator c_1 is never suspected, so no correct acceptor ever executes $OTC_1.stop$. Coordinator c_1 is correct, so all correct acceptors receive its proposal x in one communication step, and propose it to OTC_1. Now,

since no correct acceptor ever executes $OTC_1.stop$ and at most q acceptors are faulty, Optimistic Termination (q, k) implies the assertion. The number of communication steps necessary to reach a decision is $k + 1$ because one step has been used for c_1 to broadcast its proposal to the acceptors.

For example, the one-step single-value OTC from Section 2.3 with $q = f$ satisfies Optimistic Termination $(f, 1)$, which leads to

> **Latency.** If the run is timely and coordinator c_1 correct, then all correct learners will decide in two steps.

This matches the latency of several known Consensus algorithms [63, 73, 112], and cannot be improved [66]. The condition $n > 2f$, required by the OTC, is optimal as well [16].

4.2.3 Function *choose*

In the algorithm shown in Figure 4.2, acceptors choose a proposal for OTC_i based on two pieces of information: (i) the proposal x_i issued by the coordinator c_i and (ii) the collection of states $\langle S_j \rangle_{j<i}$, where S_j is a semi-complete learner state in OTC_j. This decision is made using function *choose*, which takes two parameters: the collection of semi-complete states $\langle S_j \rangle_{j<i}$, and the coordinator's proposal x_i. It returns a value x with two properties: (i) an honest acceptor received x as a proposal from some coordinator, and (ii) no decision other than x has been made in any round $j < i$. If some round $j < i$ might have decided, then *choose* returns the value of this decision; otherwise it returns x_i.

Two rounds

For starters, consider the case $i = 2$. In this case, the collection $\langle S_j \rangle_{j<i}$ consists of one semi-complete state S_1. In other words, (i) $possible_{S_1}(x)$ holds for at most one x, and (ii) $possible_{S_1}(x) \implies valid_{S_1}(x)$ for all x. The first property gives us two cases:

1. Predicate $possible_{S_1}(x)$ does not hold for any x. This means that no decision was made in OTC_1, and function *choose* can return x_2.

2. Predicate $possible_{S_1}(x)$ holds for exactly one x. Property (ii) implies $valid_{S_1}(x)$, therefore x has been proposed by an honest acceptor. This acceptor must have received x from c_1, so function *choose* can return x.

More rounds

A similar method can be applied by the acceptors in the later rounds, however, this generalization is not trivial. Consider the sixth round as an example. Assume that predicates $possible_{S_j}(x)$ and $valid_{S_j}(x)$ with semi-complete states S_j hold for the following values of x:

```
1  function choose(⟨S_j⟩_{j<i}, x_i)
2    if ¬possible_{S_j}(x) for all x and all j < i then
3      return x_i
4    else
5      let j < i be the largest round number for which possible_{S_j}(x) holds for some x
6      return the x for which possible_{S_j}(x) holds
```

Figure 4.3: Function *choose*.

	OTC_1	OTC_2	OTC_3	OTC_4	OTC_5
$possible_{S_j}(x)$	$\emptyset$	$\{2\}$	$\emptyset$	$\{4\}$	$\emptyset$
$valid_{S_j}(x)$	$\emptyset$	$\{2\}$	$\{3\}$	$\{3,4\}$	$\{4\}$

No decisions have been made in OTC_1, OTC_3, and OTC_5. However, some learners in OTC_2 might have decided on 2, and some in OTC_2 might have decided on 4. In order to avoid conflicts with previous decisions, function *choose* should return a value equal to 2 and 4 at the same time. This is not possible.

Fortunately, we can prove that no learners decided on 2 in OTC_2. If they had, then all honest acceptors in round 4 would have been forced to propose 2. However, since $valid_{S_4}(4)$ holds, we know that some honest acceptor proposed 4. Therefore, 2 could not have been a decision in OTC_2 and the coordinator of round 6 can safely propose 4. Note that the truth of $valid_{S_4}(4)$ is forced by the truth of $possible_{S_4}(4)$ and semi-completeness of S_4.

There are several reasons why one might doubt that values of predicates $possible(x)$ and $valid(x)$ in the above table can occur in an actual run. First, $possible_{S_2}(2)$ holds although we have just established that round 2 could not have made any decision. Note, however that we concluded this from the information from S_2 and S_4; the state of S_2 alone does not provide enough information to exclude 2 as a possible decision. Second, some acceptor in round 3, who did not have access to OTC_4, proposed 3, which differs from the possible decision 2 from OTC_2. The answer is that predicates *possible* and *valid* can change from learner to learner. Since OTC_2 made no decision, it is possible that at that particular learner, $possible_{S_2}(x)$ and $valid_{S_2}(x)$ held only for $x = 3$.

General solution

Figure 4.3 shows an implementation of *choose*, which uses a simple generalization of the observation from the previous section. One of two cases holds:

1. Predicate $possible_{S_j}(x)$ is false for all x and all $j < i$. This means that no decision could have been made by any previous OTC_j, so x_i is returned.

```
1   when coordinator c_i executes propose(x_i) do
2     for all j < i do
3       wait until the state S_j of c_i as a secure learner in OTC_j is semi-complete
4     broadcast x_i and ⟨S_j⟩_{j<i} to all acceptors
5   when an acceptor receives x_i and ⟨S_j⟩_{j<i} from c_i for the first time do
6     if all signatures in ⟨S_j⟩_{j<i} are correct and all states S_j are semi-complete then
7       x ← choose(⟨S_j⟩_{j<i}, x_i)                          { select the proposal }
8       OTC_i.propose(x)
9   when for each j < i, an acceptor received
10          "stop round j" from more than m + f acceptors do
11    start round j timer
12  when acceptor has not decided in OTC_i and the round i timeout expired, or
13          received message "stop round i" from more than m acceptors do
14    OTC_i.stop
15    broadcast "stop round i"
16  when a learner has OTC_i.decision(x) or
17          received "decide on x" from more than m acceptors do
18    decide(x)
19    if the learner is also an acceptor then
20      broadcast "decide on x"
21    wait until received "decide on x" from more than m + f acceptors
22    halt
```

Figure 4.4: Coordinated Consensus algorithm for the Byzantine model.

2. There is the largest $j < i$ for which $possible_{S_j}(x)$ is true for some x. This unique x is returned.

Lemma B.1.2 proves that this definition of *choose* satisfies the required properties.

4.3 Coordinated Consensus in malicious settings

In this section, we will modify the Consensus algorithm from Figure 4.2 to make it resistant to malicious processes. In order to achieve this, we have to solve two groups of problems. Firstly, malicious coordinators can jeopardize the safety of the algorithm. They can broadcast false states $\langle S_j \rangle_{j<i}$ to the acceptors. Not only that, they can send different states $\langle S_j \rangle_{j<i}$ and proposals x_i to different acceptors. The second group of problems deals with malicious acceptors. The safety of the algorithm is not threatened in this case; the previous chapters showed how to construct OTC instances resistant to malicious acceptors. However, the part of code from Figure 4.2 responsible for stopping rounds and

halting must be slightly changed, otherwise the Termination property might not hold. The following two sections deal with these issues.

4.3.1 Malicious coordinators

The most important problem with malicious coordinators is that they can broadcast false collections of states $\langle S_j \rangle_{j<i}$ to the acceptors. This might lead to acceptors proposing values different from previous decisions, thereby leading to violation of Agreement.

For example, consider a scenario in which the first round coordinator proposes 1, and only one learner reaches a decision. Assume that no acceptor decides in round one. The first round is stopped and then the coordinator of the second round broadcasts its proposal 2 along with a falsified state S_1 that indicates that no decision was taken in the first round. As a result, all acceptors will propose 2, which will eventually become a decision. This violates Agreement because a learner in the first round decided on 1.

We will show how to prevent the coordinator c_i from falsifying states $\langle S_j \rangle_{j<i}$ using digital signatures. For a moment, assume that acceptors digitally sign all messages. Normal acceptors and learners do not check these signatures; they are checked only by coordinators. In the algorithm shown in Figure 4.2, a coordinator c_i is a learner in all instances OTC_j with $j < i$. In the algorithm shown in Figure 4.4, c_i is a *secure learner* in these instances. Secure learners differ from normal ones in that they discard all messages that do not bear a valid signature.

The state of a learner consists of all messages received from acceptors. Therefore, the state of a secure learner consists of *digitally signed* messages received from acceptors. In particular, all messages in the collection $\langle S_j \rangle_{j<i}$ supplied by the coordinator c_i are digitally signed. In the algorithm from Figure 4.4, these signatures are checked by the acceptors, before choosing the value to propose to OTC_i.

Acceptors check the signatures on messages in the collection of states $\langle S_j \rangle_{j<i}$, which implies that the coordinator cannot lie about its state. More precisely, it cannot claim that it has received a message which it has not. However, it can receive a message and then claim that it has not received it, that is, deliberately lose some messages. This behaviour, although malicious in intent, is indistinguishable from the coordinator being non-maliciously faulty and the network losing the messages. Thus, this kind of malicious behaviour will be handled properly by the algorithm from Figure 4.2. Therefore, we have shown that any malicious behaviour of the coordinator will be either detected by the signature check, or it will be handled properly by the crash-stop version of the algorithm.

Note that the technique described above does not prevent a malicious coordinator c_i from sending different proposals and collections $\langle S_j \rangle_{j<i}$ to different acceptors. This might make different acceptors choose different proposals for OTC_i. Therefore, the possibility of malicious coordinators requires multi-value OTC implementations, as opposed to single-

value OTC implementations used with honest coordinators. See Section 2.3.2 for details.

Avoiding digital signatures

So far, we have assumed that acceptors sign all messages, which might require a considerable amount of computation. In their Byzantine Paxos algorithm, Castro and Liskov [15] observed that the signatures are required only by coordinators to start a new round. Therefore, computing signatures can be safely delayed until the next round is about to be started. Since no round is started without all the previous rounds being stopped, computing signatures for messages sent in round i can be delayed until this round is stopped. In particular, no digital signatures are used if all correct learners decide in the first round.

Little changes from the point of view of an acceptor; it keeps sending unsigned messages to learners, as before. When the acceptor executes $OTC_i.stop$, then it signs all messages sent in OTC_i. The number of these messages is usually quite small; OTC implementations from Chapter 2 broadcast at most three such messages.

The digital signatures in Byzantine Paxos [15] can be avoided altogether [81, 121]. The same technique can be applied here, however, it makes a difference only in rare cases when the first round does not decide. Moreover, it eliminates the computational cost of digital signatures only at the expense of one additional communication step necessary to start a new round. All in all, this is probably not worth the trouble.

4.3.2 Malicious acceptors

The malicious-resistant algorithm from Figure 4.4 handles round stopping and halting in a similar way to its non-malicious counterpart from Figure 4.2. However, there are two important differences: deciding when to stop a round and limited trust in messages from other acceptors.

Stopping a round

The benign version of the algorithm stops a round i if the failure detector suspected the coordinator c_i. This approach cannot be used in malicious setting because the failure detector abstractions from the crash-stop model are inherently not portable into Byzantine settings [34]. Therefore, the algorithm shown in Figure 4.4 uses the eventual synchrony model from Section 1.3.1 and employs timeouts to decide when to stop a round. An acceptor starts a timer for round i when more than $f + m$ acceptors report to have stopped all previous rounds $j < i$. When the timer for round i expires and the acceptor has not yet decided in OTC_i, it executes $OTC_i.stop$ and broadcasts "stop round i" to all acceptors.

As opposed to the benign versions, an acceptor cannot stop round i after receiving a single "stop round i" message, because the message might have come from a malicious

acceptor. Instead, the acceptor has to wait for more than m such messages. Similarly, a learner can decide only after receiving more than m "decide on x" messages. As a result, the **wait** instruction must wait for more than $f + m$ "decide on x" messages, not more than f as in the benign version. This ensures that more than m of them come from correct acceptors, and will eventually reach all correct learners, making them decide.

Both the benign and malicious version of the algorithm satisfy the property that either all correct learners decide or all correct acceptors stop all rounds. Eventual synchrony implies that the latter possibility does not happen, which implies the first (Termination). The details can be found in Appendix B.1; here we will only show that if the round i timer starts at all correct acceptors, then either they will all stop round i or all correct learners will decide.

If more than m correct acceptors decide in round i, then lines 16–22 ensure that all correct learners will eventually decide in that round. Similarly, if more than m correct acceptors stop round i, then lines 12–15 ensure that all correct acceptors will eventually do so. Any correct acceptor that started its round i timer will eventually either stop round i or decide. Therefore, the assertion can only be false if the number of correct acceptors is not larger than $m + m$. In other words, progress of the algorithm in Figure 4.4 requires $n - f > 2m$, that is, $n > f + 2m$. This requirement is not restrictive because any Consensus algorithm requires $n > 2f + m \geq f + 2m$ anyway [76].

Timeout considerations

The Coordinated Consensus algorithm in Figure 4.4 satisfies the same Latency property as that in Figure 4.2:

> **Latency.** If the run is timely, coordinator c_1 is correct, at most q acceptors are faulty, and OTC_1 satisfies Optimistic Termination (q, k) for some k, then all correct learners will decide in $k + 1$ communication steps.

In this case, however, we use the eventual synchrony model instead of failure detectors, so the definition of a timely run changes. Here, a run is timely if the maximum message transmission time d is "sufficiently small". What this means depends on the timeout period used for the first round. For any timeout period, we can compute $d_{\max}$ such that all $d \leq d_{\max}$ are "sufficiently small". In practice, we would like to choose the timeout large enough for all typical values of d to be smaller than the resulting $d_{\max}$, so that most runs are timely and the above Latency property ensures quick decisions. On the other hand, choosing too large a timeout results in poor performance in runs with failures.

The choice of timeout periods also influences the Termination property. As with the crash-stop version, we have to ensure that some round will eventually decide. In the eventual synchrony model, this means that some round with a correct coordinator will have enough time to decide. Since this "enough time" depends on the unknown maximum

```
1  when an acceptor a executes propose(x) as coordinators c_{i_1}, c_{i_2}, ... do
2    for i = 1, 2, ... do
3      if c_i ∈ {c_{i_1}, c_{i_2}, ...} then
4        broadcast x_i = x and ⟨S_j⟩_{j<i} to all acceptors as coordinator c_i
5      wait until the state S_i of a as a (secure) learner in OTC_i is semi-complete
```

Figure 4.5: A single acceptor implementing infinitely many coordinators.

message transmission time d, we cannot have one fixed timeout period for all rounds; instead, we increase it from round to round. For example, the timeout period t_i for round i could be a fixed multiple of the timeout t_{i-1} of the previous round, a technique similar to *exponential backoff* [15, 72].

4.3.3 Related work

The first asynchronous Byzantine Consensus algorithm was proposed by Castro and Liskov [15] and then expressed in the Paxos framework by Lampson [81]. Both algorithms assume all faulty processes being malicious ($m = f$) and require $n > 3f$, which is optimal [97]. If the run is timely and the first round coordinator is correct, these algorithms decide in three communication steps. The same can be achieved in our framework by using two-step multi-value OTCs from Section 2.4, with $q = m = f$.

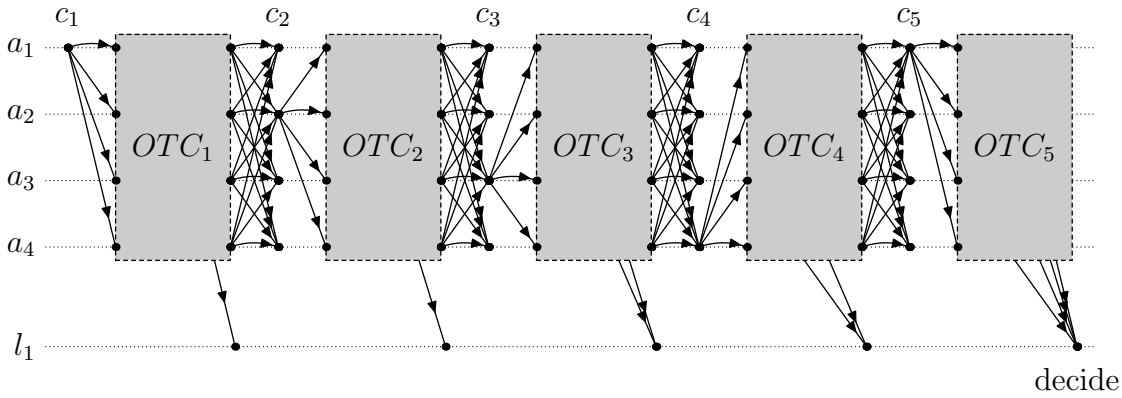
Lamport [76] showed that Byzantine Consensus requires $n > 2f + m$, and conjectured that two-step decision requires $n > f+2m+2q$. Section 2.5 showed that such an algorithm can be obtained using multi-value one-step OTC. Martin and Alvisi [87] presented an algorithm which matches this bound for the specific case $q = m = f$, requiring $n > 5f$.

My Paxos at War algorithm [121] assumes $m = f$ and requires $n > 3f + 2q$. If the run is timely and the first round coordinator is correct, it decides in two steps if at most q acceptors are faulty, and in three steps otherwise. The DGV algorithm by Dutta et al. [36] achieves the same results for any $m \leq f$. Section 2.6 showed that both of these algorithms can be reconstructed using the multi-step multi-value OTC with $q_1 = q$ and $q_2 = q_3 = f$. It also presented an OTC-based Ultimate Paxos, which generalizes DGV by allowing a two-step decision in more cases at the expense of requiring four steps in runs with many failures.

4.4 Implementing various agreement abstractions

In this section, we will show how to use the Coordinated Consensus algorithm from Section 4.3 to obtain simple implementations of various agreement abstractions.

None of these abstractions have a notion of a coordinator, therefore the coordinators will have to be played by other processes such as acceptors. Since there are only finitely




```
1  let c_1, c_2, ... = a_1, a_2, ..., a_n, a_1, a_2, ...
2  when acceptor a_i executes propose(x) do
3    explicitly start instance Coord as acceptor a_i
4    execute Coord.propose(x) as coordinators c_i, c_{i+n}, c_{i+2n}, ...
5  when Coord.decision(x) at a learner do
6    decide(x)
```

Figure 4.6: Implementing Consensus with Coordinated Consensus.

many coordinators, each acceptor will have to execute *propose*(x) as each of the infinitely many coordinators it plays. Figure 4.5 shows how this can be accomplished with finite resources. It is easy to check that this code is consistent with that in Figures 4.2 and 4.4.

4.4.1 Consensus

In the Consensus problem [16] (Section 1.2) acceptors propose values, and learners are supposed to eventually agree on one of the values proposed by the acceptors. Formally,

Validity. If all acceptors are honest and *decision*(x) holds at some learner, then some acceptor proposed x.

Agreement. There is at most one x for which *decision*(x) holds at some learner.

Termination. If all correct acceptors executed *propose*, then all correct learners will eventually decide.

The case most commonly considered in the literature assumes that the sets of acceptors and learners are the same, calling both of them simply "processes".

Consensus is similar to Coordinated Consensus, except that coordinators do not appear in the specification explicitly. This observation leads to the idea that Consensus can be implemented using Coordinated Consensus with the coordinators played by acceptors using the *rotating coordinator* paradigm: acceptor a_1 coordinates the first round,

acceptor a_2 the second, and so on. When all acceptors have been used, we start again, therefore round n is coordinated by a_n, round $n+1$ by a_1, etc. In general, round i has $c_i = a_{((i-1) \bmod n)+1}$ as the coordinator. Of course, other choices of c_i are possible, as long as the sequence $c_1, c_2, \ldots$ contains every acceptor infinitely often.

The algorithm in Figure 4.6 solves Consensus using an instance *Coord* of Coordinated Consensus. When an acceptor proposes a value x to Consensus, it first explicitly starts the instance *Coord*, and then proposes x as each coordinator it plays. Any decision made by *Coord* automatically becomes the final decision in Consensus.

Consensus implemented in such a way has the same Latency property as Coordinated Consensus. For example, consider the crash-stop model with all rounds using one-step single-value OTC from Section 2.3 with $q = f$ and $m = 0$. This OTC implementation satisfies Optimistic Termination $(f, 1)$, which results in the following Latency property:

> **Latency.** If the run is timely and acceptor a_1 is correct, then all correct learners will decide on the value proposed by a_1 in two communication steps.

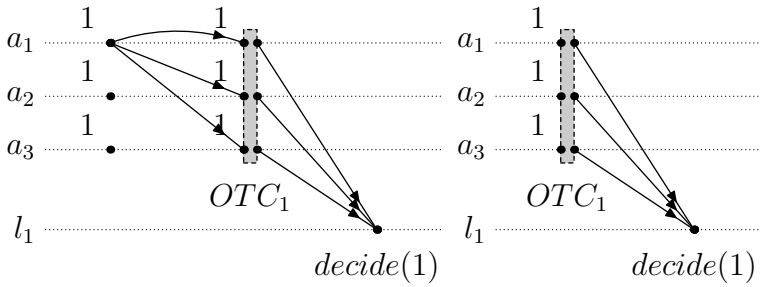
This two-step latency is optimal [66], and matches that of several known Consensus algorithms [63, 73, 112]. One-step single-value OTC implementation with $q = f$ and $m = 0$, requires $n > f + 2m + q = 2f$, which is also optimal [16].

4.4.2 One-step Consensus

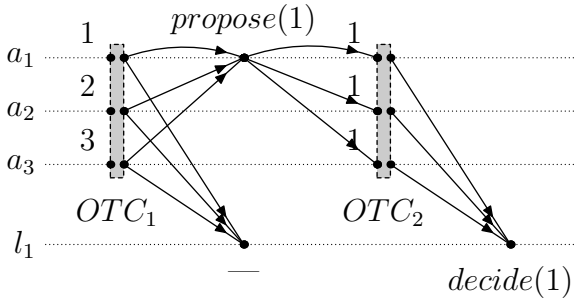
In good runs in which all acceptors propose the same value, the Consensus implementation given in Section 4.4.1 decides in two communication steps (Figure 4.7(a)). The coordinator of the first round (acceptor a_1) broadcasts its proposal (1) to all acceptors, who propose it to the first instance of OTC, which decides on 1. Observe that regardless of the implementation of the first round OTC, at least two communication steps are required: one for a_1 to broadcast its proposal and one for the acceptors to send theirs to the learners.

This two-step latency cannot be improved because Consensus requires two communication steps, even in good runs [67]. More precisely, there is no Consensus algorithm that guarantees a decision in fewer than two steps in *all* good runs. However, this does not prevent us from constructing Consensus algorithms that sometimes take only one step to decide, for example, when all acceptors proposed the same value [13, 51].

In order to implement one-step Consensus in the OTC framework, we allow the possibility of the first round not having a coordinator. Instead of waiting for the coordinator's proposal, acceptors propose directly to the OTC, as shown in Figure 4.7(b). By eliminating the first round coordinator and the corresponding proposal-broadcasting phase, we reduce the number of communication steps by one. The total latency of the Consensus algorithm in good runs is now equal to the latency of the first round OTC. For exam-



(a) With a coordinator (b) Without a coordinator



(c) Without a coordinator,

Figure 4.7: Comparison of runs with real and virtual coordinators.

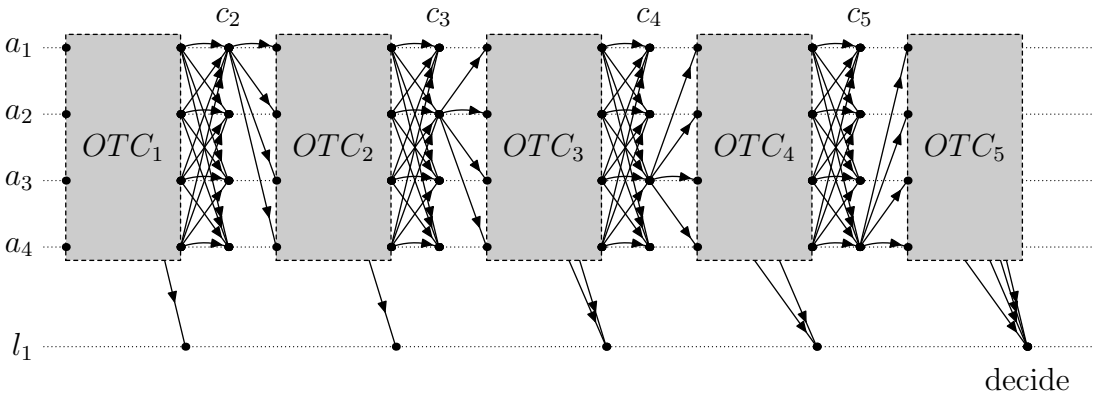
ple, one-step Consensus [13] can be achieved by using one-step multi-value OTC from Section 2.3.

Integrating the concept of the first round not having a coordinator with the rest of our framework is surprisingly easy. Instead of modifying the framework, we model a round without the coordinator as a round with a *virtual coordinator*. Acceptors' proposals are formally treated as proposals received from this virtual coordinator. The case of acceptors issuing identical proposals corresponds to the virtual coordinator being correct. Acceptors issuing different proposals corresponds to the virtual coordinator being malicious. This requires the first round to use a multi-value variant of OTC, even in the crash-stop model.

Figure 4.8 shows an implementation of one-step Consensus using an instance $Coord$ of Coordinated Consensus. The first round has a virtual coordinator, whereas further rounds are coordinated, as in Consensus, by acceptors a_1, a_2, etc. The rest of the algorithm is identical to Consensus from Figure 4.6, except that acceptors do not wait for the proposal issued by coordinator c_1. Instead, each acceptor a behaves as if it had received from c_1 the value proposed by a itself.

Virtual first-round coordinators affect the Latency property. On the one hand, having acceptors proposing their own values, as opposed to the one received from c_1, decreases the number of communication steps by one. On the other hand, the decision in the first round is now guaranteed only if all correct acceptors propose the same value:

> **Latency.** If the run is timely, all correct acceptors propose the same value x, at most q acceptors are faulty, and OTC_1 satisfies Optimistic Ter-




```
1  coordinator c_1 is virtual
2  coordinators c_2, c_3, ... = a_1, a_2, ..., a_n, a_1, a_2, ...
3  when acceptor a_i executed propose(x) do
4    explicitly start instance Coord as acceptor a_i
5    execute coord.propose(x) as coordinators c_{i+1}, c_{i+n+1}, c_{i+2n+1}, ...
6    behave as if received x_1 = x from c_1
7  when Coord.decision(x) at a learner do
8    decide(x)
```

Figure 4.8: Implementing one-step Consensus with Coordinated Consensus.

mination (q, k) for some k, then all correct learners will decide in k communication steps.

For reasons described below, virtually coordinated rounds must use one-step OTC algorithms. For example, the one-step multi-value OTC from Section 2.3 with $q = f$ and $m = 0$ satisfies Optimistic Termination $(f, 1)$, which leads to

Latency. If the run is timely and all correct acceptors propose the same value, then all correct learners will decide in one communication step.

For $q = f$ and $m = 0$, the one-step multi-value OTC requires $n > f + 2m + 2q = 3f$, which matches the requirements of the one-step Consensus algorithm by Brasileiro et al. [13]. We can obtain a new Byzantine version of this algorithm by setting $m > 0$ in the OTC implementations.

Further rounds

Consider a scenario in which acceptors propose different values. This corresponds to the virtual coordinator c_1 being malicious, and requires the acceptors to eventually stop the first round, so that (real) coordinator $c_2 = a_1$ can take over. In Figure 4.7(c), each acceptor a_i proposes its number i, so no learner decides in the first round OTC. The second round coordinator a_1 proposes 1, which all four acceptors pass to the second round OTC. All correct learners decide in the second round in three communication steps in total.

Why did all acceptors stop the first round immediately after proposing? And, for that matter, why did they stop it at all if they had no coordinator to suspect? Section 2.3.1 explained that in one-step OTC algorithms proposing a value implicitly stops the instance. Since a virtual coordinator ensures that all correct acceptors propose, no explicit *stop* is necessary in this case.

4.4.3 Individual Consensus

Consensus algorithms can decide on a value proposed by *any* acceptor. In some situations, however, we might need more control over whose proposal can become the decision. In this section, we will consider *Individual Consensus*, a variant of Consensus which can decide only on the value proposed by a particular proposer p, called the *owner*. Later in this chapter, we shall see that this abstraction is useful for providing efficient implementations of common agreement abstractions, such as Interactive Consistency [97] and Atomic Commitment [48]. Section 5.4 will present an optimal two-step Atomic Broadcast algorithm for closed groups, which also relies on an efficient implementation of Individual Consensus.

Strenghtening the Validity condition so that only the value proposed by the owner can become the decision makes it impossible to ensure the termination of the algorithm in cases when the owner is faulty. In fact, if the owner crashed before sending any messages, other processes have no way of determining its proposal. To deal with this problem we will relax the Validity condition and permit a special value ABORT to be the decision in such cases:

Sensitive Validity. If the owner is honest and $decision(x)$ holds at some learner, then x has either been proposed by the owner or equals ABORT. If the owner is correct and the run is timely, the former case must hold.

Agreement. There is at most one x for which $decision(x)$ holds at some learner.

Termination. All correct learners will eventually decide.

Sensitivity and quittability

We call the Validity property of Individual Consensus *sensitive* because it allows the learners to abort if the owner is faulty or the run is not timely. We call abstractions that require this form of Validity *sensitive*.

The notion of sensitivity is similar to the notion of *quittability* introduced by Delporte-Gallet et al. [29]. The difference is that while Sensitive Validity allows the learners to abort if the owner is faulty *or* the run is not timely, Quittable Validity allows aborting only because of the owner's failure:

Quittable Validity. If the owner is honest and *decision*(x) holds at some learner, then x has either been proposed by the owner or equals ABORT. If the owner is correct, the former case must hold.

Examples of quittable abstractions include: Interactive Consistency [97], Atomic Commitment [49, 50], and Quittable Consensus [29].

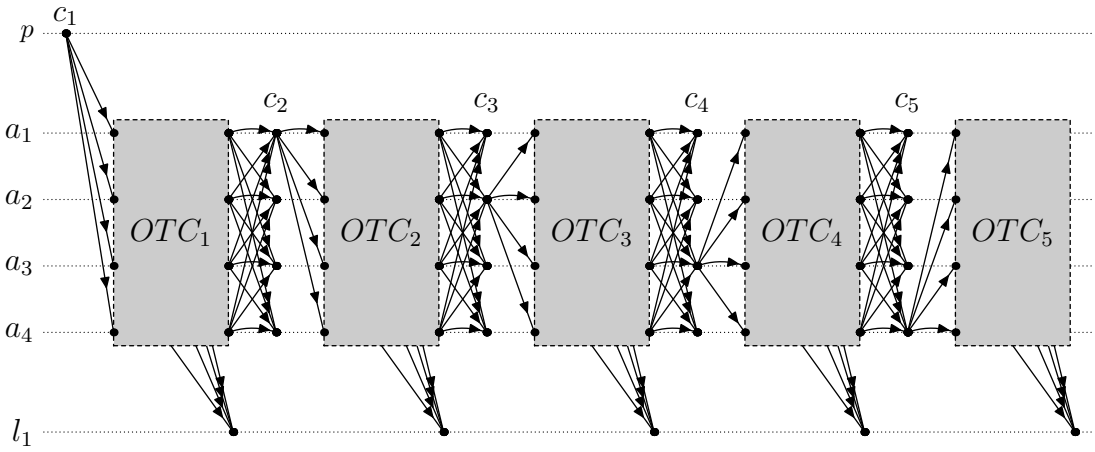
Recall that an asynchronous system extension is safe if it does not add any safety properties to the original model (Section 1.3). Examples of safe extensions include unreliable failure detectors $\Diamond S$, Ω, and eventual synchrony. Quittable abstractions cannot be implemented in such models because these models do not provide a way of distinguishing slow processes from crashed ones [46]. For this reason, quittable abstractions require other, unsafe failure detectors such as $?P$ [49] or P [16]. Since here we are interested only in safe models, we consider sensitive versions of abstractions that are traditionally quittable. That said, quittable solutions can easily be obtained from sensitive ones by replacing the $\Diamond S$ failure detector in the Coordinated Consensus algorithm shown in Figure 4.2 with the (unsafe) failure detector required by the quittable abstraction.

Implementation

Individual Consensus can be solved as a special case of Coordinated Consensus. In the algorithm presented in Figure 4.9, the owner p coordinates the first round, issuing its own proposal x. All other coordinators propose ABORT. This way, the eventually made decision will be one of the values proposed by the coordinators: x or ABORT. To guard against malicious non-first round coordinators, their actual proposals are ignored and the acceptors behave as if they had received ABORT from these coordinators. In other words, coordinators c_i with $i > 1$ are used only to broadcast the collections of states $\langle S_j \rangle_{j<i}$ (Figure 4.4).

To guarantee the second part of the Sensitive Validity condition, we have to ensure that if the owner is correct and the run is timely, then the first round will decide. This can be achieved by using an OTC_1 implementation that decides whenever all correct acceptors propose the same value, that is, one that satisfies Optimistic Termination $(f, \bullet)$.

This algorithm illustrates a general method of transforming various agreement algorithms, with a normal Validity condition, into their sensitive counterparts. Instead of letting coordinators c_i with $i > 1$ propose their own values, the acceptors just ignore their proposals and behave as if ABORT had been proposed. This technique can be used, for example, to transform the Consensus algorithm from Section 4.4.1 into Sensitive Consensus, which can abort if the run is untimely or a failure occurred. Sensitive Consensus is the sensitive counterpart of the Quittable Consensus introduced by Delporte-Gallet et al. [29].



1 $c_1 = p$
2 $c_2, c_3, \ldots = a_1, a_2, \ldots, a_n, a_1, a_2, \ldots$
3 **when** the owner p executes *propose*(x) **do**
4 execute *Coord.propose*(x) as coordinator c_1
5 **task** at acceptor a_i **is**
6 explicitly start *Coord* as acceptor a_i
7 execute *Coord.propose*(ABORT) as coordinators c_{i+1}, c_{i+n+1}, c_{i+2n+1}, ...
8 **when** an acceptor received proposal x_i from c_i with $i > 1$ **do**
9 ignore the actual x_i and behave as if $x_i =$ ABORT was received
10 **when** *Coord.decision*(x) at a learner **do**
11 *decide*(x)

Figure 4.9: Implementing Individual Consensus with Coordinated Consensus.

Latency

Individual Consensus guarantees the same Latency property as Coordinated Consensus:

> **Latency.** If the run is timely, the owner correct, at most q acceptors are faulty, and OTC_1 satisfies Optimistic Termination (q, k) for some k, then all correct learners will decide in $k + 1$ communication steps.

For example, assume the crash-stop model and all OTCs being implemented as single-value one-step OTC from Section 2.3 with $q = f$. Then, OTC_1 satisfies Optimistic Termination $(f, 1)$, which results in

> **Latency.** If the run is timely and the owner is correct, then all correct learners will decide in two communication steps.

4.4.4 Fast Individual Consensus in the crash-stop model

Even in timely runs with a correct owner, Individual Consensus takes at least two communication steps to decide. The first step is necessary for the owner to broadcast its

```
1  when the owner executes propose(x) do
2    Ind.propose(x)
3    broadcast "propose x" to all learners
4  when a learner receives "propose ABORT" from the owner do
5    decide(ABORT)
6  when Ind.decision(x) at a learner do
7    decide(x)
```

Figure 4.10: Fast Individual Consensus in the crash-stop model.

proposals to the acceptors, and the second step for the first round OTC to make a decision. Keidar and Rajsbaum [66] showed that this latency of two steps cannot generally be improved, even in the crash-stop model.

In the crash-stop model, however, we can achieve one-step decision in the special case when the owner proposes ABORT. Assume that the owner is an acceptor, so it can broadcast its proposal not only to other acceptors but also to all learners. As a result, if the owner proposes ABORT, then all the learners will know this in one communication step. The Sensitive Validity property implies that ABORT is the only possible decision. Therefore, a learner can decide on ABORT as soon as it has found out that this is the owner's proposal.

The Fast Individual Consensus algorithm is shown in Figure 4.10. It requires an honest owner and uses an instance Ind of normal Individual Consensus. When the owner proposes x, the algorithm passes it to instance Ind, and also broadcasts "propose x" to all learners. When a learner receives "propose ABORT", it immediately decides on ABORT. If the underlying instance of Individual Consensus decides on some x, this value becomes the decision as well.

Fast Individual Consensus satisfies the following property:

Latency. If the run is timely and the owner is correct, then all correct learners decide in two communication steps. If, in addition, the owner proposed ABORT, then the decision is made in one communication step.

4.4.5 Atomic Commitment

Atomic Commitment [48] is probably the most important agreement problem in distributed databases, where several replicas, here modelled as acceptors, must agree on the outcome of a distributed transaction. There are two possible outcomes: COMMIT or ABORT. Each acceptor (replica) proposes one of these values, according to the result of the part of the transaction it has executed. If all acceptors want to commit, the transaction should be committed. On the other hand, if at least one acceptor wants to abort, the

```
1  when acceptor a_i executes propose(x) do
2    broadcast "propose x"
3    explicitly start Ind as acceptor a_i
4  when Ind.decision(x) at a learner do
5    decide(x)
6  predicate the virtual owner executed propose(x) is
7    x = f(x_1, ..., x_n), where x_i is the proposal of a_i
8  predicate received proposal x from the virtual owner is
9    x = f(x_1, ..., x_n), where x_i is the proposal received from a_i
```

Figure 4.11: Computing global functions using Individual Consensus.

transaction should be aborted. The transaction can be aborted also if failures occurred, even if all acceptors wanted to commit.

As Individual Consensus, the Atomic Commitment problem comes in two variants: sensitive [45, 55] and quittable [48]. We focus on the sensitive variant, which satisfies the following validity property:

Sensitive Validity. If all acceptors are honest, then

- If the run is timely, and all acceptors are correct and proposed COMMIT, then COMMIT is the only possible decision.
- If at least one acceptor proposed ABORT, then ABORT is the only possible decision.

The quittable variant differs from the sensitive one in that the first part of its Validity property does not require a run to be timely.

Implementation

Atomic Commitment can be seen as Individual Consensus with a virtual owner simulated by all acceptors. The virtual owner is deemed to have proposed COMMIT if all acceptors proposed COMMIT, and ABORT if at least one acceptor proposed ABORT. The virtual owner is faulty/suspected iff at least one acceptor is faulty/suspected.

The Atomic Commitment algorithm in Figure 4.11 uses a single instance Ind of Individual Consensus. When an acceptor proposes a value, it broadcasts it to all acceptors and explicitly starts the instance Ind. Consider a function

$$f(x_1, \ldots, x_n) = \begin{cases} \text{COMMIT} & \text{if } x_i = \text{COMMIT for all } i, \\ \text{ABORT} & \text{if } x_i = \text{ABORT for at least one } i. \end{cases}$$

If an acceptor has received COMMIT proposals from all acceptors, it behaves as if it had received COMMIT from the virtual owner. Similarly, if it has received ABORT from at least one acceptor, it behaves as it had received ABORT from the virtual owner. Atomic Commitment decides on the value decided on by Individual Consensus.

The algorithm in Figure 4.11 solves a more general problem of *Distributed Function Computation*. Here, each acceptor a_i proposes a value x_i, and all learners must agree on the value of $f(x_1, \ldots, x_n)$ for a given function f. Formally, we require:

> **Sensitive Validity.** If all acceptors are honest and $decision(x)$ holds at some learner, then $x = f(x_1, \ldots, x_n)$ or $x =$ ABORT. If all acceptors are correct and the run is timely, the former case must hold.

Latency

The latency of Atomic Commitment is the same as that of Individual Consensus.

> **Latency.** If the run is timely, the virtual owner is correct, at most q acceptors are faulty, and OTC_1 satisfies Optimistic Termination (q, k) for some k, then all correct learners will decide in $k + 1$ communication steps.

The virtual owner is correct if all acceptors are correct ($q = 0$), which together with the timeliness assumptions implies that we are interested only in good runs. In Byzantine settings, we can use multi-value multi-step OTC from Section 2.6 with $q_1 = 0$ and $q_2 = q_3 = f$. It requires $n > 2f + m$, and satisfies Optimistic Termination $(f, \bullet)$ required by Individual Consensus. The Latency property becomes:

> **Latency.** In good runs, all correct learners decide in two communication steps.

In the crash-stop model, we use single-value one-step OTC from Section 2.3 with $q = f$ instead, which requires $n > f + q = 2f$. We also use Fast Individual Consensus from Section 4.4.4, which decides in one step if the owner proposed ABORT. Since the virtual owner proposes ABORT if at least one acceptor does so, we get:

> **Latency.** In good runs, all correct learners decide in two communication steps. If in addition, some acceptor proposed ABORT, the decision is made in one step.

Related work

Atomic Commitment is a well known problem in distributed databases. The most commonly used solution is *Two Phase Commit* (2PC) by Gray [44], in which all acceptors send their proposals to the coordinator, which computes the decision and broadcasts it

to learners. This algorithm requires two communication steps to decide in good runs, however, it does not terminate if the coordinator is faulty. The *Three Phase Commit* algorithm (3PC) by Skeen [114] corrects this problem, however, it requires three communication steps to decide, even in good runs.

Guerraoui and Schiper [53] proposed the first Atomic Broadcast algorithm that requires only two communication steps to decide in good runs. Guerraoui et al. [56] improved this result by observing that a learner can decide after receiving a message from only $n-f$ acceptors, not n as in [53]. In good runs, this algorithm behaves almost identically to ours. To guarantee Termination, it uses a variant of Consensus that treats COMMIT as the privileged value. Since only one symbol can be privileged, database replicas cannot propose versions of COMMIT that include the actual outcome of the transaction to be committed. A generalization to Distributed Function Computation is also problematic.

Gray and Lamport [45] proposed a two-step Paxos-based Atomic Commitment protocol that does not have this problem. However, their algorithm is more complicated than ours; it uses n parallel instances of Individual Consensus, each owned by a different acceptor, thereby (unnecessarily) solving a stronger problem of Interactive Consistency, discussed in Section 4.4.6.

The two communication step latency achieved by [45, 53, 56] and our algorithm cannot be improved because Atomic Commitment is a special case of Weak Consensus [66, 116]. All these algorithms require that less than half of acceptors are faulty ($n > 2f$), which is also optimal [46]. Our algorithm is the only one that tolerates malicious acceptors; in that case it requires $n > 2f + m$ and still decides in two communication steps in good runs.

Before the introduction of failure detectors by Chandra and Toueg [16], all Atomic Broadcast algorithms were specified for various timed network models such as eventual synchrony. Using the $\Diamond S$ failure detector [16] to solve (sensitive) Atomic Commitment has been shown first directly [55], and then by reduction to (Uniform) Consensus [48]. Solving the quittable variant of Atomic Commitment requires two new failure detectors: unsafe $?P$ [47, 49] and Ψ [50].

4.4.6 Interactive Consistency

Interactive Consistency [97] is an agreement abstraction in which learners agree on a vector of proposals $[v_1, \ldots, v_n]$, such that each v_i corresponds to the proposal issued by acceptor a_i. Of course, such an abstraction is impossible to implement if at least one of the acceptors fails, because in that case other processes would have no way of determining its proposal. Therefore, similarly as with Individual Consensus, we allow v_i to be ABORT if a_i is faulty or the run is not timely. Formally, we require the following Validity property:

Sensitive Validity. If a learner decides on $[v_1, \ldots, v_n]$ and acceptor a_i is

```
1  when acceptor a_i executes propose(x) do
2    explicitly start parallel instances Ind_1, Ind_2, ..., Ind_n
3    Ind_i.propose(x)
4  when Ind_i.decision(x_i) for all i = 1, ..., n at a learner do
5    decide([x_1, ..., x_n])
```

Figure 4.12: Implementing Interactive Consistency with Individual Consensus.

> honest, then v_i has either been proposed by a_i or equals ABORT. If a_i is correct and the run is timely, the former case must hold.

The first part of this property ensures that, for honest acceptors, the decision vector v contains only the acceptor proposals or ABORT values. The second parts states that for correct acceptors in timely runs, ABORT is not an option, unless the acceptor actually proposed it.

Interactive Consistency is similar to Distributed Function Computation from Section 4.4.5 with $f(x_1, \ldots, x_n) = [x_1, \ldots, x_n]$, where x_i is a_i's proposal. However, Interactive Consistency provides stronger guarantees in the sense that a failure of a single acceptor a_i affects only a single entry v_i, not the whole vector $[v_1, \ldots, v_n]$. For example, if a_1 crashes at the beginning, Interactive Consistency decides on the vector $[\text{ABORT}, x_2, x_3, \ldots, x_n]$, whereas Distributed Function Computation decides on ABORT, not giving any information about other acceptors' proposals.

Implementation

The implementation of Interactive Consistency shown in Figure 4.12 uses n parallel instances of Individual Consensus. Each instance Ind_i is owned by acceptor a_i, which uses it to propose its own proposal from Interactive Consistency. A learner decides on a vector $[x_1, \ldots, x_n]$ if each instance Ind_i decided on x_i. The correctness of this algorithm is implied directly by the correctness of each instance of Individual Consensus.

Latency

The latency of Interactive Consistency is same as that of Individual Consensus:

> **Latency.** If the run is timely, the owner is correct, at most q acceptors are faulty, and OTC_1 satisfies Optimistic Termination (q, k) for some k, then all correct learners will decide in $k + 1$ communication steps.

For Byzantine settings, using the same reasoning and OTC implementations as in Atomic Commitment, we obtain:

Latency. In good runs, all correct learners decide in two communication steps.

Similarly, in the crash-stop model, we can use instances of Fast Individual Consensus from Section 4.4.4 to achieve an even stronger Latency property. Remembering from Section 1.4 that "one communication step" is defined as the maximum message transmission time d between correct processes, we get:

Latency. In good runs, if all acceptors proposed by time $t + d$, and all acceptors with proposals other than ABORT proposed by time t, then the decision will be made by time $t + 2d$.

This property will be essential in the two-step Atomic Broadcast algorithm for closed groups, presented in Section 5.4.

Related work

In this thesis, we consider a sensitive variant of the Interactive Consistency abstraction, which allows $v_i =$ ABORT when a_i is faulty or the run is not timely. The quittable version allows $v_i =$ ABORT only in the former case, but requires the (unsafe) failure detector P [18, 29].

Interactive Consistency has first been proposed by Pease et al. [97] in the fully synchronous model. In asynchronous systems, only crash-stop implementations of Interactive Consistency have been presented in the literature so far. Delporte-Gallet et al. [27, 29] present an Interactive Consistency algorithm for that model, which decides in two communication steps, which they prove optimal.

To the best of our knowledge, our algorithm is the first to tolerate malicious processes. It achieves the same optimal two-step latency, even in malicious settings. Under the crash-stop model, it guarantees a stronger Latency property than [27]; for example, it decides in one communication step if all acceptors propose ABORT.

Gray and Lamport [45] used a similar idea to implement Atomic Commitment in crash-stop settings. In their solution, acceptors propose COMMIT or ABORT, and use n parallel instances of Paxos to agree on a common vector v of proposals. If all entries in v are COMMIT, then learners decide on COMMIT, otherwise the decision is ABORT. Section 4.4.5 discusses Atomic Commitment in more detail.

Doudou and Schiper [33] proposed Vector Consensus, an agreement abstraction similar to Interactive Consistency in Byzantine Settings. Their solution, however, uses digital signatures and requires at least four communication steps.

4.5 Other agreement frameworks

In Chapter 2, we introduced the notion of Optimistically Terminating Consensus (OTC) and presented several implementations tailored to different requirements on the number of acceptors, type of faults, and decision latency. This chapter showed how to use OTC instances to implement a variety of agreement abstractions. Such a collection of algorithms and methods is commonly known as an *agreement framework* [51]. It allows us to construct customized agreement protocols in by reusing small, well-defined modules, such as OTC instances.

Mostéfaoui and Raynal [90] proposed a generic Consensus algorithm that could use one of the two failure detectors $\Diamond S$ and S [16]. Hurfin et al. [65] generalized this method by allowing the message exchange pattern to be chosen for each round of the protocol. In other words, the designer could specify their approach to the time vs. message complexity problem: whether they preferred a low latency or a small number of messages. This trade-off does not occur in our model, however, because latency is the only performance measure we are interested in. The option of using the S failure detector does not exist in our case either because it is unsafe. Mostéfaoui et al. [92] extended the choice of options here to include the leader elector Ω and randomization, however, the protocols they presented have higher latency than ad-hoc solutions.

Boichat et al. [10] presented a modular deconstruction of Paxos into an *eventual leader elector*, similar to Ω, and a *ranked register*. By modifying the implementation of these two modules, they obtained Fast Paxos [10], Disk Paxos [42], and two variants of Paxos for the crash-recovery model. Later [11, 12], they replaced the ranked register with the *eventual register*.

Guerraoui and Raynal [51] unified the approaches presented in the last two paragraphs. A generic Consensus algorithm presented there uses a new *Lambda* abstraction, which can be implemented with different failure detectors in a modular way. Most known Consensus crash-stop protocols for asynchronous systems with failure detectors can be implemented in this framework without increasing latency.

The OTC framework has several advantages over Lambda:

- It can tolerate malicious processes, whereas all the other frameworks are limited to the crash-stop model.
- OTC algorithms are simple enough to make correctness testing and discovery fully automatic (Chapter 3).
- Lambda can implement only Consensus, with no distinction between acceptors and learners. Our framework can also implement Coordinated Consensus, leading to efficient implementation of Individual Consensus, Atomic Commitment, and Interactive Consistency.

- OTC instances are completely independent in their implementation and specification, which makes them conceptually simpler than Lambda modules. They can be replaced on a per-instance basis, so it is possible to use different implementations of OTC instances in the same run of Consensus, possibly with different sets of acceptors. For example, one might use an implementation that is fast in good runs for the first round, and a more fault-tolerant one for the others.

- The OTC abstraction is implementable in purely asynchronous settings. All external factors, such as choosing the proposals and the time for stopping, are clearly separated from the implementation of the OTC instances. As a result, they can also be modified independently from the rest of the algorithm.

- OTC instances do not have to terminate. This makes them easier to implement because, instead of worrying about Termination at every place in the algorithm, one can just trust the Coordinated Consensus algorithm to stop the current round if it does not make progress. For example, assume that an instance is supposed to decide (in one step) if all correct acceptors have proposed the same value [51]. An OTC instance can just wait for any $n - f$ identical proposals. On the other hand, if the *first* $n - f$ received proposals are not the same, a Lambda module must explicitly abort because it could jeopardize progress by waiting for more.

Recently, Guerraoui and Raynal [52] proposed *Alpha*: an abstraction similar to Lambda but with a slightly different goal. Alpha provides an agreement framework that allows one to construct a Consensus algorithm for different communication models such as message passing, shared memory, and independent disks.

Lampson [81] presented Abstract Paxos, which can be used to obtain Byzantine Paxos [15], Classic Paxos [73], and Disk Paxos [42]. Recently, Li et al. [83] showed how to deconstruct Classic Paxos and Byzantine Paxos using two new abstractions: a *register* that encapsulates quorum operations and a *token* that encapsulates a proof that the leader has read a particular value from the register.

4.6 Summary and future work

In this chapter, we have shown how to use the OTC abstraction introduced in Chapter 2 to solve a variety of agreement problems. We have formalized the Coordinated Consensus problem, and presented OTC-based algorithms for solving it in both benign and malicious settings. These algorithms served as a base for developing efficient solutions to other agreement problems: Consensus, Individual Consensus, Atomic Commitment, and Interactive Consistency, all for both failure models. By using OTC implementations from Chapter 2, we were able to provide implementations of these agreement problems. They match or improve the latency of known ad-hoc solutions.

In comparison to other agreement frameworks [10, 11, 12, 51, 65, 92], our approach makes it possible to reconstruct the highest number of known algorithms as well as to construct new ones. Firstly, this is because no other agreement framework tolerates malicious processes. Secondly, because OTC, the main unit in our solution, is relatively small; it encompasses only a single round as opposed to the whole algorithm in other frameworks. This, and the independence of OTC instances used in different rounds, gives us high flexibility with designing Consensus algorithms. The parameters of every round, such as the coordinator, the type of the OTC, or set of acceptors used, can be set independently. Some of the methods involve parallel execution of instances of OTC or Consensus. We will see more application of this technique in Atomic Broadcast protocols in Chapter 5.

In contrast to some Consensus algorithms, notably most variants of Paxos [73], the OTC framework assumes reliable channels. Reliable channels can be implemented over unreliable ones by periodic retransmission, without any latency overhead [9]. This solution, however, requires keeping unconfirmed messages in memory, therefore direct support for unreliable channels in the OTC framework would be preferable.

Chapter 5

Atomic Broadcast

In the previous chapters, we investigated agreement problems in distributed systems. Chapter 2 introduced the notion and implementations of Optimistically Terminating Consensus, the basic building block of all Consensus algorithms presented in this thesis. Chapter 3 extended this work by presenting an *automatic* way of generating OTC algorithms that satisfy given criteria. Finally, Chapter 4 showed how to use OTC to solve a variety of agreement problems, such as Consensus, Atomic Commitment, or Interactive Consistency.

All agreement abstractions presented in the previous chapters were *static* in the sense that processes issue *one* proposal and make *one* decision. Multiple instances of static agreement abstractions can be used to make multiple *independent* decisions. In this chapter, we will investigate *dynamic* agreement problems, in which proposers issue multiple proposals (message broadcasts) and make multiple mutually *dependent* decisions (message deliveries).

As an example, consider *state machine replication* [77, 113], which was used in the beginning of Chapter 1 to implement a fault-tolerant hotel booking system. It consists of clients and servers; clients broadcast requests to the servers, who execute them. To ensure consistency of the system, all the servers must receive client requests in the same order. The broadcast protocol that ensures this property is called *Atomic Broadcast* [59]. The use of Atomic Broadcast is not limited to replication, however. Other applications include [26]: clock synchronization [111], cooperative document creation, distributed shared memory, distributed locking [78], and distributed databases [2, 69, 103].

Atomic Broadcast is a truly dynamic agreement problem; clients can issue multiple proposals (broadcast many messages) and the servers must make multiple decisions (deliver these messages). Most importantly, these decisions are mutually dependent; the order in which messages are delivered must be the same at all servers.

In this chapter, we will investigate implementations of Atomic Broadcast in distributed systems. We will present efficient implementations of several variants of the problem as well as lower bounds proving optimality of our solutions. As in the previous chapters, our

goal will be to minimize the latency in good runs. This time, however, we consider only the crash-stop model, with no malicious processes. For this reason, we use the symbol "m" to denote messages, not the number of malicious acceptors (which is zero).

This chapter is structured in the following way. Section 5.1 formalizes the notion of Atomic Broadcast and presents a Consensus-based algorithm by Chandra and Toueg [16]. Based on its design, we construct an algorithm especially designed for low latency in typical runs. Section 5.2 provides a latency-optimal implementation of Generic Broadcast, a generalization of Atomic Broadcast in which only *conflicting* messages must be delivered in the same order. This algorithm uses infinitely many instances of Consensus running in parallel; a method of implementing this with finite resources is presented in Section 5.3. Section 5.4 considers a restricted, *closed-group* variant of Atomic Broadcast, in which only acceptors (servers) are allowed to atomically broadcast messages. We present an algorithm whose latency is better than what is possible without this restriction. All the algorithms presented in this chapter are optimal in terms of latency and the required number of acceptors. Some of the proofs come from the literature, the others are presented in Section 5.5. Section 5.6 concludes this chapter.

5.1 Atomic Broadcast

Atomic Broadcast is specified in terms of three classes of processes: proposers, acceptors, and learners. Proposers broadcast messages to acceptors, who cooperate in order to enable learners to deliver them in the same order. The abstraction is defined in terms of two primitives: $abcast(m)$ and $adeliver(m)$, available at proposers and learners, respectively. A proposer executes $abcast(m)$ whenever it wants to atomically broadcast ("abcast") message m. A learner atomically delivers ("adelivers") a message m by executing $adeliver(m)$.

Formally, we require the following properties:

Validity. For any message m, every learner delivers m at most once, and only if m was abcast by a proposer.

Agreement. For any two different messages m and m', it is impossible that one learner delivers m without having previously delivered m', and another learner delivers m' without having previously delivered m.

Termination Validity. If a correct proposer abcasts a message, then all correct learners will eventually deliver it.

Termination Agreement. If a learner delivers a message, then all correct learners will eventually deliver it.

The above properties have been classified according to two orthogonal characteristics: safety-liveness and validity-agreement. All Termination properties are liveness properties,

whereas all the others are safety properties. A property is a validity property if it relates input to output (broadcast to delivery), and an agreement property if it relates output to output (delivery to delivery). This convention makes the property names consistent with those used for other agreement abstractions, notably Consensus. Property names traditionally used in the Atomic Broadcast literature conflict with those used for Consensus, which might be confusing. The correspondence table is:

this thesis	traditional
Validity	Integrity
Agreement	Total Order
Termination Validity	Validity
Termination Agreement	Agreement

Several versions of the Agreement (Total Order) property have been proposed in the literature [26]. For example, the Prefix Order property considers the sequences of messages delivered by any two learners and requires that one is a prefix of the other. The Agreement property used here is equivalent to Prefix Order, but as opposed to the latter it can be easily generalized for use in Generic Broadcast. Aguilera et al. [5] proposed Gap Free Total Order, which requires that if some learner delivers message m' after message m, then every learner delivers m' only after it has delivered m. The problem with this definition is that it allows the algorithm to deliver disjoint sets of messages at different learners, in any order, and still be considered safe.

The Atomic Broadcast properties, as defined above, specify the *uniform* variant of the abstraction [26], which offers guarantees to all participating processes, not only to correct ones. For example, a non-uniform Atomic Broadcast algorithm allows faulty learners to deliver messages in a different order than the correct ones. As in the rest of this thesis, we assume that the considered abstractions are uniform, unless explicitly stated otherwise.

System model summary

We assume the same system model as in the previous chapters: a network of processes communicating via asynchronous reliable channels. This means that channels do not create or modify messages, and all messages between correct processes eventually reach their destination.

Our system consists of three, possibly overlapping, groups of processes: proposers, acceptors, and learners (see Section 1.1). There are exactly n acceptors, out of which at most f are non-maliciously faulty (i.e., we assume no malicious faults in this chapter). There are no restrictions on the number of proposers or learners, or on the number of failures in these groups. For common replicated systems, servers are modelled as acceptors, and clients as proposers and learners (every server is also a client).

The Consensus and Atomic Broadcast problems are equivalent in the sense that the solvability of either of these problems implies the solvability of the other [16]. In particular, Atomic Broadcast is not solvable in purely asynchronous systems. Various model extensions can be used to remedy the situation, such as failure detectors or eventual synchrony. In this chapter, however, we simply assume that the model is strong enough to make Consensus solvable, and use it as a subroutine in our Atomic Broadcast algorithms.

Performance measures

OTC-based Consensus algorithms usually decide on the value proposed by the first round coordinator c_1. For this reason, we call process c_1 the *leader* of the algorithm. More generally, in any Consensus algorithm, the *leader* is a process with the following property: if the run is timely and the leader is correct, then learners decide on the value proposed by the leader.

To tolerate malicious processes, the previous chapters assumed that the leader c_1 is fixed and known in advance to all acceptors. In the crash-stop model, considered here, some Consensus algorithms [35, 73] allow the leader to be elected dynamically, for example, using the Ω failure detector [18]. To accommodate this possibility, this chapter introduces the notion of a stable run. We say that a run is *stable* if it is timely, all correct acceptors perceive the same leader, which is correct and never change. We also strengthen the definition of a *good run* by requiring it to be stable as well. Note that the assumption of c_1 being fixed and known in advance made by previous chapters automatically makes all good runs stable, so this definition of a good run is consistent with that from Section 1.3.4.

We evaluate Atomic Broadcast algorithms by considering their *latency* in good and stable runs. In other words, we measure the time that passes between a message being abcast by its sender and it being adelivered by all correct learners. Note that some papers [39, 89] ignore the first step, in which the proposer broadcasts the message to the acceptors; in that case one communication step must be added to the latency reported in those papers.

5.1.1 Related work

Atomic Broadcast can easily be used to solve Consensus by having every acceptor abcast its proposal and the learners decide on the first value that is delivered. This reduction, shown in Figure 5.1, requires no additional message delay. As a result, all lower bounds for Consensus apply to Atomic Broadcast as well. In particular, Atomic Broadcast cannot be solved in purely asynchronous settings [40], and requires at least two communication steps, even in good runs [19, 66, 74].

The previous paragraph showed that Atomic Broadcast can implement Consensus. The reduction in the other direction is also possible; Chandra and Toueg [16] showed an

```
1  when an acceptor executes propose(x) do
2    abcast(x)
3  task deciding at learners is
4    wait for adeliver(y) for some y
5    decide(y)
```

Figure 5.1: Reducing Consensus to Atomic Broadcast by atomically broadcasting proposals and adopting the first delivered message as the decision.

Algorithm	same order	different orders
Chandra-Toueg [16]	3	3
Optimistic Atomic Broadcast [99]	2	4
Our algorithm	2	3

Figure 5.2: Comparison of the number of communication steps required by several Atomic Broadcast algorithms in stable runs. The numbers are reported for two cases: when all acceptors receive proposer messages in the same order and in different orders.

algorithm that solves Atomic Broadcast using a sequence of Consensus instances. The details of this algorithm will be discussed in Section 5.1.2. As opposed to the reduction shown in Figure 5.1, the Chandra-Toueg algorithm requires one additional communication step. In other words, the algorithm has a latency of three communication steps, assuming that the underlying Consensus algorithm has a two-step latency.

The three-step latency of the Chandra-Toueg algorithm cannot be improved (Theorem 5.5.3). Informally, this is because one communication step is needed for the proposer to broadcast its message to the acceptors, and two more for the acceptors to agree on the order of messages [19, 66, 74]. This means that no Atomic Broadcast algorithm can guarantee a latency lower than three communication steps in *all* good runs. It is possible, however, to develop algorithms that deliver all messages in two communication steps in some of them.

In many networks, especially LANs, it is common for messages sent by proposers to be received by all correct acceptors in the same order. Pedone and Schiper [99, 101] presented an Atomic Broadcast algorithm that delivers messages in two steps in stable runs with this property. Later in this section, we will present an Atomic Broadcast protocol that achieves the same latencies. The advantage of our algorithm becomes apparent in stable runs in which acceptors receive proposer messages in different orders. In those runs, Optimistic Atomic Broadcast [99] requires at least four communication steps. On the other hand, our algorithm requires only three communication steps, which Theorem 5.5.3 shows to be the lower bound. Figure 5.2 shows that, in terms of latency, our algorithm is strictly better than both the Chandra-Toueg and Optimistic Atomic Broadcast algorithms.

The two-step latency Atomic Broadcast algorithms reported in [39, 89] results from

using a different definition of latency, which ignores the first step, in which the proposer broadcasts its message to acceptors. In these cases, one communication step must be added to the reported delivery latency. Taking this into account, the algorithm by Mostéfaoui and Raynal [89] requires at least three communication steps – the same as Chandra-Toueg [16]. The algorithm by Ezhilchelvan et al. [39] can decide in two communication steps, however, it assumes that a new abstraction, called *Notifying Broadcast*, can be implemented in one communication step. Although this may be true in some networks such as Ethernet, implementing Notifying Broadcast in the standard message passing model requires two communication steps. As a result, [39] requires at least three communication steps in our model.

Other algorithms

The number of different implementations of Atomic Broadcast described in the literature is tremendous. A survey by Défago et al. [26] divides them into five groups: with fixed sequencer, with moving sequencer, privilege-based, communication-history-based, and with destination agreement. Our model corresponds to "destination agreement" protocols, in which proposers send their messages to acceptors, who cooperate in choosing the delivery order.

The algorithms listed by Défago et al. [26] use various extensions of the purely asynchronous model: randomization [7], failure detectors [16], eventual synchrony [37], and group membership services [21]. Some of these algorithms support features that we do not consider here, such as multiple destination groups, acceptors that can join and leave the system dynamically, recovery of crashed processes, or malicious processes. Finally, several algorithms implement variants of Atomic Broadcast with weaker safety guarantees: non-uniformity, unsafe optimistic delivery, or safety only in timely runs.

5.1.2 Chandra-Toueg algorithm

The original Atomic Broadcast algorithm proposed by Chandra and Toueg [16] assumes that all learners are acceptors. In brief, proposers broadcasts their messages to acceptors, who use instances of Consensus to deliver these messages in the same order.

When a proposer wants to abcast a message m, it broadcasts m to the acceptors (Figure 5.3). Each acceptor maintains the set $\mathcal{M}$ of received but undelivered messages, initially empty. Acceptors add each new received message to $\mathcal{M}$. To broadcast messages, proposers use Reliable Broadcast [59], in which acceptors rebroadcast received messages to other acceptors. This ensures that if one correct acceptor receives a message, then all correct acceptors eventually will.

The main task of the algorithm is an infinite loop (lines 6–12). In each iteration, the acceptor waits until $\mathcal{M}$ is not empty, that is, until there are some undelivered messages.

```
1   M is the set of received undelivered messages, initially ∅
2   when a proposer executes abcast(m) do
3     reliably broadcast m to the acceptors
4   when an acceptor reliably receives m do
5     insert m to M
6   task delivery at acceptors/learners is
7     for k = 1, 2, ... do
8       wait until M ≠ ∅
9       batch_k.propose(M)
10      wait until batch_k.decision(B_k)
11      adeliver all undelivered messages in B_k in some deterministic order
12      remove from M all messages in B_k
```

Figure 5.3: Atomic Broadcast algorithm proposed by Chandra and Toueg [16]

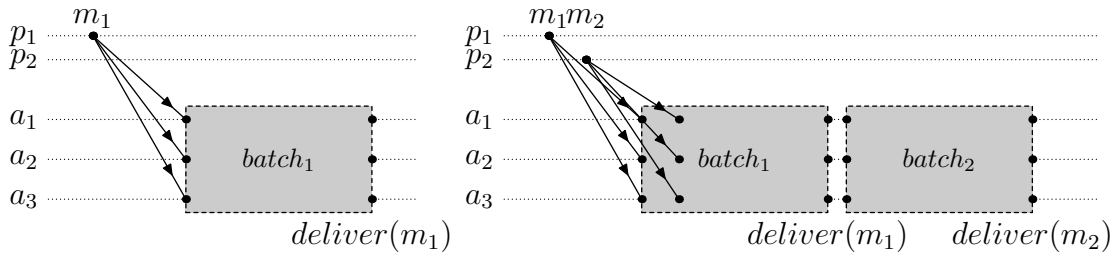
The acceptor tries to deliver these messages by proposing $\mathcal{M}$ to the Consensus instance $batch_k$. At the same time, other correct acceptors also propose their sets $\mathcal{M}$. The Termination property of Consensus implies that all correct learners (acceptors) will eventually decide. When an acceptor decides on some batch of messages B_k, it delivers them in some deterministic order, and removes them from $\mathcal{M}$. Messages from B_k that have already been delivered are not delivered again. The Agreement property of Consensus implies that all acceptors end up with the same batches B_k, so they deliver the same messages in the same order (Agreement). For Validity, observe that any delivered message m must have been proposed to one of the instances $batch_k$. As a result, m must have been abcast by some proposer.

For Termination Validity, observe that any message m abcast by a correct proposer will eventually either be delivered or belong to $\mathcal{M}$ at all correct acceptors. In the latter case, all correct acceptors will propose $\mathcal{M} \ni m$ to some instance $batch_k$, which will deliver m. The original Chandra-Toueg algorithm is not uniform; Termination Agreement is guaranteed only for correct learners (acceptors).

Latency

Figure 5.4(a) shows a good run in which only one message, m_1, is abcast. In one communication step, m_1 reaches all acceptors, who add it to their respective sets $\mathcal{M}$, and propose $\mathcal{M} = \{m_1\}$ to $batch_1$. Assuming a two-step implementation of Consensus [73, 112], instance $batch_1$ will decide on $\{m_1\}$ and deliver m_1 two communication steps later. This gives the total latency of three steps.

Now, consider the run shown in Figure 5.4(b), in which message m_2 is abcast just after m_1. It arrives at the acceptors just after they proposed $\mathcal{M} = \{m_1\}$ to $batch_1$. When $batch_1$ decides and m_1 is delivered, all acceptors propose $\mathcal{M} = \{m_2\}$ to $batch_2$.



(a) A run with one message m

(b) A run with two messages m and m'

Figure 5.4: Two runs of Chandra-Toueg Atomic Broadcast. The left one exhibits a three-step delivery latency, however, the right one shows that almost five steps are necessary in some good runs.

Instance $batch_2$ decides two communication steps later, leading to a total latency for m_2 of almost five communication steps. Section 5.1.3 will show how we can modify the original Chandra-Toueg algorithm to achieve a latency of three communication steps in all stable runs.

5.1.3 Modified Chandra-Toueg algorithm

In this section, we will present a modified version of the Chandra-Toueg algorithm, which guarantees a three-step delivery latency in all stable runs. In brief, we avoid five-step latency in runs like the one in Figure 5.4(b) by starting $batch_2$ immediately after m_2 arrives, without waiting for $batch_1$ to finish. In the original algorithm, $batch_2$ has to wait until $batch_1$ finishes, because it removes all messages delivered in $batch_1$ from the proposals to $batch_2$. In the modified version, we cannot remove these messages from $\mathcal{M}$ because $batch_2$ might start before $batch_1$ finishes. Instead, learners take special care not to deliver messages twice. Not removing delivered messages from $\mathcal{M}$ allows us also to separate the roles of acceptors and learners in the new algorithm.

The algorithm is shown in Figure 5.5. The sending mechanism is similar to that in the original version: in order to abcast a message m, a proposer broadcasts m to all acceptors (lines 2–3). Note that we use ordinary broadcast, not reliable broadcast. To cope with faulty proposers, we will use a different mechanism (lines 4–5), which also ensures the uniformity of our solution. Section 5.1.4 will explain the details.

Each acceptor maintains a set $\mathcal{M}$ of received messages, and runs an infinite loop with $k = 1, 2$, etc. The k-th iteration of this loop corresponds to the k-th received message. In that iteration, the acceptor first waits until the k-th message m has been received, and adds it to the set $\mathcal{M}$ of received messages. At this point, $\mathcal{M}$ contains the first k messages received by the acceptor. Then, the acceptor proposes the current set $\mathcal{M}$ to the k-th instance of Consensus ($batch_k$). Compared to the original algorithm, delivered messages are not removed from $\mathcal{M}$. This is because, in our model, learners (not acceptors) deliver messages, so acceptors have no notion of "delivered" messages.

```
1  M is the set of received messages, initially ∅
2  when a proposer executes abcast(m) do
3    broadcast m to the acceptors
4  when an acceptor sees m eventually do
5    abcast(m)
6  task broadcasting at acceptors is
7    for k = 1, 2, ... do
8      wait for some message m ∉ M
9      insert m into M
10     batch_k.propose(M)
11 task delivery at learners is
12   for k = 1, 2, ... do
13     wait until batch_k.decision(B_k)
14     deliver all undelivered messages from B_k in some deterministic order
```

Figure 5.5: A modified version of the Chandra-Toueg Atomic Broadcast algorithm, which guarantees three-step delivery in all stable runs.

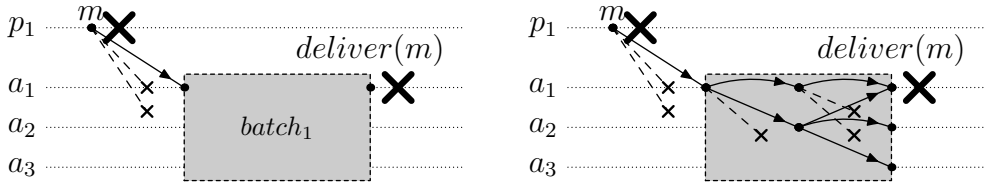


Figure 5.6: Two diagrams of a run that violates Termination Agreement.

The delivery task run at learners is similar to the broadcasting task at acceptors. Each learner runs an infinite loop with $k = 1, 2$, etc. In each iteration k, the leader waits for the Consensus instance $batch_k$ to decide. Recall that, in this instance, each correct acceptor proposed the set of the first k messages it received. Therefore, the decision B_k contains exactly k messages. The learner delivers all messages in B_k that have not been delivered yet, in some deterministic order.

This algorithm meets the Validity, Agreement, and Termination Validity properties for the same reasons as the original Chandra-Toueg algorithm. In the next section, we will show how to satisfy Termination Agreement.

5.1.4 Termination Agreement

As explained so far, the algorithm does not satisfy Termination Agreement. For example, it is possible that only faulty learners will deliver a given message m. Consider the scenario from Figure 5.6, in which all learners are acceptors. There is only one message m sent in the system, by a faulty proposer. This message is received only by one, faulty acceptor a_1, who proposes $\{m\}$ to the Consensus instance $batch_1$. Other acceptors have not received

m, so they do not propose anything. The specification of Consensus does not include the Termination Agreement property, so acceptor a_1 can be the only one to decide. As a result, message m is delivered by a_1, and not delivered by correct learners a_2 and a_3, which violates Termination Agreement.

Observe that having acceptors rebroadcast all received messages (Reliable Broadcast) does not improve the situation. In our scenario, the only acceptor to receive m is a_1, so even if it rebroadcasts m_1, other acceptors might not receive it because a_1 is faulty. Reliable Broadcast guarantees that all correct acceptors will eventually receive a message if a *correct* acceptor received it. This correctness restriction is removed in Uniform Reliable Broadcast [60]. This abstraction, however, requires two communication steps, which would slow our algorithm down by one step. We need another way of ensuring Termination Agreement.

The essence of the problem is that some/all of the messages sent by a *faulty* proposer to correct acceptors might get lost. Our approach is to ensure that every delivered message was abcast by a *correct* proposer, in which case Termination Validity will imply Termination Agreement. Of course, we cannot make a faulty proposer correct nor can we magically distinguish a correct proposer from a faulty one. What we can do is to make other proposers abcast the same message and guarantee that at least one of them is correct. In particular, since all acceptors are proposers, we can make all of them re-abcast all messages that they "see".

We say that an acceptor *sees* a message m if, in any Consensus instance, it received any proposal containing m (not necessarily directly). For example, in Figure 5.6, all acceptors see message m: acceptor a_1 received m directly from the proposer, a_2 received m from a_1, and a_3 received m from a_2.

Lemma C.1.1 shows that, in *any* Consensus algorithm, if some learner decides on x, then at least one *correct* acceptor has seen x. For our Atomic Broadcast algorithm, this implies that if any learner delivers m, then a correct acceptor has seen m. Since correct acceptors abcast all messages they see (lines 4–5), Termination Validity guarantees that m will be delivered by all correct learners (Termination Agreement).

The action specifier **eventually do** in line 4 indicates that re-abcasting seen messages does not have to be performed immediately. Indeed, this action is necessary only to deal with faulty proposers, to which Latency requirements do not apply. Delayed re-abcasting reduces network traffic, because if all acceptors report having received a message m, then m does not have to be re-abcast.

To sum up, we have shown that if a learner delivers a message, then it will be abcast by at least one correct acceptor, which – by Termination Validity – implies that all correct learners will eventually deliver it (Termination Agreement). This technique allows us to automatically guarantee Termination Agreement in any Consensus-based algorithm that satisfies Termination Validity.

5.1.5 Delivery in three steps

In this section, we will investigate the latency of the algorithm from Figure 5.5 in stable runs. The algorithm consists of two phases: proposers sending messages to the acceptors, and the acceptors executing Consensus to let the learners deliver messages. The broadcast phase takes one communication step. Consensus requires at least two communication steps [66], and this can be implemented using a variety of known algorithms [35, 63, 73, 112] or using the OTC-based Consensus algorithms from Chapter 4. All these algorithms have the following property:

C2: In stable runs, all correct learners decide on the value proposed by the leader two communication steps after the leader proposed.

For our Atomic Broadcast algorithm, Property **C2** implies that a message m is delivered two communication steps after the leader proposed $\mathcal{M}$ containing m, that is, three communication steps after m was abcast.

Latency. In stable runs, a message abcast by a correct proposer is delivered by all correct learners in three communication steps.

5.1.6 Delivery in two steps

The modified Chandra-Toueg algorithm presented in the previous section achieves the latency of three communication steps in stable runs, provided that the underlying Consensus algorithm satisfies Property **C2**. Theorem 5.5.3 shows that, even in good runs, this latency cannot be improved. We can, however, attain a two-step latency in *ordered runs* – runs in which all correct acceptors receive proposers' messages in the same order.

Assume that the underlying Consensus algorithm has the following property:

C1: In stable runs, if all correct acceptors proposed the same value, then all correct learners decide on that value in one communication step.

In ordered runs, all correct acceptors receive messages from the proposers in the same order, so in each instance $batch_k$ all acceptors propose the same set $\mathcal{M}$. Therefore, if the leader is correct, Property **C1** implies that $batch_k$ decides one communication step after all acceptors proposed, that is, one communication step after all acceptors received the k-th message. This implies

Latency. In ordered stable runs, a message abcast by a correct proposer is delivered by all correct learners in two communication steps.

Related work

A Consensus algorithm satisfying Property **C1** has been proposed by Brasileiro et al. [13]. Alternatively, one can use an OTC-based Consensus with the first round consisting of a virtual coordinator and the one-step multi-value OTC from Section 2.3. Both algorithms satisfy Property **C1** and require $n > 3f$.

The above algorithms do not satisfy property **C2**. In fact, if acceptors issue different proposals, they can take up to three communication steps to decide. Thus, using these Consensus algorithms in the Atomic Broadcast algorithm from Figure 5.5 results in a four-step delivery latency in stable runs that are not ordered. Note that the Optimistic Atomic Broadcast protocol [99] exhibits the same latencies in stable runs: two communication steps in ordered ones, and four in others.

5.1.7 Delivery in two steps and three steps

The latency of the Chandra-Toueg algorithm in stable runs depends on the properties of the underlying Consensus algorithm. Section 5.1.5 showed that if Property **C2** holds, then all messages are delivered within three communication steps. Section 5.1.6 showed that if Property **C1** holds and the run is ordered, then messages are delivered within two communication steps. Section 5.1.8 will present a Consensus algorithm that satisfies both of these properties at the same time. Therefore, applying this algorithm to the Atomic Broadcast algorithm from Figure 5.5 will result in

> **Latency.** In stable runs, a message abcast by a correct proposer is delivered by all correct learners in two steps if the run is ordered, and three steps otherwise.

Related work

Figure 5.2 showed that, in terms of latency, the above Atomic Broadcast algorithm takes the best of both Chandra-Toueg [16] and Optimistic Atomic Broadcast [99]. It guarantees that, in stable runs, messages will be delivered in two steps if the run is ordered [99], and in three otherwise [16].

Our Atomic Broadcast algorithm satisfies several lower bounds. Section 5.5.1 shows that a latency of less than two communication steps is impossible in any run. The three-step latency in good runs is also optimal (Theorem 5.5.3). Finally, the requirement $n > 3f$ is necessary for any Atomic Broadcast algorithm to deliver messages in two steps in good runs [100].

The Consensus algorithm from Section 5.1.8 satisfies Properties **C1** and **C2**. A Consensus algorithm satisfying these properties can also be obtained in the OTC framework directly by using a simple generalization of the crash-stop OTC algorithm $\mathcal{T}_4$ that was

```
1   when acceptor a executes propose(x) do
2     broadcast(x, a)
3     propose_1(x)
4     propose_2(x, a)
5     propose_L(l) where l is the output of Ω
6   task decide at learners is
7     wait until decision_L(l)
8     wait until one of the conditions is true and decide on x
9       condition 1: decision_1(x) and receive(x, l)
10      condition 2: decision_2(x, l)
11      condition 3: decision_1(x) and decision_2(y, q) with q ≠ l
```

Figure 5.7: The One-Two Consensus algorithm.

automatically discovered in Section 3.5. No such algorithm satisfying both **C1** and **C2** has been previously proposed in the literature. Known algorithms satisfy only **C1** [13] or only **C2** [16, 63, 73, 112]. Guerraoui and Raynal [51] described a Consensus algorithm that satisfies both Properties **C1** and **C2**, but the former only for a single privileged value.

5.1.8 Consensus with C1 and C2

Figure 5.7 presents a Consensus algorithm that satisfies both Properties **C1** and **C2** at the same time. This algorithm uses three underlying Consensus instances: 1, 2, and L, for 1-step decision, 2-step decision, and leader election, respectively. When an acceptor a proposes x, it first broadcasts the pair (x, a), and then uses instance 1 to propose x, instance 2 to propose the pair (x, a), and instance L to propose the current output of its leader oracle Ω (lines 1–5). Instances 1 and L satisfy Property **C1**; they decide in one communication step in stable runs in which all correct acceptors propose the same value. Instance 2 satisfies Property **C2**; in stable runs, it decides two communication steps after the leader proposed. For example, we can use the algorithm by Brasileiro et al. [13] for instances 1 and L, and Paxos [73] for instance 2. This way, instances 1 and L require $n > 3f$, and instance 2 requires $n > 2f$, leading to the total requirement of $n > 3f$. See Sections 5.1.5 and 5.1.6 for other algorithms satisfying Properties **C1** and **C2**, respectively.

Lines 6–11 show the actions performed by a learner. It first waits until instance L decides on some leader l. In stable runs, in which the output of Ω is the same at all correct acceptors, this should happen within one communication step. Then, the learner waits until one of the three conditions in Figure 5.7 holds. The learner decides on x in the following situations:

1. Instance 1 decided on x, and the learner received (x, l) from the leader l decided by

instance L. In stable runs in which all correct acceptors propose the same value, this condition will hold at all correct learners in one communication step, satisfying Property **C1**.

2. Instance 2 decided on (x, l), where l is the leader decided by instance L. In stable runs, this condition will hold at all correct learners in two communication steps, satisfying Property **C2**.

3. Instance 1 decided on x and instance 2 decided on some (y, q) with q different from the decision l of instance L. Property **C2** of instance 2 ensures that this condition will never occur in stable runs. Together with the previous condition, it is used to ensure that the algorithm will terminate in any run.

Appendix C.3 proves that the algorithm in Figure 5.7 implements Consensus.

5.2 Generic Broadcast

Theorem 5.5.3 shows that no Atomic Broadcast protocol can guarantee a latency of less than three communication steps in *all* good runs. However, a smaller latency can be achieved in *some* runs that occur frequently in practice. Section 5.1 discussed Atomic Broadcast algorithms that deliver messages in two communication steps, provided that acceptors receive them in the same order. In this section, we will investigate another technique that allows us to deliver messages in two communication steps.

Pedone and Schiper [102] observed that, in most practical applications, the Agreement property, which requires *all* messages to be ordered, is too strong. As an example, they consider state machine replication [113], in which Atomic Broadcast is used by the clients to send requests to the servers. The Agreement property guarantees that all servers receive the requests in the same order, and thus perform the same sequences of operations. However, performing operations in different orders is only dangerous if these operations conflict in some sense. For example, two "read" requests or any two requests operating on unrelated objects are non-conflicting and the order of their execution does not matter.

To formalize this observation, Pedone and Schiper [102] introduced Generic Broadcast, which differs from Atomic Broadcast in that only conflicting messages must be delivered in the same order. Non-conflicting messages can be delivered by different learners in different orders. Formally, the Agreement property of Atomic Broadcast can be replaced with

> **Generic Agreement.** For any two *conflicting* messages m and m', it is impossible that one learner delivers m without having previously delivered m', and another delivers m' without having previously delivered m.

The notion of conflict is captured by a binary conflict relation on the set of all possible messages, which is a parameter of the problem [102]. This relation and therefore the (infinite) set of all possible messages are both fixed and known to all processes in the system. For example, one might consider a relation on *read* and *write* requests in which all pairs of messages conflict unless both of them are *read*s.

To distinguish Atomic Broadcast and Generic Broadcast, in the latter abstraction, proposers broadcast a message m by executing $gbcast(m)$, and learners deliver it by executing $gdeliver(m)$.

5.2.1 Genuine Generic Broadcast

The Agreement property of Atomic Broadcast is stronger than that of Generic Broadcast, so any protocol that implements the former abstraction also implements the latter. However, such implementations are not very useful because they defeat the purpose of Generic Broadcast: to deliver non-conflicting messages faster than is possible with Atomic Broadcast. To distinguish "genuine" Generic Broadcast protocols from those that order all messages, Pedone and Schiper [102] proposed the following definition. A Generic Broadcast algorithm is *strict* if there are runs in which non-conflicting messages are delivered in different orders by different learners. Since Atomic Broadcast orders all messages, it might seem that only "genuine" Generic Broadcast protocols are strict. However, this is not true, because the strictness condition is trivial to satisfy by a simple modification of any Atomic Broadcast protocol, without any performance gain [102].

To solve this problem, Aguilera et al. [5] proposed two definitions. A Generic Broadcast algorithm is *non-trivial* if it uses an oracle, such as a failure detector or an agreement abstraction, only if some of the messages conflict. A *thrifty* algorithm additionally ensures that if eventually messages do not conflict, the algorithm will eventually stop using the oracle. In this context, the oracle can be a failure detector or an agreement abstraction such as Consensus or Atomic Broadcast.

In this thesis, we take a more direct approach and distinguish genuine and non-genuine Generic Broadcast protocols based on their latency. We are interested only in Generic Broadcast algorithms that offer better latency than is possible with Atomic Broadcast.

5.2.2 Optimistic Generic Broadcast

Optimistic Atomic Broadcast [99] takes two communication steps to deliver messages in *ordered* runs. On the other hand, Generic Broadcast [5, 98, 102] delivers messages in two steps in stable *conflict-free* runs, with no conflicting messages. Our Optimistic Generic Broadcast algorithm subsumes these two approaches and delivers messages in two steps in all stable *conflict-ordered* runs, where all *conflicting* messages are received by correct acceptors in the same order.

Algorithm	all	conflict-free	ordered	conflict-ordered
Chandra and Toueg [16]	3	3	3	3
Generic Broadcast [102]	4	2	4	4
Opt. Atomic Broadcast [101]	4	4	2	4
Atomic Broadcast (Section 5.1.7)	3	3	2	3
Opt. Generic Broadcast	3	2	2	2

Figure 5.8: A comparison of the latencies of several Atomic/Generic Broadcast algorithms in four kinds of stable runs.

Every ordered run is conflict-ordered because if acceptors receive *all* messages in the same order, then they also receive the conflicting ones in the same order. Similarly, every conflict-free run is conflict-ordered because in the absence of conflicting messages, acceptors receive all conflicting messages (none) in the same order. On the other hand, there are conflict-ordered runs which are neither ordered nor conflict-free. For example, assume that two acceptors a_1 and a_2 received messages in orders

$$a_1 : m_1, m_2, m_3, \qquad a_2 : m_2, m_3, m_1,$$

and only messages m_2 and m_3 conflict.

Figure 5.8 compares the latencies of several Atomic/Generic Broadcast algorithms in four categories of stable runs: all, conflict-free, ordered, and conflict-ordered. For each of these, it shows the number of communication steps needed by the algorithms to deliver messages. Each of the algorithms has one category of runs for which it has been optimized. The number of communication steps for these categories have been boxed. However, the comparison shows that, in terms of latency, our algorithm performs at least as well as the other algorithms *in all categories*. Moreover, our algorithm is strictly better than the other known protocols, even if we ignore conflict-ordered runs, for which our algorithm has been optimized. In particular, the latency of our algorithm never exceeds that of the modified version of Chandra-Toueg algorithm from Section 5.1.3. Neither Optimistic Atomic Broadcast [101] nor Generic Broadcast [102] has this property.

5.2.3 Lower bounds

The latency of our algorithm is optimal in all four categories. First, Section 5.5.1 shows that no Generic Broadcast algorithm can deliver messages faster than in two communication steps. Theorem 5.5.3 shows that three communication steps in general stable runs are also necessary. Internally, our Generic Broadcast algorithm uses the one-two step Consensus from Section 5.1.8, which requires that less than a third of the acceptors are faulty ($n > 3f$). Pedone and Schiper [100] showed that this condition is necessary for any Generic Broadcast algorithm capable of delivering messages in two communication steps.

5.2.4 Basic Generic Broadcast algorithm

In this section, we present a simplified version of our Optimistic Generic Broadcast algorithm. This version is always safe; it works correctly if no failures occur, but it might not make progress in the presence of faults. Section 5.2.5 then shows how to extend this algorithm to obtain a fully correct Generic Broadcast algorithm.

Partial order on messages

The algorithm operates by agreeing on the delivery order of each pair of conflicting messages. More precisely, acceptors cooperate in building a partial order "$\rightarrow$" on conflicting messages, and learners deliver messages in any order consistent with this partial order. For any two messages m and m', the relation $m \rightarrow m'$ requires m to be delivered before m'. Since non-conflicting messages can be delivered in any order, the relation "$\rightarrow$" is defined only for pairs of messages $\{m, m'\}$ that conflict. For these, we expect that eventually either $m \rightarrow m'$ or $m' \rightarrow m$.

The following diagram shows an example with four messages. All pairs of messages conflict, except for m_2 and m_3, which can be delivered in different orders by different learners.

Learners deliver messages in any order consistent with the partial order "$\rightarrow$". In the first example, "$\rightarrow$" is defined for all pairs of conflicting messages. Learners can deliver the four messages in one of two orders m_1, m_2, m_3, m_4 or m_1, m_3, m_2, m_4, as both are consistent with "$\rightarrow$". Messages m_2 and m_3 can be delivered in different orders by different learners. This does not violate the Generic Agreement property because these messages do not conflict.

In the second example, the relation between conflicting messages m_2 and m_4 is not known (yet). As a result, none of them can be delivered. However, whatever the order of m_2 and m_4 will be, one of the orders: m_1, m_3, m_2, m_4 and m_1, m_3, m_4, m_2 will be consistent with "$\rightarrow$". These two orders share a common prefix m_1, m_3, so messages m_1 and m_3 can be delivered straight away.

Another way of looking at the delivery process is to realize that m_1 can be delivered because $m_1 \rightarrow m$ for all $m \neq m_1$. After m_1 has been delivered, we can deliver m_3 because $m_3 \rightarrow m$ for all undelivered m's conflicting with m_3. Conflicting messages m_2 and m_4 will be delivered only when their order is known.

```
1   when a proposer executes gbcast(m) do
2     broadcast m to the acceptors
3   when an acceptor receives m for the first time do
4     for all possible non-received messages m' conflicting with m do
5       first_{m,m'}.propose(m)
6   when first_{m,m'}.decision(m) at a learner do
7     set m → m'
8   when a learner has not gdelivered m,
9         and has m → m' for all undelivered messages m' conflicting with m do
10    gdeliver(m)
```

Figure 5.9: Basic Generic Broadcast algorithm.

Basic algorithm

In our algorithm, learners agree on the partial order "$\rightarrow$" by using Consensus to agree on the delivery order of every pair of conflicting messages. In other words, for each *unordered* pair $\{m, m'\}$ of conflicting messages, we use a separate instance of Consensus. In each such instance $\mathit{first}_{m,m'}$, each acceptor a proposes the message m or m' that arrived at a first.

The resulting partial order is built based on decisions of the Consensus instances $\mathit{first}_{m,m'}$. If the instance decides on m, then m is deemed to be the first message of the two ($m \rightarrow m'$). Hence, if the instance $\mathit{first}_{m,m'} = \mathit{first}_{m',m}$ decides on m', then $m' \rightarrow m$. Messages are delivered in an order consistent with "$\rightarrow$".

Figure 5.9 shows the basic algorithm. Proposers gbcast their messages using ordinary broadcast. When an acceptor receives a message m, it proposes m to the instances $\mathit{first}_{m,m'}$ for all messages m' conflicting with m that have not been received (yet). In other words, the acceptor proposes m to precede all such messages m' in the delivery order. A learner delivers a message m once it has $m \rightarrow m'$ for all possible undelivered messages m' conflicting with m.

Since the set of all messages is usually infinite, the $\mathit{receive}(m)$ action involves executing infinitely many parallel instances of Consensus. Section 5.3 will show how to achieve this with finite resources. Until then, we stick with the "infinite" version of the algorithm because it is easier to understand.

The algorithm satisfies the safety properties: Validity and Generic Agreement. For the former, we assume the existence of an artificial message $\perp$, which is never sent and conflicts with all other messages. Therefore, delivering any message m requires $m \rightarrow \perp$, which means that some acceptor must have proposed m to $\mathit{first}_{m,\perp}$, so some proposer must have gbcast m.

To prove Generic Agreement, we will assume, to derive a contradiction, that conflicting

messages m and m' are delivered in different orders at different learners. This would mean that $m \rightarrow m'$ at one of the learners, and $m' \rightarrow m$ at another, which is impossible by the Agreement property of the underlying Consensus.

Termination Validity and Termination Agreement are not always satisfied by this basic version of the Generic Broadcast algorithm.

Latency

The basic algorithm satisfies

> **Latency.** In stable runs, a message gbcast by a correct proposer is delivered by all correct learners within two steps if the run is conflict-ordered, and three steps otherwise.

To prove this, we assume that the underlying Consensus algorithm satisfies:

C1: In stable runs, if all correct acceptors proposed the same value, then all correct learners decide on that value in one communication step.

C2: In stable runs, all correct learners decide on the value proposed by the leader in two communication steps after the leader proposed.

Such a Consensus algorithm was presented in Section 5.1.8.

In stable runs, any gbcast message is received by the leader in one communication step. Property **C2** ensures that the order is decided two communication steps later, leading to three communication steps in total for latency. If all acceptors receive conflicting messages in the same order, then they propose the same order to Consensus instances. Therefore, Property **C1** implies that a decision will be made in one communication step, resulting in a total latency of two steps in conflict-ordered runs.

5.2.5 Full Generic Broadcast algorithm

Before presenting the full version of the algorithm, we will highlight two main problems with the basic version in runs with failures.

Cycles

In scenarios with failures, the relation "$\rightarrow$" built by the basic algorithm may contain cycles, thereby leading to a deadlock. In stable runs, this is impossible, because Property **C2** ensures that "$\rightarrow$" reflects the linear order of message reception at the leader. In non-stable runs, however, different parts of the "$\rightarrow$" relation might have been proposed by different acceptors.

As an example, consider a system with three acceptors a_1, a_2, a_3. Each of these processes receives three messages m_1, m_2, m_3, in different orders:

$$\begin{aligned} a_1 &: m_1, m_2, m_3 && \text{executes } \mathit{first}_{m_1,m_2}.\mathit{propose}(m_1) \\ a_2 &: m_2, m_3, m_1 && \text{executes } \mathit{first}_{m_2,m_3}.\mathit{propose}(m_2) \\ a_3 &: m_3, m_1, m_2 && \text{executes } \mathit{first}_{m_3,m_1}.\mathit{propose}(m_3) \end{aligned}$$

If all of the above proposals become decisions, then the cycle

$$m_1 \rightarrow m_2 \rightarrow m_3 \rightarrow m_1$$

will be formed, and as a result none of these messages will ever be delivered.

To cope with cycles, we introduce the notion of blocked messages. A message is *blocked* if it belongs to a cycle, or it is a successor of a blocked message. In our example, all three messages are blocked, and any message m_4 with, say, $m_2 \rightarrow m_4$ would be blocked as well. Obviously, blocked messages will never be delivered by the basic algorithm. In the full version of the algorithm, we will sometimes deliver blocked messages to break cycles and avoid deadlocks.

Faulty proposers

Another problem with the basic algorithm are faulty proposers. If a proposer crashes, then its message may reach only a subset of the acceptors, which might prevent progress. This problem has already been discussed in Section 5.1.4. Recall that the solution requires acceptors to re-gbcast every message they see.

Algorithm

Figure 5.10 shows the full version of our algorithm. To resolve cycles and be able to deliver blocked messages, we use an auxiliary Atomic Broadcast protocol. Since cycles do not appear in stable runs, the latency of that Atomic Broadcast protocol does not affect the latency of our algorithm.

A proposer gbcasts a message m by broadcasting it to all acceptors. As in the basic version, when an acceptor receives m for the first time, it executes $\mathit{first}_{m,m'}.\mathit{propose}(m)$ for all possible messages m' that it has not received yet. It also abcasts m using the underlying Atomic Broadcast protocol. As we will see later, this will help resolving potential cycles.

Lines 9–10 construct the order "$\rightarrow$" in the same way as in the basic version. In the full version, messages can be delivered either normally or during cycle resolution. To distinguish these two kinds of deliveries, we call the former 1-delivery, and the latter 2-delivery. Messages are 1-delivered in exactly the same way as in the basic version (lines 11–13).

```
1   when a proposer executes gbcast(m) do
2     broadcast m to the acceptors
3   when an acceptor sees m eventually do
4     gbcast(m)
5   when an acceptor receives m for the first time do
6     for all possible non-received messages m' conflicting with m do
7       first_{m,m'}.propose(m)
8     abcast(m)
9   when first_{m,m'}.decision(m) at a learner do
10    set m → m'
11  when a learner has not gdelivered m,
12        and has m → m' for all undelivered messages m' conflicting with m do
13    gdeliver_1(m)
14  task cycle resolution at a learner is
15    repeat forever
16      wait until adeliver(m)
17      wait until m has been gdelivered or
18                 all undelivered messages conflicting with m are blocked
19      if m has not been gdelivered yet then
20        gdeliver_2(m)
```

Figure 5.10: Full Generic Broadcast algorithm.

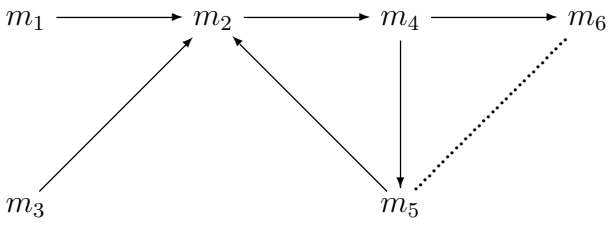
For 2-delivery, each learner executes the cycle resolution loop over messages adelivered by the underlying Atomic Broadcast protocol. For each such message m, the learner waits until one of the two conditions holds. If m has already been delivered, then the loop goes to the next iteration. Otherwise, if all undelivered messages conflicting with m are blocked, then the learner 2-delivers m. The rationale behind this strategy is that, since none of the blocked messages can be 1-delivered, it is safe to deliver m, thereby, possibly, breaking the cycle. The use of Atomic Broadcast ensures that the messages m chosen to break cycles are the same at all learners.

To guarantee Termination Agreement, we apply the method from Section 5.1.4: when an acceptor sees a message, it re-gbcasts it (lines 3–4). In this algorithm, we assume that an acceptor sees a message m iff it sees it in the underlying Atomic Broadcast algorithm or in one of the Consensus instances $\mathit{first}_{m,m'}$.

To show that lines 17–18 always terminate, we must prove that all never-blocked messages m' conflicting with m will eventually be delivered. Since m' is not blocked, the graph of paths $m' \leftarrow m_1 \leftarrow \cdots \leftarrow m_k$ of undelivered messages does not contain cycles, and forms a tree rooted at m'. The leaves of this tree will be successively 1-delivered, resulting eventually in 1-delivery of m'.

Example

Consider a run with six messages, and assume that the underlying Atomic Broadcast protocol delivers the messages in order $m_1, \ldots, m_6$.



Both m_1 and m_3 can be 1-delivered because the only message conflicting with them (m_2) is their successor. Messages m_1 and m_3 can be delivered in any order; this does not violate Generic Agreement because they do not conflict. At the same time, the cycle resolution task adelivers m_1. Since the only message conflicting with it (m_2) is blocked by the cycle $m_2 \rightarrow m_4 \rightarrow m_5 \rightarrow m_2$, message m_1 is 2-delivered if it has not been 1-delivered yet.

After m_1 and m_3 have been delivered, no other message can be 1-delivered, because all messages are blocked. The cycle resolution task adelivers m_2, and since all undelivered messages conflicting with it (m_4, m_5) are blocked, m_2 is 2-delivered. This delivery breaks the cycle, and now all undelivered messages conflicting with m_4 (i.e., m_5 and m_6) are its successors, so m_4 is 1-delivered. Conflicting messages m_5 and m_6 cannot be delivered until their order is known. If, for example, $m_5 \rightarrow m_6$, then message m_5 will be 1-delivered, followed by m_6.

5.3 Handling infinitely many instances of Consensus

Our Generic Broadcast algorithm uses infinitely many Consensus instances. This section briefly explains how to implement an infinite number of *virtual instances* of a distributed algorithm using only finitely many *physical instances* at every process. As we will see, this is possible, provided that there are only finitely many *different* virtual instances. For example, in our Generic Broadcast algorithm from Figure 5.10, a process proposes *the same* message m to an infinite number of instances of Consensus.

Let us start with finitely many virtual instances, and denote these by $i_1, \ldots, i_k$. The usual approach is to tag any event (a message or a function call) with the identifier of the instance. Each process runs k physical instances of the algorithm. Every event tagged with i_k is directed to the k-th instance, and every event produced by the k-th instance is tagged with i_k. In this case, virtual and physical instances are the same, so this method can be used only with finitely many virtual instances.

To implement an infinite number of virtual instances, some of them must share a single physical instance. The algorithm in Figure 5.11 maintains a family $\mathcal{I}$ of disjoint sets of virtual instances, initially empty. Each element $I \in \mathcal{I}$ is a set of virtual instances that

```
1  I is the family of disjoint sets of virtual instances, initially empty
2  when a process receives an event e tagged with the set E do
3    for each I ∈ I do                                          { refine I }
4      split I into I ∩ E and I \ E
5    J ← E                                                { compute J = E \ ⋃I }
6    for each I ∈ I do
7      J ← J \ I
8    add J to I
9    for each I ∈ I do          { pass the event to the appropriate physical instances }
10     if I ∩ E ≠ ∅ then
11       send the event e to instance A_I
```

Figure 5.11: Emulating infinitely many virtual instances.

share a single physical instance denoted as A_I. All events generated by A_I are tagged with the set I.

When an event e tagged with a set of virtual instances E arrives, the process does the following. First, if some virtual instances sharing the same physical instance of the algorithm start to differ, the physical instance is cloned. This is done by splitting all elements $I \in \mathcal{I}$ into $I \cap E$ and $I \setminus E$, so that every element of $\mathcal{I}$ is either a subset of E or disjoint with it. When such a split happens, the physical instance A_I is replaced with two new instances $A_{I \cap E}$ and $A_{I \setminus E}$, both identical to A_I. Also, a new physical instance is created for $J = E \setminus \bigcup \mathcal{I}$ to ensure that every virtual instance corresponds to some physical instance; in other words, we want to make sure that $E \subseteq \bigcup \mathcal{I}$. Any instances A_J with $J = \emptyset$ introduced by the above steps are ignored. Finally, the event is sent to all physical instances corresponding to any virtual instances in E.

Example

Consider a two-acceptor system running four virtual instances of Atomic Commitment, with the following proposals:

	i_1	i_2	i_3	i_4
acceptor a_1	COMMIT	COMMIT	COMMIT	ABORT
acceptor a_2	COMMIT	ABORT	ABORT	ABORT

Acceptor a_1 proposes COMMIT to virtual instances i_1, i_2, i_3, and ABORT to instance i_4. In other words, it makes two invocations of the algorithm from Figure 5.11. The first invocation, with $e = \mathit{propose}(\text{COMMIT})$ and $E = \{i_1, i_2, i_3\}$, encounters $\mathcal{I} = \emptyset$. It adds $J = E = \{i_1, i_2, i_3\}$ to $\mathcal{I}$ and creates a new physical instance A_{123} corresponding to $\{i_1, i_2, i_3\}$. Finally, the event $e = \mathit{propose}(\text{COMMIT})$ is passed to physical instance A_{123}. The second

invocation, with $e = \mathit{propose}(\text{ABORT})$ and $E = \{i_4\}$, encounters $\mathcal{I} = \{\{i_1, i_2, i_3\}\}$. Similarly, it adds $\{i_4\}$ to $\mathcal{I}$, creates a new instance A_4, and passes e to it. At this point, acceptor a_1 has $\mathcal{I} = \{\{i_1, i_2, i_3\}, \{i_4\}\}$.

At the same time, acceptor a_2 proposes ABORT to virtual instances i_2, i_3, i_4, and COMMIT to instance i_1. The first invocation, with $e = \mathit{propose}(\text{ABORT})$ and $E = \{i_2, i_3, i_4\}$, encounters $\mathcal{I} = \emptyset$, adds $\{i_2, i_3, i_4\}$ to $\mathcal{I}$, creates a new instance A_{234}, and passes e to it. At this point $\mathcal{I} = \{\{i_2, i_3, i_4\}\}$.

Consider a message (event) e informing a_2 that a_1 proposed COMMIT to instances i_1, i_2, i_3. To make this example more interesting, assume e arrives at a_2 before a_2 proposes COMMIT to instance i_1. This results in the algorithm from Figure 5.11 being invoked with $E = \{i_1, i_2, i_3\}$ and $\mathcal{I} = \{\{i_2, i_3, i_4\}\}$. First $\mathcal{I} = \{\{i_2, i_3, i_4\}\}$ is refined by splitting $I = \{i_2, i_3, i_4\}$ into $I \cap E = \{i_2, i_3\}$ and $I \setminus E = \{i_4\}$, resulting in $\mathcal{I} = \{\{i_2, i_3\}, \{i_4\}\}$. This operation replaces A_{234} with two identical physical instances A_{23} and A_4. Then, the set $E \setminus \bigcup \mathcal{I} = \{i_1\}$ is added to $\mathcal{I}$, and the corresponding physical instance A_1 is created. The message e is passed to instances A_1 and A_{23}. By proposing ABORT to virtual instance i_4, acceptor a_2 does not change $\mathcal{I}$ nor does it create any new physical instances. The event $\mathit{propose}(\text{ABORT})$ is passed to A_4.

In a similar way, a_1 will eventually end up with the same $\mathcal{I} = \{\{i_1\}, \{i_2, i_3\}, \{i_4\}\}$. No further refinement will happen, because both acceptors issued the same proposal to virtual instances i_2 and i_3. As a result, these instances will be handled by the same physical instance A_{23}:

	A_1	A_{23}	A_4
acceptor a_1	COMMIT	COMMIT	ABORT
acceptor a_2	COMMIT	ABORT	ABORT

5.3.1 Representing sets

The above method can be used to execute an infinite number of Consensus instances at the same time, provided that we can represent infinite sets of instances in a finite form. For use in the algorithm from Figure 5.11, the families of representable sets must be closed under subtraction and intersection. Closedness under set union is not necessary; $E \setminus \bigcup \mathcal{I}$ can be computed by iteratively subtracting elements of $\mathcal{I}$ from E. In this section, we briefly discuss such representations for some families of sets that are useful in our Generic Broadcast algorithm.

Border sets

Finite sets can be trivially represented by listing their elements. The family of *border sets* contains all sets that are either finite (F) or are complements of finite sets ($\overline{F}$). For

example, the set $\overline{\{m_1, m_2\}}$ consists of all messages except for m_1 and m_2. The representation of a border set consists of the finite set F and a flag indicating whether the set is F or $\overline{F}$. The family of border sets is closed under negation and intersection (which implies subtraction):

$$F_1 \cap F_2 = F_1 \cap F_2, \qquad F_1 \cap \overline{F_2} = F_1 \setminus F_2,$$
$$\overline{F_1} \cap F_2 = F_2 \setminus F_1, \qquad \overline{F_1} \cap \overline{F_2} = \overline{F_1 \cup F_2}.$$

M-sets

If only messages m_1 and m_2 have been received, then the border set $\overline{\{m_1, m_2\}}$ represents the set of all non-received messages. Can we use border sets to represent more complex sets such as "the set of all non-received messages conflicting with m"? The answer depends on the conflict relation. It is often the case that messages can be divided into a small number of categories (e.g., "read" and "write"), such that conflict properties of messages are determined by the categories they belong to. Consider a system with k categories $C_1, \ldots, C_k$, where C_i is the set of all messages in the i-th category. For any border sets $B_1, \ldots, B_k$ satisfying $B_i \subseteq C_i$, we define an *m-set*

$$\langle B_1, \ldots, B_k \rangle = B_1 \cup \cdots \cup B_k$$

to be the set containing all messages from sets $B_1, B_2, \ldots, B_k$.

As an example, consider a system with two categories: "read" and "write", where any two requests conflict unless they are both reads. Assume that requests w_1, w_2, r_1, and r_2 have been received. The set of all non-received requests conflicting with r_2 is $\langle \emptyset, \overline{\{w_1, w_2\}} \rangle$, that is, no read requests and all possible write requests except for w_1 and w_2.

The family of m-sets is closed under subtraction and intersection:

$$\langle B_1, \ldots, B_k \rangle \cap \langle B'_1, \ldots, B'_k \rangle = \langle B_1 \cap B'_1, \ldots, B_k \cap B'_k \rangle,$$
$$\langle B_1, \ldots, B_k \rangle \setminus \langle B'_1, \ldots, B'_k \rangle = \langle B_1 \setminus B'_1, \ldots, B_k \setminus B'_k \rangle.$$

Sets of message pairs

In our Generic Broadcast algorithm, each Consensus instance is identified by an unordered pair of messages. By $\{\!\{m, M\}\!\}$ we denote the set of pairs containing m and one element of the m-set M:

$$\{\!\{m, M\}\!\} = \{\, \{m, m'\} \mid m' \in M \,\}$$

For example, M can be the set of all possible non-received messages m' that conflict with a given message m. Consider the actions performed by acceptors in our Generic

Broadcast algorithm in Figure 5.10 upon receiving a new message m. We can replace infinitely many invocations of $\mathit{first}_{m,m'}.\mathit{propose}(m)$ with a single $\mathit{first}_{\{\!\{m,M\}\!\}}.\mathit{propose}(m)$. The family of sets $\{\!\{m, M\}\!\}$ can be used in the algorithm from Figure 5.11 because it is closed under intersection and subtraction (we assume $m \neq m'$):

$$\begin{aligned}
\{\!\{m, M\}\!\} \cap \{\!\{m, M'\}\!\} &= \{\!\{m, M \cap M'\}\!\},\\
\{\!\{m, M\}\!\} \setminus \{\!\{m, M'\}\!\} &= \{\!\{m, M \setminus M'\}\!\},\\
\{\!\{m, M\}\!\} \cap \{\!\{m', M'\}\!\} &= \begin{cases} \{\!\{m, \{m'\}\}\!\} & \text{if } m \in M' \text{ and } m' \in M,\\ \{\!\{m, \emptyset\}\!\} & \text{otherwise,} \end{cases}\\
\{\!\{m, M\}\!\} \setminus \{\!\{m', M'\}\!\} &= \begin{cases} \{\!\{m, M \setminus \{m'\}\}\!\} & \text{if } m \in M' \text{ and } m' \in M,\\ \{\!\{m, M\}\!\} & \text{otherwise.} \end{cases}
\end{aligned}$$

5.3.2 Representing intervals

In Section 5.4, we will describe an Atomic Broadcast algorithm that uses sets of instances identified by intervals of real numbers. This section shows how to represent these sets. Let us start with the family of closed-open intervals $[a, b) = \{\, x \mid a \leq x < b \,\}$. Intersection is easy to define

$$[a, b) \cap [a', b') = \big[\max\{a, a'\}, \min\{b, b'\}\big)$$

However, subtraction is tricky because it might produce a union of two intervals, for example

$$[1, 5) \setminus [2, 3) = [1, 2) + [3, 5).$$

In general, by $X = X_1 + \cdots + X_k$ we mean that $\{X_1, \ldots, X_k\}$ is a partition of X. In other words, $X = \bigcup_i X_i$ and sets X_i are pairwise disjoint.

The algorithm from Figure 5.11 assumes that for any representable sets I and J, sets $I \cap J$ and $I \setminus J$ are also representable. Figure 5.12 shows an extended version that allows $I \setminus J$ to be a disjoint union of representable sets. In other words, it assumes that $I \setminus J = I_1 + \cdots + I_k$ for some representable sets I_i, that is, $\{I_1, \ldots, I_k\}$ is a partition of $I \setminus J$. This modification allows us to represent all intervals $[a, b) = \{\, x \mid a \leq x < b \,\}$; the necessary operations are defined as

$$\begin{aligned}
[a, b) \cap [a', b') &= \big[\max\{a, a'\}, \min\{b, b'\}\big)\\
[a, b) \setminus [a', b') &= \big[a, \min\{a', b\}\big) + \big[\max\{a, b'\}, b\big)\\
[a, b) &= \emptyset \Leftrightarrow a \geq b
\end{aligned}$$

Other intervals, such as $[a, b]$, $(a, b]$ or (a, b), can be represented in a similar way. Consider an extended set of real numbers, such that for each number x, the symbol x^+

```
1   I is the family of disjoint sets of virtual instances, initially empty
2   when a process receives an event e tagged with the set E do
3     for each I ∈ I do                                          { refine I }
4       let {I_1, ..., I_k} be a partition of I \ E
5       split I into I ∩ E and I_1, ..., I_k
6     let J = {E}                          { compute J such that ⋃J = E \ ⋃I }
7     for each I ∈ I do
8       for each J ∈ J do
9         let {J_1, ..., J_k} be a partition of J \ I
10        replace J in J with J_1, ..., J_k
11    for each J ∈ J do
12      add J to I and create a new physical instance A_J
13    for each I ∈ I do          { pass the event to the appropriate physical instances }
14      if I ∩ E ≠ ∅ then
15        send the event e to instance A_I
```

Figure 5.12: Emulating infinitely many virtual instances (extended version).

represents a "number" that is infinitesimally larger than x. Formally, consider $\mathbb{R}^+ = \{\, x, x^+ \mid x \in \mathbb{R} \,\}$, with the order "$<_{\mathbb{R}^+}$" defined as follows

$$x <_{\mathbb{R}^+} x^+ \text{ for all } x \in \mathbb{R} \qquad \text{and} \qquad x^{(+)} <_{\mathbb{R}^+} y^{(+)} \iff x <_{\mathbb{R}} y \text{ for any } x \neq y,$$

where "$<_{\mathbb{R}}$" is the standard order "$<$" on $\mathbb{R}$.

Then, we can represent all intervals in $\mathbb{R}$ as closed-open intervals over $\mathbb{R}^+$

$$[a,b] = \{\, x \mid a \leq x \leq b \,\} = [a\ \ , b^+) \cap \mathbb{R}, \qquad [a,b) = \{\, x \mid a \leq x < b \,\} = [a\ \ , b) \cap \mathbb{R},$$
$$(a,b] = \{\, x \mid a < x \leq b \,\} = [a^+, b^+) \cap \mathbb{R}, \qquad (a,b) = \{\, x \mid a < x < b \,\} = [a^+, b) \cap \mathbb{R}.$$

5.4 Atomic Broadcast in closed groups

Défago et al. [26] distinguish between Atomic Broadcast algorithms operating in *open groups* and those operating in *closed groups*. In our model, open groups correspond to the scenario in which proposers, who abcast messages, are not necessarily acceptors. In closed groups, the sets of acceptors and proposers are the same. All broadcast protocols presented so far in this chapter can be used in open groups because they do not assume anything about the proposers.

The open-group model is more general, so algorithms designed for it remain correct in the closed-group model. However, restricting the set of proposers to the acceptors may enable us to achieve a latency that would be impossible in open groups. In this section,

we present an Atomic Broadcast protocol for closed groups that delivers messages in two communication steps in *all* good runs. In comparison, the best algorithm for open groups achieved a latency of two steps only when all conflicting messages were received by all correct acceptors in the same order (Section 5.2). Theorem 5.5.3 shows that no open group Atomic Broadcast protocol can guarantee latency of less than three communication steps in all good runs. The closed-group Atomic Broadcast algorithm presented in this section guarantees a two-step latency in all good runs, which is optimal (Section 5.5.1).

The algorithm presented in this section uses real-time clocks. It delivers messages in two steps in all good runs in which the clocks are synchronized. However, the good-run and clock-synchronization assumptions are required only to meet the Latency property; the other properties are satisfied even if these assumptions do not hold.

Our algorithm satisfies several lower bounds. Firstly, it requires only a majority of correct acceptors ($n > 2f$) and the $\Diamond S$ failure detector [16]. Secondly, in good runs, it delivers all messages in two communication steps (Section 5.5.1). Thirdly, Theorems 5.5.1 and 5.5.4 show that the good-run and clock-synchronization conditions, under which it achieves the two-step latency, cannot be relaxed. Finally, our algorithm is *quiet*, that is, no network messages are sent unless some messages are abcast.

5.4.1 Related work

Défago et al. [26] survey more than fifty Atomic Broadcast protocols. In this section, however, we are only interested in algorithms which exhibit latency of less than three communication steps in all good runs. Theorems 5.5.1 and 5.5.3 show that such algorithms require synchronized real-time clocks and cannot handle open groups. The only two algorithms satisfying these criteria are HAS [25] and the algorithm proposed by Vicente and Rodrigues [117].

Vicente and Rodrigues [117] proposed an Atomic Broadcast algorithm for closed groups that achieves a latency of $2d + \delta$, where $\delta > 0$ is an arbitrarily small constant. The price for having a very small δ is high network traffic; the number of messages used by the algorithm is proportional to $1/\delta$. Moreover, these messages are sent even if no messages are abcast in the system. In comparison, our algorithm achieves the latency of $2d$ and sends network messages only if acceptors actually abcast messages.

To the best of our knowledge, no other Atomic Broadcast algorithm with latency of less than $3d$ in all good runs has been proposed in the literature. The aforementioned HAS [25] bases its *safety* on timing assumptions, which can lead to the violation of Agreement if they do not hold [26]. The optimal latency of two communication steps reported in [39, 89] results merely from not counting the first step, in which the proposers broadcast their messages to the acceptors.

```
1   when an acceptor executes abcast(m) do
2     insert m into M
3   task broadcasting at acceptors is
4     for t = t_0, t_0 + δ, t_0 + 2δ, ... do
5       M ← ∅
6       wait until current-time ≥ t + δ
7       messages_t.propose(M)
8   task delivery at learners is
9     for t = t_0, t_0 + δ, t_0 + 2δ, ... do
10      wait until messages_t.decision([M_1, ..., M_n])
11      for i = 1, 2, ..., n do
12        adeliver all messages in M_i in some deterministic order
```

Figure 5.13: Basic version of the two-step Atomic Broadcast algorithm for closed groups.

5.4.2 Basic version

We will start by presenting a simplified version of the algorithm. This version is always safe; it works correctly if no failures occur, but it might not progress in the presence of faults. The delivery latency of our basic algorithm is $2d + \delta$, where $\delta > 0$ is an arbitrarily small parameter. The next section will show to reduce the latency to $2d$ and extend the algorithm to obtain a fully correct Atomic Broadcast algorithm.

The algorithm uses a sequence of instances of Interactive Consistency, which was discussed in detail in Section 4.4.6. Recall that in Interactive Consistency, each acceptor a_i issues a proposal x_i, and all learners agree on some vector $[y_1, \ldots, y_n]$. In good runs, this vector is $[x_1, \ldots, x_n]$, that is, it contains acceptors' proposals. If the run is not timely or acceptor a_i is faulty, the entry y_i can be ABORT instead of x_i. The Agreement property holds in all runs; the decision vector $[y_1, \ldots, y_n]$ is the same at all learners.

Figure 5.13 shows the basic version of our Atomic Broadcast algorithm. We assume that acceptors are equipped with perfectly synchronized real time clocks. Conceptually, we divide the continuous real time into discrete *timeframes* of length δ each. We label the timeframes by the times at which they begin, that is, timeframe t starts at time t and ends at time $t + \delta$. The first timeframe is t_0, the next is $t_0 + \delta$, then $t_0 + 2\delta$, etc.

When an acceptor wants to abcast a message m, it just inserts it to its set $\mathcal{M}$. This set is emptied at the beginning of each timeframe. At the end of each timeframe, all messages in $\mathcal{M}$ are broadcast together in a single batch.

To perform this batch broadcasting, the acceptor loops over all timeframes (lines 3–7). In each timeframe t, the acceptor empties the set $\mathcal{M}$, and waits until the timeframe finishes, at time $t + \delta$. At this time, $\mathcal{M}$ contains all messages abcast by the acceptor in timeframe t. The acceptor proposes $\mathcal{M}$ to an Interactive Consistency instance $messages_t$. Each timeframe t uses its own, independent Interactive Consistency instance $messages_t$.

The delivery task at each learner also contains a loop that iterates over all timeframes. For each timeframe t, the learner first waits until the Interactive Consistency instance $message_t$ decides on some vector $[\mathcal{M}_1, \ldots, \mathcal{M}_n]$. In good runs, each $\mathcal{M}_i$ is the set of messages abcast by acceptor a_i in timeframe t. For all acceptors $a_1, \ldots, a_n$, the learner delivers all messages in $\mathcal{M}_i$ in some deterministic order. The value ABORT is treated synonymously with the empty set $\emptyset$, that is, $\mathcal{M}_i = \text{ABORT} \iff \mathcal{M}_i = \emptyset$.

The above algorithm is always safe. The Agreement property of Interactive Consistency implies that all timeframes decide on the same sets of messages. Therefore, the sequences of delivered messages are the same at all learners (Agreement). For Validity, consider the decision vector $[\mathcal{M}_1, \ldots, \mathcal{M}_n]$ of instance $messages_t$. Each entry $\mathcal{M}_i$ can either contain the messages a_i abcast during timeframe t or be empty (ABORT). In both cases, $\mathcal{M}_i$ contains only messages abcast by a_i (Validity).

Liveness properties are guaranteed only in good runs. For Termination Validity, notice that each message m abcast by a correct acceptor a_i eventually becomes an element of a_i's proposal to some instance $messages_t$. Consider the decision vector $[\mathcal{M}_1, \ldots, \mathcal{M}_n]$ of instance $messages_t$. In good runs, the Validity property of Interactive Consistency guarantees that $m \in \mathcal{M}_i$. Since all instances $messages_t$ will eventually decide, message m will eventually be delivered by all correct learners. In non-good runs, however, $\mathcal{M}_i$ can be empty (ABORT) and message m_i might not be delivered. In good runs, all proposers (acceptors) are correct, so Termination Agreement follows from Validity and Termination Validity.

Latency

In good runs, our algorithm delivers all messages within $2d + \delta$ time. Consider any message m abcast by acceptor a_i in timeframe t, that is, between t and $t + \delta$. At time $t + \delta$, all acceptors propose to $messages_t$ the set of messages received in timeframe t. The instance $messages_t$, implemented as in Section 4.4.6, decides on vector $[\mathcal{M}_1, \ldots, \mathcal{M}_n]$ in two communication steps from $t + \delta$, that is, at time $t + \delta + 2d$. Since $m \in \mathcal{M}_i$, message m is delivered at time $t + 2d + \delta$, at most $\delta + 2d$ units of time after it was sent. The Atomic Broadcast algorithm by Vicente and Rodrigues [117] achieves the same latency.

5.4.3 Full version

Section 5.4.2 presented an algorithm with a latency of $2d + \delta$, where δ is the size of each timeframe. The smaller the timeframe, the smaller the delivery latency. However, each timeframe requires a separate instance of Interactive Consistency. As a result, the amount of resources required (messages and computation) is inversely proportional to the size of the timeframes. As the size of the timeframes goes to zero, the latency approaches $2d$ and the resource usage goes to infinity. The algorithm by Vicente and Rodrigues [117]

suffers from the same problem. In this section, we will modify the basic algorithm from Section 5.4.2 to guarantee the latency of $2d$. Effectively, we will show how to set the timeframe size δ to 0 without incurring infinite resource usage.

Zero-sized timeframes correspond to each real time t having a separate timeframe t. In this case, the basic algorithm from Figure 5.13 suffers from two problems. First, it uses an infinite number of instances $messages_t$. Second, the broadcasting and delivery tasks contain loops that iterate over now infinite number of timeframes. We will deal with these problems in the next two sections.

Infinitely many instances

First, we present a method of dealing with an infinite number of $message_t$ instances. As shown in Section 5.3, executing an infinite number of instances with finite resources is possible, provided that there are only finitely many *different* instances. We will show that this is true in this case. Let us call a given timeframe *active* iff at least one acceptor abcasts a message in that timeframe. At any time, the number of active timeframes t is finite, because the acceptors have abcast only a finite number of messages. Observe that all instances $message_t$ that correspond to inactive timeframes t are identical: all acceptors propose $\emptyset$ and the decision vector is $[\emptyset, \ldots, \emptyset]$. Therefore, at any given time, the number of different instances $message_t$ is finite, so the method from Section 5.3 can be used to emulate them with finite resources.

As an example, consider a run with two acceptors, in which acceptor a_1 abcasts message m_1 at time t_1, and acceptor a_2 abcasts message m_2 at time $t_2 > t_1$. The following table shows actions executed by processes in the timeframes from intervals (t_0, t_1), $[t_1, t_1]$ (denoted from now on as $[t_1]$), (t_1, t_2) and $[t_2]$.

timeframes	acceptor a_1	acceptor a_2	learners
(t_0, t_1)	$propose(\emptyset)$	$propose(\emptyset)$	$decision([\emptyset, \emptyset])$
$[t_1]$	$propose(\{m_1\})$	$propose(\emptyset)$	$decision([\{m_1\}, \emptyset])$
(t_1, t_2)	$propose(\emptyset)$	$propose(\emptyset)$	$decision([\emptyset, \emptyset])$
$[t_2]$	$propose(\emptyset)$	$propose(\{m_2\})$	$decision([\emptyset, \{m_2\}])$

All timeframes in (t_0, t_1) and (t_1, t_2) are inactive, so both acceptors propose $\emptyset$ and the decision vector is $[\emptyset, \emptyset]$. In timeframe t_1, acceptor a_1 proposes $\{m_1\}$, acceptor a_2 proposes $\emptyset$, so the decision vector is $[\{m_1\}, \emptyset]$. Similarly, in timeframe t_2, acceptor a_1 proposes $\emptyset$, acceptor a_2 proposes $\{m_2\}$, and the decision vector is $[\emptyset, \{m_2\}]$. As a result, all correct learners deliver m_1 followed by m_2. In this example, the infinite number of virtual instances $messages_t$ with $t_0 \leq t \leq t_2$ can be emulated by four physical instances of Interactive Consistency corresponding to intervals (t_0, t_1), $[t_1]$, (t_1, t_2), and $[t_2]$. Representing intervals was discussed in Section 5.3.2.

In the basic algorithm from Section 5.4.2 with $\delta = 0$, acceptor a_1 would execute $messages_t.propose(\emptyset)$ for each of the infinitely many timeframes $t \in (t_0, t_1)$. Each of these executions $messages_t.propose(\emptyset)$ would take place at a different time t. On the other hand, the method described in the previous paragraph processes all instances $messages_t$ with $t \in (t_0, t_1)$ in one step, so it requires all executions $messages_t.propose(\emptyset)$ to take place at the same time. Fortunately, all timeframes $t \in (t_0, t_1)$ are inactive; no messages are sent, so all executions $messages_t.propose(\emptyset)$ can be delayed until the next active time, t_1.

Infinite loops

In the basic algorithm in Figure 5.13, both broadcasting and delivery tasks contain loops that enumerate all timeframes. If $\delta > 0$, the set of all timeframes is countable, and we can easily enumerate them: t_0, $t_0 + \delta$, $t_0 + 2\delta$, etc. With $\delta = 0$, the set of all timeframes is no longer countable, so it cannot be enumerated.

Fortunately, enumerating all timeframes, one by one, is not necessary. All $message_t$ instances corresponding to inactive timeframes are identical; all acceptors propose $\emptyset$ and the decision vector is $[\emptyset, \ldots, \emptyset]$. Therefore blocks of contiguous inactive timeframes, such as (t_0, t_1) or (t_1, t_2) in our example, can be processed in one step.

This observation leads to the advanced version of our Atomic Broadcast algorithm, shown in Figure 5.14. Consider the delivery task first and compare it with the delivery task in Figure 5.13. In lines 15–22, a simple loop enumerating all timeframes in the basic algorithm has been replaced by a more complicated structure that iterates over active timeframes. The learner maintains a variable t_{old}, initially t_0. Every iteration of the loop assumes that all timeframes in $(t_0, t_{\text{old}}]$ have already been dealt with.

The loop first waits until there is $t > t_{\text{old}}$ such that all instances $messages_{t'}$ with $t' \in (t_{\text{old}}, t)$ have decided on $[\emptyset, \ldots, \emptyset]$. All entries in this vector are $\emptyset$, therefore no messages are delivered and no action needs to be taken for any $messages_{t'}$ with $t' \in (t_{\text{old}}, t)$. Then, the learner waits until instance $messages_t$ has itself decided on some vector $[\mathcal{M}_1, \ldots, \mathcal{M}_n]$. All messages in this vector are delivered, exactly as in the basic algorithm in Figure 5.13. Finally, the loop updates t_{old} to t, to indicate that all timeframes in $(t_0, t]$ have been dealt with.

As in the basic algorithm, whenever an acceptor abcasts a message m, it inserts m into the set $\mathcal{M}$. In addition, the acceptor broadcasts a message informing all acceptors, including itself, that the timeframe corresponding to the current real time *current-time* should be considered active (lines 1–3).

The transformation the broadcasting loop from the basic algorithm is similar to that of the delivery loop. Instead of enumerating all timeframes, we only enumerate (some) active ones. The acceptor maintains variable t_{old}, which indicates that the acceptor has proposed in all instances $messages_t$ with $t \in (t_0, t_{\text{old}}]$. Each iteration of the loop starts

```
1   when an acceptor executes abcast(m) do
2     insert m into M
3     broadcast "active t" with t = current-time
4   task broadcasting at acceptors is
5     t_old ← t_0
6     repeat forever
7       M ← ∅
8       wait until received "active t" with some t > t_old
9       wait until current-time ≥ t
10      for all t' ∈ (t_old, t) do
11        messages_t'.propose(∅)
12      messages_t.propose(M)
13      t_old ← t
14      increase current-time
15  task delivery at learners is
16    t_old ← t_0
17    repeat forever
18      wait until messages_t'.decision([∅, ..., ∅]) for all t' ∈ (t_old, t) for some t > t_old
19      wait until messages_t.decision([M_1, ..., M_n])
20      for i = 1, 2, ..., n do
21        adeliver all undelivered messages in M_i in some deterministic order
22      t_old ← t
23  task retransmission at an acceptor is
24    periodically do
25      for all seen messages m do
26        abcast(m)
```

Figure 5.14: Full version of the two-step Atomic Broadcast algorithm for closed groups.

with emptying the set $\mathcal{M}$ and waits for a message "active t" with some $t > t_{\text{old}}$, possibly sent by the acceptor itself. Then, the acceptor waits until the current time is at least t. This is done to eliminate anomalies caused by messages "active t" from the future, sent by acceptors whose clock skew exceeds the maximum message transmission time d. If the clocks are synchronized, the condition *current-time* $\geq t$ always holds.

After completing the two **wait** instructions, the acceptor proposes $\emptyset$ to all instances $message_{t'}$ with $t' \in (t_{\text{old}}, t)$ and the current value of $\mathcal{M}$ to $messages_t$. If the message "active t" comes from the acceptor itself, then the set $\mathcal{M}$ will contain the message that has just been abcast by this acceptor. If "active t" comes from another acceptor, then $\mathcal{M}$ is usually empty. The loop ends with setting t_{old} to t, to indicate that the acceptor proposed in all instances $messages_t$ with $t \in (t_0, t]$.

The statement "increase *current-time*" explicitly requires the real time clock to progress at the end of each iteration. This ensures that no more messages will be abcast in the current timeframe. Without such a progress condition, synchronized clocks could be triv-

ially implemented by reporting the same time 0 at all acceptors all the time. Achieving a delivery latency of two steps under such conditions would violate Theorem 5.5.1.

Latency

To simplify the reasoning, we will assume that $t_0 = -\infty$. Consider a good run in which only one message m is abcast, by acceptor a_1 at time t. Acceptor a_1 adds m to its set $\mathcal{M}$ and broadcasts "active t". Since self-addressed messages incur no delay, the broadcasting task at acceptor a_1 receives "active t" at time t. No messages have been sent so far, so $t_{\text{old}} = -\infty$, and the current time is t, so the conditions of both **wait** instructions hold. The acceptor proposes $\emptyset$ to all instances $messages_{t'}$ with $t' \in (-\infty, t)$. It also proposes $\mathcal{M} = \{m\}$ to $messages_t$.

All other acceptors receive "active t" at time $t+d$. They did not abcast any messages, so their sets $\mathcal{M}$ are empty. As a result, after receiving "active t" at time $t + d$, they propose $\emptyset$ to all instances $messages_{t'}$ with $t' \in (-\infty, t]$.

Consider any instance $message_{t'}$ with $t' \in (-\infty, t]$. All acceptors (except possibly a_1) proposed $\emptyset$ at time $t+d$. Acceptor a_1 proposed at time t. Since $\emptyset = \text{ABORT}$ is the privileged value, the Latency property of Interactive Consistency from Section 4.4.6 implies that any instance $message_{t'}$ with $t' \in (-\infty, t]$ will decide by time $t + 2d$.

Consider the delivery loop at a learner. It starts by waiting for all instances $messages_{t'}$ with $t' \in (-\infty, t)$ to decide on $[\emptyset, \ldots, \emptyset]$ and the instance $messages_t$ to decide on some vector $[\mathcal{M}_1, \ldots, \mathcal{M}_n]$. In our case, these two conditions will be met by time $t+2d$, with the vector $[\mathcal{M}_1, \ldots, \mathcal{M}_n]$ being $[\{m\}, \emptyset, \ldots, \emptyset]$. As a result, the learner will deliver message m by time $t + 2d$, achieving a latency of $2d$. It turns out that this latency is achieved in all good runs, regardless of the number of messages being abcast at the same time:

> **Latency.** In good runs with synchronized clocks, every abcast message is delivered by all correct learners within two communication steps.

Termination properties

None of the changes we have made so far to the basic algorithm from Section 5.4.2 affect the liveness of the algorithm; messages sent by correct acceptors in untimely runs might still not be delivered. This is because, in such runs, Interactive Consistency instances $message_t$ can decide on $[\emptyset, \ldots, \emptyset]$, regardless of the proposals. Some modifications must be made to the basic algorithm to avoid this problem.

The retransmission task in lines 23–26 periodically re-abcasts all seen messages. The failure detector $\Diamond S$ guarantees that there is a correct acceptor that is eventually never suspected. This acceptor will see all messages abcast by correct acceptors, and will keep abcasting them until they get delivered (Termination Validity). As explained in Section 5.1.4, re-abcasting all seen messages also ensures Termination Agreement.

We assume that Interactive Consistency instances $message_t$ are implemented as in Section 4.4.6: each $message_t$ consists of Individual Consensus instances $message_t.Ind_i$ with $i = 1, \ldots, n$. In this algorithm, we say that an acceptor sees a message m iff it sees a set $\mathcal{M} \ni m$ in some of the Individual Consensus instances $message_t.Ind_i$.

5.5 Lower bounds

In this section, we prove several impossibility results for Atomic Broadcast protocols that can tolerate one faulty acceptor ($f > 0$). Unless stated otherwise, the results apply to the closed-group model, and therefore also to the more general open-group model.

5.5.1 Two steps are required in any run

Any Atomic/Generic Broadcast protocol requires at least two communication steps in *any* run. To obtain a contradiction, assume that an acceptor a atomically delivers its own message m faster than in two communication steps, that is, before getting any feedback from other processes. If acceptor a crashes immediately after delivering m and all messages from a to other processes are lost, then no other learner will ever receive or deliver m. This would violate Termination Agreement. Pedone and Schiper [100] proved a similar result.

5.5.2 Latency below three steps requires synchronized clocks

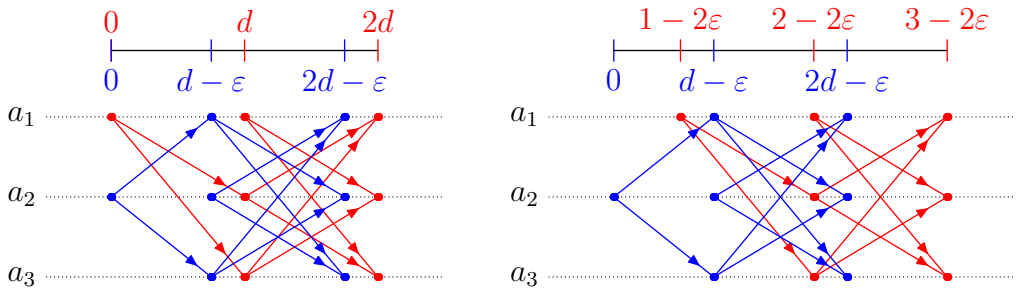
Section 5.4 shows a closed-group algorithm that guarantees the latency of $2d$ in all good runs in which all acceptors are equipped with perfectly synchronized clocks. In this section, we will show that the synchronized clocks assumption is necessary.

Theorem 5.5.1. *Only Atomic Broadcast algorithms that use synchronized clocks can guarantee a latency of less than $3d$ in all good runs.*

Proof. To obtain a contradiction, assume the existence of an Atomic Broadcast algorithm that does not use synchronized clocks but in good runs delivers all messages within $K < 3$ communication steps. We will show that such an assumption leads to a contradiction.

Consider a family of good runs $r(k)$ for $k = 0, 1, \ldots, n$, in which acceptors a_1 and a_2 abcast two messages m_1 and m_2, respectively, at time 0, and no other messages are abcast. All processes are correct and almost all messages have the latency of d. The only exceptions are some messages sent at time 0: those from acceptor a_1 to acceptors $a_1, \ldots, a_k$, and those from a_2 to $a_{k+1}, \ldots, a_n$. These messages have the latency of $d - \varepsilon$, for some small $\varepsilon > 0$ which we will define later. All other messages have a latency of d.

We will first prove that, for any $k = 1, \ldots, n$, runs $r(k)$ and $r(k-1)$ deliver messages m_1 and m_2 in the same order. For each $i \in \{k-1, k\}$, consider a run $r_k(i)$, which is



(a) $r(0)$ (b) r'

Figure 5.15: Runs $r(0)$ and r' used in the proof.

identical to $r(i)$, except that all messages sent by acceptor a_k to other processes at time $d-\varepsilon$ or later are lost. Runs $r(i)$ and $r_k(i)$ are identical until time $d-\varepsilon$. Since all messages sent after time 0 have latencies d, runs $r(i)$ and $r_k(i)$ are indistinguishable to acceptors other than a_k until time $2d-\varepsilon$, and to a_k itself until $3d-\varepsilon$. Since $K < 3$, we have $3d-\varepsilon > Kd$ for sufficiently small ε. This means that acceptor a_k delivers the same message first in both runs $r(i)$ and $r_k(i)$. Agreement and Termination Agreement imply that all correct acceptors deliver the same message first in $r(i)$ and $r_k(i)$.

To show that the runs $r(k)$ and $r(k-1)$ deliver the same message first, it is then sufficient to show the same for runs $r_k(k)$ and $r_k(k-1)$. Runs $r_k(k)$ and $r_k(k-1)$ differ only in the delays of messages sent by acceptors a_1 and a_2 to acceptor a_k at time 0. However, these messages arrive at a_k at time $d-\varepsilon$ or later, and from that time all messages from a_k to other processes are lost. Therefore, these two runs are indistinguishable to any correct acceptor $a \neq a_k$, who delivers the same message first in both of them.

We have shown that, for any $k = 1, \ldots, n$, runs $r(k)$ and $r(k-1)$ deliver messages m_1 and m_2 in the same order. Simple induction on k shows that the same is true for runs $r(0)$ and $r(n)$. Without loss of generality, assume m_1 is delivered first in both runs and focus on run $r(0)$. (The other case, in which m_2 is delivered first, is analogous and requires considering run $r(n)$.) In run $r(0)$, shown in Figure 5.15, all message latencies are d, except for those sent by acceptor a_2 at time 0; these have the latency of $d-\varepsilon$.

Consider a good run r' which is identical to $r(0)$ except that acceptor a_1 abcasts m_1 at time $d-2\varepsilon$ instead of at time 0. Figure 5.15 shows that runs r' and $r(0)$ are causally identical, so acceptors without synchronized clocks cannot distinguish them. As a result, the same message (m_1) is delivered first in both of them.

Section 5.5.1 proved that m_1 cannot be delivered faster than in two communication steps, that is, before $3d-2\varepsilon$ in run r'. Since message m_2, abcast at time 0, is delivered after m_1, it cannot be delivered before time $3d-2\varepsilon$ either. Since $K < 3$, this is bigger than Kd for sufficiently small ε, which contradicts the assumption that, in good runs, all messages are delivered in $K < 3$ steps. □

5.5.3 Dealing with faulty proposers requires three steps

Theorem 5.5.2. *Consider two proposers p_1 and p_2 (possibly acceptors). No Atomic Broadcast algorithm can guarantee a latency of less than three communication steps in all timely runs with all processes correct, except for possibly one of p_1 and p_2.*

We deliberately allow the proposers p_1 and p_2 to be acceptors, to address the open-group and closed-group models at the same time. In the former model, the modified Chandra-Toueg algorithm in Section 5.1.3 guarantees a latency of $3d$ in all stable runs. Figure 5.2 showed that none of the algorithms for open groups presented in this chapter can guarantee a latency strictly smaller than this. Using Theorem 5.5.2 with proposers p_1 and p_2 not being acceptors, we can show that this is inevitable:

Theorem 5.5.3. *No open-group Atomic Broadcast algorithm can guarantee a latency of less than three communication step in all good runs.*

Note that this result is specific to open-group algorithms; Section 5.4 shows a solution for closed groups that guarantees a latency of $2d$ in all good runs. This is one step better than open-group solutions, however, with stronger correctness assumptions. Open-group algorithms from this chapter guarantee three-step delivery in all *stable* runs. On the other hand, the algorithm in Section 5.4 guarantees two-step delivery, but only in *good* runs. Using Theorem 5.5.2 with p_1 and p_2 being any two acceptors other than the leader, we can show that this requirement is inevitable:

Theorem 5.5.4. *No closed-group Atomic Broadcast algorithm can guarantee a latency of less than three communication step in all timely runs with at most one non-leader acceptor being faulty.*

We will prove Theorem 5.5.2 by considering runs in which proposer p_1 abcasts message m_1, and proposer p_2 message m_2, both at time 0. First, consider a run r, in which all processes are correct and all messages have latencies d. Without loss of generality, we can assume that message m_1 is delivered first in this run. Otherwise, we can simply exchange the roles of proposers p_1 and p_2.

Consider a family of runs $r(k)$ with $k = 0, \ldots, n$. Run $r(k)$ is identical to r, except that proposer p_1 is faulty and crashes at some time in the future, say $3d$. All messages sent by proposer p_1 to acceptors $a_1, \ldots, a_k$ have the latency $3d$ instead of d; the latencies of messages from p_1 to other acceptors remain d. Proposer p_2 is correct. All messages between correct processes a latency of d, so all correct learners deliver message m_2 before time $3d$.

The only difference between run r and $r(0)$ is the correctness of proposer p_1; these runs are indistinguishable to any process before time $3d$, so $r(0)$ delivers message m_1 first. We will later show that, for any k, runs $r(k-1)$ and $r(k)$ deliver the same message

first. By simple induction, run $r(n)$ delivers message m_1 before m_2. We have previously shown that, in run $r(n)$, all correct learners deliver m_2 before time $3d$, so m_1 must also be delivered before that time. However, no process other than p_1 knows about m_1 before time $3d$, so it is impossible for all correct acceptors to deliver m_1 before that time in run $r(n)$. This contradiction proves the assertion.

Runs $r(k-1)$ and $r(k)$ deliver the same message first

For any $i \in \{k-1, k\}$ consider a run $r_k(i)$ which is identical to $r(i)$, except that acceptor a_k is faulty and crashes at time $3d$. All messages from a_k to other processes sent at time d or later are lost. Also, in run $r_k(i)$, each proposer p_j with $j \in \{1, 2\}$ is correct, unless $p_j = a_k$. Runs $r(i)$ and $r_k(i)$ are identical until time d, so they are indistinguishable to acceptors other than a_k until time $2d$. For acceptor a_k, these two runs seem the same until time $3d$.

We have already shown that, in run $r(i)$, all correct learners, in particular acceptor a_k, deliver at least one message (m_2) before time $3d$. Since a_k cannot distinguish runs $r(i)$ and $r_k(i)$ before that time, it delivers the same message first in both runs. Agreement and Termination Agreement imply that all correct acceptors deliver the same message first in runs $r(i)$ and $r_k(i)$.

To show that runs $r(k-1)$ and $r(k)$ deliver the same message first, it is now sufficient to show the same for runs $r_k(k-1)$ and $r_k(k)$. Before time $3d$, these runs differ only in the latency of messages from proposer p_1 to acceptor a_k. These two runs are indistinguishable to a_k until time d, but from that time on, all messages from a_k to other processes are lost. As a result, other acceptors cannot distinguish runs $r_k(k-1)$ and $r_k(k)$ before time $3d$, so they must deliver the same message first in both runs. This way, we have proved that runs $r(k-1)$ and $r(k)$ deliver the same message first.

5.6 Summary

In this chapter, we have investigated the Atomic Broadcast problem in distributed systems. Our analysis focused on minimizing the latency in good and stable runs. We started with modifying the Atomic Broadcast protocol proposed by Chandra and Toueg [16] to achieve the latency of three communication steps in such runs. In the rest of the chapter, we presented several Atomic/Generic algorithms that, under some conditions, deliver messages in two communication steps. Finally, we proved several new lower bounds for the Atomic Broadcast problem, including that a latency of less than two steps is impossible in any run.

This chapter presented several broadcast algorithms. In Section 5.1.3, we showed a modified Chandra-Toueg Atomic Broadcast algorithm [16]. It achieves a latency of $3d$ in

all stable runs and requires $n > 2f$. Both figures are optimal. Theorem 5.5.3 showed that no Atomic Broadcast algorithm for open groups can guarantee a latency lower than $3d$ in all good runs. Chandra and Toueg [16] proved that $n > 2f$ is necessary as well.

Section 5.1.7 presented an Atomic Broadcast protocol that achieves a stable-run latency of $2d$ if all correct acceptors receive abcast messages in the same order. Section 5.2 presented a Generic Broadcast protocol that achieves this latency if only *conflicting* messages are received in the same order. In contrast to algorithms proposed previously in the literature [5, 101, 102], which require four steps in some stable runs, our solutions guarantee the optimum latency of three steps. All these algorithms require $n > 3f$, which is necessary for two-step delivery [100].

Further improvements in latency can be achieved by restricting the set of proposers. Section 5.4 presented an Atomic Broadcast protocol for closed groups, which guarantees a latency of two steps in all good runs. As shown by Theorem 5.5.3, this is impossible in open groups. To achieve this latency, the algorithm requires perfectly synchronized clocks and all acceptors correct. Theorems 5.5.1 and 5.5.4 showed that both these assumptions are necessary.

The algorithms presented in this chapter are optimal in terms of latency, but can require large amounts of resources such as memory, threads, or processing. They all use a number of instances of Consensus or similar abstractions. Chapter 4 shows that these abstractions consist of sequences of rounds themselves. The complexity of broadcast algorithms can probably be reduced by using these rounds, implemented – for example – as OTC instances, directly.

Some of the algorithms generate a number of messages that are never used. For example, the Consensus algorithm from Section 5.1.8 does not use instance 2 in ordered runs. Further research is required to determine whether this problem can be avoided in latency-optimal protocols.

Chapter 6

Conclusion

In 1985, Fischer, Lynch, and Paterson [40] proved that Consensus is unsolvable in asynchronous distributed systems in which one process can crash. This classic result prompted a number of researchers to investigate model extensions that would make Consensus and other agreement problems solvable [7, 16, 23, 24, 37, 104]. This main focus on *solvability* rather than *efficiency* contributed to the reputation of fault-tolerant agreement protocols as being inherently inefficient and impractical. Despite the efforts by Guerraoui and Schiper [54] to clarify this widespread misinterpretation of the FLP impossibility theorem, the gap between practical and theoretical fault tolerance still exists.

The goal of the presented research was to understand and bridge this gap by investigating efficient solutions to a number of agreement problems, with the efficiency criterion being latency: the number of communication steps required by the algorithm. In comparison to the solvability aspect, our knowledge of the efficiency of agreement protocols seems rather limited and uneven. For example, it is known that no Consensus algorithm can guarantee latency of less than two steps in all good runs [19, 66, 74]. On the other hand, Consensus is sometimes solvable in one step [13], but no one has so far examined the exact circumstances in which this is possible. In Atomic Broadcast, the situation is opposite: the circumstances under which the low, two-step latency is possible have been investigated [5, 99, 100, 102], but the general lower bound of three steps has never been shown. One of the aims of this investigation was to complete the picture by proving the "missing" lower bounds.

Parallel to proving lower bounds is the effort to design more efficient agreement protocols. Many latency-efficient Consensus algorithms [35, 63, 65, 73, 112] and agreement frameworks [10, 11, 12, 51, 65, 92] for the crash-stop model have been investigated in the literature. Recently, several Byzantine Consensus algorithms have been proposed [15, 36, 81, 87, 121], however, no agreement framework tolerating malicious participants has appeared in the literature yet. One of my goals was to design such a framework, capable of implementing not only Consensus but also other agreement abstractions. To the best of my knowledge, no such framework had been proposed yet, even for the crash-stop

model.

In order to build this framework, it was necessary to identify a common pattern in agreement protocols, and use it to design a new lightweight agreement abstraction. This abstraction could then later be used to reconstruct existing protocols and design new ones, without any latency overhead. As a side-effect, the simplicity requirement on the abstraction reduced the solution space, thereby making it possible to perform automatic correctness verification. From here, it would not be difficult to search the solution space to automatically discover new agreement algorithms.

Naturally, the first step to accomplish all the tasks described above was to identify the common pattern in various agreement protocols. Chapter 2 first showed that the round-based structure can be viewed as such, and then introduced the notion of Optimistically Terminating Consensus (OTC), a lightweight agreement abstraction that formalizes the notion of a round. As opposed to Consensus, OTC guarantees Termination only if all correct acceptors propose the same value. In exchange, it ensures stronger Validity and Agreement properties, which make it possible for the next round to take over if the current one does not terminate for some reason.

The OTC abstraction is easy to implement, even with malicious acceptors; the learners decide on a given value if a sufficient number of acceptors report to have proposed it. This simple one-step implementation, called Generic Agreement in Section 2.3, is sufficient to match the latency of a large number of Consensus algorithms for the crash-stop model [13, 16, 63, 73, 80, 112] as well as the Byzantine model [41, 71, 87]. Combining several instances of one-step Generic Agreement leads to new OTC implementations, which match the latency of other Consensus algorithms [15, 36, 121] and allow us to construct new ones. Chapter 2 proves that the latencies of all these OTC implementations are optimal.

Chapter 3 showed how OTC algorithm candidates can be tested for correctness automatically. In this test, a positive result proves that the given algorithm satisfies all properties required by the OTC abstraction. A negative result shows a state in which one of the OTC properties is violated. Negative results are useful as well: they help to understand why the tested OTC algorithm candidate is incorrect, and can often be generalized to impossibility theorems.

To implement automatic correctness testing, I developed a theory for reasoning about states that evolve according to an event-based execution model. For given Optimistic Termination requirements, we construct the weakest possible algorithm satisfying OTC properties Possibility and Integrity, and then test whether it satisfies the remaining two properties as well. Generating OTC algorithm candidates and using the above correctness-testing method allows us to automatically discover new OTC algorithms, thereby skipping the manual algorithm design process altogether. In other words, instead of using automatic correctness testing to verify individual OTC algorithms, one can just specify a set of requirements and use a computer to search the solution space for OTC algorithms that

satisfy them.

Both manually and automatically generated OTC algorithms can be used to solve Consensus and other agreement problems in a modular way. To achieve this, Chapter 4 formalized the Coordinated Consensus problem, and presented OTC-based algorithms for both benign and malicious settings. This algorithm served as a base for developing efficient solutions to other agreement problems: Consensus, Individual Consensus, Atomic Commitment, and Interactive Consistency, all for both failure models. By using OTC implementations from Chapters 2 and 3, I was able to provide implementations that match or improve the latency of known ad-hoc solutions.

In comparison to other agreement frameworks [10, 11, 12, 51, 65, 92], this approach makes it possible to reconstruct the highest number of known algorithms as well as to construct new ones. Firstly, this is because no other agreement framework tolerates malicious processes. Secondly, this is because OTC, the main unit in our solution, is relatively small; it encompasses only a single round, as opposed to the whole algorithm in other frameworks. This and the independence of OTC instances used by different rounds allow for high flexibility in designing Consensus algorithms; the parameters of every round, such as the coordinator, the type of the OTC, or the set of acceptors used, can be all set independently.

Because of the dynamic nature of Atomic Broadcast, we considered it independently from other agreement abstractions. Chapter 5 started by modifying the algorithm proposed by Chandra and Toueg [16] to achieve a latency of three communication steps in stable runs. Then, we presented several broadcast algorithms that, under some conditions, deliver messages in two communication steps, and showed that their latencies are optimal.

Firstly, we presented a modified Chandra-Toueg Atomic Broadcast algorithm [16], which achieves a three-step latency in stable runs. Then, we improved it to guarantee a latency of two steps if all correct acceptors receive abcast messages in the same order. Optimistic Generic Broadcast went even further by ensuring the two-step latency if only *conflicting* messages are received in the same order. As opposed to algorithms proposed previously in the literature [5, 101, 102], which require four steps in some stable runs, my solutions guarantee the optimum latency of three steps in *all* stable runs. Finally, I showed that, in closed groups, this latency can be reduced to two steps.

All in all, this thesis investigated implementations of various agreement abstractions from the point of view of their latency, consolidating existing results as well as presenting new algorithms and lower bounds. To achieve this, I identified a common pattern in existing algorithms and formalized it into the OTC abstraction. The investigation of this abstraction can be summarized in two main conclusions: (i) latency-optimal fault-tolerant algorithms can be constructed in a modular way even in Byzantine settings, and (ii) auto-generation of efficient agreement protocols is possible and practical.

Future work

The OTC abstraction proposed in this thesis has been designed specifically to tolerate malicious behaviour, so the agreement protocols implemented on top of it are capable of that as well. The only exception is Atomic Broadcast; since we investigated this abstraction in considerably more detail than others, the scope of this study was limited to the crash-stop model. The investigation of efficient Atomic Broadcast protocols and lower bounds for Byzantine settings is a promising direction of future research.

All results derived here measure the efficiency of distributed algorithms as the number of communication steps required. We ignore other factors such as processor usage, memory usage, network load, the number of messages transmitted, etc. Future research is required to evaluate the impact of these factors and to examine their compatibility with the efficiency measure considered in this thesis.

Our communication model assumes reliable channels, which never lose messages between correct processes. In practice, however, communication channels are unreliable and such messages can be lost. Reliable channels can be implemented over unreliable ones without any latency overhead by periodic retransmission [9], however, this requires the sender to remember all unconfirmed messages. Because of the high memory requirements of this solution, it would be preferable to embed support for unreliable channels in agreement protocols directly. Inspiration for future research in this direction comes from existing omission-resistant protocols such as Paxos [73] or Consensus with stubborn channels [57].

The channel communication model is by no means the only possible one. It would be interesting to extend this work to other communication models, such as shared memory, as well as the passive and active disk models [3]. Recent steps in this direction include Byzantine Disk Paxos [1], which implements Consensus with possibly malicious passive disks, and Alpha [52], an agreement framework that can use all the communication models described above, but only in the crash-stop model. The ultimate goal would be to unify these two approaches, and create a communication-model-independent extension of the OTC abstraction.

As explained before, the OTC abstraction has a number of advantages over other frameworks [10, 11, 12, 51, 65, 92], with automatic testing and discovery being probably the most interesting. Although well established for security protocols [14, 22, 84, 88, 96], automatic reasoning has not yet been widely used in the field of distributed algorithms. This work demonstrates that searching the entire state space is not necessary, which makes this approach not only theoretically possible but also practical. I hope that similar techniques could be developed for other distributed problems, such as Mutual Exclusion [8], Atomic Broadcast [26], or various forms of Group Communication [21].

Bibliography

[1] Ittai Abraham, Gregory V. Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine Disk Paxos: Optimal resilience with Byzantine shared memory. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, St. John's, Newfoundland, Canada, July 2004.

[2] Divyakant Agrawal, Gustavo Alonso, Amr El Abbadi, and Ioana Stanoi. Exploiting Atomic Broadcast in replicated databases (extended abstract). In *European Conference on Parallel Processing*, pages 496–503, 1997.

[3] Marcos K. Aguilera, Burkhard Englert, and Eli Gafni. On using network attached disks as shared memory. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, pages 315–324, New York, NY, USA, 2003. ACM Press.

[4] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and Consensus in the crash-recovery model. In *International Symposium on Distributed Computing*, pages 231–245, 1998.

[5] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Thrifty Generic Broadcast. *Lecture Notes in Computer Science*, 1914:268–282, 2000.

[6] Bowen Alpern and Fred B. Schneider. Recognizing Safety and Liveness. *Distributed Comp.*, 2:117–126, 1987.

[7] James Aspnes. Randomized protocols for asynchronous Consensus. *Distributed Computing*, 16(2-3):165–175, 2003.

[8] Yoah Bar-David and Gadi Taubenfeld. Automatic discovery of Mutual Exclusion algorithms. In *Proceedings of the 17th International Symposium on Distributed Computing*, 2003.

[9] Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, pages 105–122, 1996.

[10] Romain Boichat, Partha Dutta, Svend Frolund, and Rachid Guerraoui. Deconstructing Paxos. Technical Report DSC/2002/032, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, January 2001.

[11] Romain Boichat, Partha Dutta, Svend Frolund, and Rachid Guerraoui. Deconstructing Paxos. *ACM SIGACT News*, 34, 2003.

[12] Romain Boichat, Partha Dutta, Svend Frolund, and Rachid Guerraoui. Reconstructing Paxos. *ACM SIGACT News*, 34, 2003.

[13] Francisco Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in one communication step. *Lecture Notes in Computer Science*, 2127:42–50, 2001.

[14] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990. ISSN 0734-2071.

[15] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186, New Orleans, Louisiana, February 1999. USENIX Association.

[16] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[17] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving Consensus. In Maurice Herlihy, editor, *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 147–158, Vancouver, BC, Canada, 1992. ACM Press.

[18] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving Consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[19] Bernadette Charron-Bost and André Schiper. Uniform Consensus is harder than Consensus. Technical Report DSC/2000/028, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, May 2000.

[20] Bernadette Charron-Bost, Sam Toueg, and Anindya Basu. Revisiting safety and liveness in the context of failures. In *Proceedings of the 11th International Conference on Concurrency Theory*, pages 552–565, London, UK, 2000. Springer-Verlag.

[21] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001. URL `citeseer.ist.psu.edu/chockler01group.html`.

[22] Edmund M. Clarke, Somesh Jha, and Will Marrero. Verifying security protocols with Brutus. *ACM Transactions on Software Engineering and Methodology*, 9(4): 443–487, 2000. ISSN 1049-331X.

[23] Miguel Correia, Nuno Ferreira Neves, Lau Cheuk Lung, and Paulo Veríssimo. Low complexity Byzantine-resilient Consensus. DI/FCUL TR 03–25, Department of Informatics, University of Lisbon, August 2003.

[24] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.

[25] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic Broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings 15th International Symposium on Fault-Tolerant Computing*, pages 200–206, Ann Arbor, MI, USA, 1985. IEEE Computer Society Press.

[26] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[27] Carole Delporte-Gallet, Hugues Fauconnier, Jean-Michel Hélary, and Michel Raynal. Early stopping in global data computation. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 258–258. ACM Press, 2002.

[28] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Shared memory vs message passing. Technical report, LPD, December 2003.

[29] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 338–346. ACM Press, 2004.

[30] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[31] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed Consensus. *Journal of the ACM*, 34(1):77–97, 1987.

[32] Danny Dolev, Ruediger Reischuk, and H. Raymond Strong. Early stopping in Byzantine agreement. *Journal of the ACM*, 37(4):720–741, 1990.

[33] Assia Doudou and André Schiper. Muteness detectors for Consensus with Byzantine processes. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 315–316, New York, June 1998.

[34] Assia Doudou, Benoît Garbinato, and Rachid Guerraoui. Encapsulating failure detection: From crash to Byzantine failures. In *Proceedings of the 7th International Conference on Reliable Software Technologies*, pages 24–50, June 2002.

[35] Partha Dutta and Rachid Guerraoui. Fast indulgent Consensus with zero degradation. In Fabrizio Grandoni and Pascale Thévenod-Fosse, editors, *Proceedings of 4th European Dependable Computing Conference*, volume 2485 of *Lecture Notes in Computer Science*, pages 191–208. Springer, 2002.

[36] Partha Dutta, Rachid Guerraoui, and Marko Vukolic. Asynchronous Byzantine Consensus: Complexity, resilience and authentication. Technical Report 200479, EPFL, September 2004.

[37] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[38] Patrick Th. Eugster, P.A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. Technical Report DSC ID:200104, EPFL, January 2001.

[39] Paul Ezhilchelvan, Doug Palmer, and Michel Raynal. An optimal Atomic Broadcast protocol and an implementation framework. In *Proceedings of the 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003)*, pages 32–41, January 2003.

[40] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed Consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[41] Roy Friedman, Achour Mostéfaoui, and Michel Raynal. Simple and efficient oracle-based Consensus protocols for asynchronous Byzantine systems. In *Proceedings of 23rd IEEE International Symposium on Reliable Distributed Systems*, pages 228–237, October 2004.

[42] Eli Gafni and Leslie Lamport. Disk Paxos. In *International Symposium on Distributed Computing*, pages 330–344, 2000.

[43] Stephen Garland and John Guttag. A guide to lp, the larch prover. Technical report, DEC Systems Research Center, 1991.

[44] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[45] Jim Gray and Leslie Lamport. Consensus on Transaction Commit. Technical Report MSR-TR-2003-96, Microsoft, January 2004.

[46] Rachid Guerraoui. Indulgent algorithms. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 289–298, NY, July 2000. ACM Press.

[47] Rachid Guerraoui. Non-Blocking Atomic Commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15(1):17–25, 2002.

[48] Rachid Guerraoui. Revisiting the relationship between Non-Blocking Atomic Commitment and Consensus. In *Proceedings of the* 9^{th} *International Workshop on Distributed Algorithms (WDAG-9)*, number 972 in Lecture Notes in Computer Science, pages 87–100, Le Mont-St-Michel, France, September 1995. Springer-Verlag.

[49] Rachid Guerraoui and Petr Kouznetsov. On the weakest failure detector for Non-Blocking Atomic Commit. Technical report, School of Computer and Communication Sciences, Swiss Institute of Technology in Lausanne, 2002.

[50] Rachid Guerraoui and Petr Kouznetsov. Finally the weakest failure detector for Non-Blocking Atomic Commit. Technical report, LPD, December 2003.

[51] Rachid Guerraoui and Michel Raynal. The information structure of indulgent Consensus. Technical Report PI-1531, IRISA, April, 2003.

[52] Rachid Guerraoui and Michel Raynal. The alpha and omega of asynchronous Consensus. Technical Report PI-1676, IRISA, January 2005.

[53] Rachid Guerraoui and André Schiper. The decentralized Non-Blocking Atomic Commitment protocol. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing (SPDP-7)*, pages 2–9, San Antonio, Texas, USA, 1995.

[54] Rachid Guerraoui and André Schiper. Consensus: the big misunderstanding. In *Proceedings of the 6th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-6)*, pages 183–188, Tunis, Tunisia, 1997. IEEE Computer Society Press.

[55] Rachid Guerraoui, Mikel Larrea, and André Schiper. Non-Blocking Atomic Commitment with an unreliable failure detector. In *Proceedings of the 14th Symposium on Reliable Distributed Systems*, pages 41–50, Bad Neuenahr, Germany, 1995.

[56] Rachid Guerraoui, Mikel Larrea, and André Schiper. Reducing the cost for Non-Blocking in Atomic Commitment. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS-16)*, pages 692–697, Hong Kong, 1996.

[57] Rachid Guerraoui, Riucarlos Oliveira, and André Schiper. Stubborn communication channels. Technical Report 98/272, Swiss Federal Institute of Technology, Switzerland, 1998.

[58] Rachid Guerraoui, Michel Hurfin, Achour Mostéfaoui, R. Oliveira, Michel Raynal, and André Schiper. Consensus in asynchronous distributed systems: A concise guided tour. In S. Shrivastava and S. Krakowiak, editors, *Advances in Distributed Systems*, number 1752 in Lecture Notes in Computer Science, pages 33–47. Spinger, 2000.

[59] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcast and related problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5, pages 97–146. ACM Press, New York, 2nd edition, 1993.

[60] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Computer Science Department, May 1994.

[61] Jean-Michel Helary, Michel Hurfin, Achour Mostéfaoui, Michel Raynal, and Frédéric Tronel. Computing global functions in asynchronous distributed systems with perfect failure detectors. *IEEE Transactions Parallel Distrib. Syst.*, 11(9):897–909, 2000.

[62] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[63] Michel Hurfin and Michel Raynal. A simple and fast asynchronous Consensus protocol based on a weak failure detector. *Distributed Computing*, 12(4):209–223, 1999.

[64] Michel Hurfin, Achour Mostéfaoui, and Michel Raynal. Consensus in asynchronous systems where processes can crash and recover. Technical Report 1207, IRISA, September 1998.

[65] Michel Hurfin, A. Mostfaoui, and Michel Raynal. A versatile family of Consensus protocols based on Chandra-Toueg's unreliable failure detectors. *IEEE Transactions Comput.*, 51(4):395–408, 2002.

[66] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant Consensus when there are no faults. *ACM SIGACT News*, 32, 2001.

[67] Idit Keidar and Sergio Rajsbaum. A simple proof of the Uniform Consensus synchronous lower bound. *Information Processing Letters*, 85(1):47–52, 2003. ISSN 0020-0190. doi: http://dx.doi.org/10.1016/S0020-0190(02)00333-2.

[68] Pertti Kellomaki. An annotated specification of the Consensus protocol of Paxos using superposition in PVS. Technical Report 36, Tampere University of Technology, Institute of Software Systems, 2004.

[69] Bettina Kemme, Fernando Pedone, Gustavo Alonso, André Schiper, and Matthias Wiesmann. Using optimistic Atomic Broadcast in transaction processing systems. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):1018–1032, July 2003.

[70] Kim Potter Kihlstrom, Louise E. Moser, and Peter M. Melliar-Smith. Solving Consensus in a Byzantine environment using an unreliable fault detector. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, pages 61–75, 1997.

[71] Klaus Kursawe. Optimistic asynchronous Byzantine agreement. Technical Report RZ 3202 (#93248), IBM Research, January 2000.

[72] Simon S. Lam and Leonard Kleinrock. Packet-switching in a multi-access broadcast channel: Dynamic control procedures. *IEEE Transactions on Communications*, 23: 891–904, September 1975.

[73] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[74] Leslie Lamport. Lower bounds on Consensus. Unpublished note, March 2002.

[75] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Professional, 2002.

[76] Leslie Lamport. Lower bounds on asynchronous Consensus. In André Schiper, Alex A. Shvartsman, Hakim Weatherspoon, and Ben Y. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 22–23. Springer, 2003.

[77] Leslie Lamport. Implementation of reliable distributed multiprocess systems. *Computer Networks: The International Journal of Distributed Informatique*, 2(2):95–114, May 1978. ISSN 0376-5075.

[78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[79] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[80] Leslie Lamport and Mike Massa. Cheap Paxos. In *Proceedings of 2004 International Conference on Dependable Systems and Networks*, pages 307–314, Florence, Italy, June 2004.

[81] Butler Lampson. The ABCD of Paxos. In *Proceedings of the twentieth Annual ACM Symposium on Principles of Distributed Computing*, page 13. ACM Press, 2001.

[82] Mikel Larrea, Antonio Fernández, and Sergio Arévalo. Eventually consistent failure detectors. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 326–327, Crete Island, Greece, July 3–6, 2001. SIGACT/SIGARCH and EATCS.

[83] Harry C. Li, Lorenzo Alvisi, and Allen Clement. The game of Paxos. Technical Report CS-TR-05-24, The University of Texas at Austin, Department of Computer Sciences, May 2005.

[84] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using fdr. In *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 147–166, London, UK, 1996. Springer-Verlag.

[85] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[86] Dahlia Malkhi and Michael Reiter. Unreliable intrusion detection in distributed computations. In *Proceedings of the 10th Computer Security Foundations Workshop*, pages 116–124, Rockport, MA, 1997.

[87] Jean-Philippe Martin and Lorenzo Alvisi. Fast Byzantine Paxos. Technical Report TR-04-07, University of Texas at Austin, Department of Computer Science., 2004.

[88] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using mur/spl phi/. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 141–153, Washington, DC, USA, 1997. IEEE Computer Society.

[89] Achour Mostéfaoui and Michel Raynal. Low cost Consensus-based Atomic Broadcast. In *Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing*, pages 45–52. IEEE Computer Society, 2000.

[90] Achour Mostéfaoui and Michel Raynal. Solving Consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach. In *Proceedings of*

the 13th International Symposium on Distributed Computing, pages 49–63, London, UK, 1999. Springer-Verlag.

[91] Achour Mostéfaoui, Sergio Rajsbaum, and Michel Raynal. Conditions on input vectors for Consensus solvability in asynchronous distributed systems. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, pages 153–162. ACM Press, 2001.

[92] Achour Mostéfaoui, Sergio Rajsbaum, and Michel Raynal. A versatile and modular Consensus protocol. In *Proceedings of International IEEE Conference on Dependable Systems and Networks*, pages 364–373, 2002.

[93] Sape Mullender, editor. *Distributed Systems*. ACM Press, New York, 2nd edition, 1993.

[94] Sam Owre, John M. Rushby, N. Shankar, and Friedrich W. von Henke. Formal verification of fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[95] Philippe Raipin Parvedy and Michel Raynal. Optimal early stopping Uniform Consensus in synchronous systems with process omission failures. In *SPAA '04: Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 302–310. ACM Press, 2004.

[96] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998. URL `citeseer.ist.psu.edu/paulson00inductive.html`.

[97] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[98] Fernando Pedone and André Schiper. Handling message semantics with Generic Broadcast protocols. *Distributed Computing*, 15(2):97–107, 2002.

[99] Fernando Pedone and André Schiper. Optimistic Atomic Broadcast: a pragmatic viewpoint. *Theoretical Computer Science*, 291(1):79–101, 2003.

[100] Fernando Pedone and André Schiper. On the inherent cost of Generic Broadcast. Technical Report IC/2004/46, Swiss Federal Institute of Technology (EPFL), May 2004.

[101] Fernando Pedone and André Schiper. Optimistic Atomic Broadcast. In *Proceedings of the* 12^{th} *International Symposium on Distributed Computing*, pages 318–332, September 1998.

[102] Fernando Pedone and André Schiper. Generic Broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 94–108, 1999.

[103] Fernando Pedone, Rachid Guerraoui, and André Schiper. Exploiting Atomic Broadcast in replicated databases. In *Proceedings of EuroPar*, pages 513–520, 1998.

[104] Fernando Pedone, André Schiper, Péter Urbán, and David Cavin. Solving agreement problems with weak ordering oracles. In *Proc. 4th European Dependable Computing Conference (EDCC-4)*, number 2485 in LNCS, pages 44–61, Toulouse, France, October 2002. Springer. URL `http://lsewww.epfl.ch/Publications/ById/309.html`.

[105] David Powell. Group communication. *Commun. ACM*, 39(4):50–53, 1996. ISSN 0001-0782.

[106] Roberto De Prisco, Butler W. Lampson, and Nancy A. Lynch. Revisiting the Paxos algorithm. In *Workshop on Distributed Algorithms*, pages 111–125, 1997.

[107] Python. Python programming language, 2005. URL `http://www.python.org/`.

[108] Michel Raynal. Consensus in synchronous systems: a concise guided tour. Technical Report 1497, IRISA, Jul 2002.

[109] Michel Raynal. A short introduction to failure detectors for asynchronous distributed systems. Technical Report PI 1613, IRISA, 2004.

[110] Luís E. T. Rodrigues and Michel Raynal. Atomic Broadcast in asynchronous crash-recovery distributed systems. In *20th International Conference on Distributed Computing Systems (ICDCS '00)*, pages 288–297. IEEE, April 2000.

[111] Luís E. T. Rodrigues, Paulo Veríssimo, and Antonio Casimiro. Using Atomic Broadcast to implement a posteriori agreement for clock synchronization. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 115–124, Princeton, New Jersey, October 1993. IEEE.

[112] André Schiper. Early Consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, April 1997.

[113] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[114] Dale Skeen. Nonblocking commit protocols. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 133–142, New York, NY, USA, 1981. ACM Press.

[115] Alfred Tarski. A fixed point theorem and its applications. *Pacific Journal of Mathematics*, pages 285–309, 1955.

[116] Peter van Emde Boas, Jaroslav Pokorný, Mária Bieliková, and Julius Stuller, editors. *SOFSEM 2004: Theory and Practice of Computer Science, 30th Conference on Current Trends in Theory and Practice of Computer Science, Merin, Czech Republic, January 24-30, 2004*, volume 2932 of *Lecture Notes in Computer Science*, 2004. Springer.

[117] Pedro Vicente and Luís Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. In *Proceedings of 21st Symposium on Reliable Distributed Systems*, Osaka, Japan, 2002. IEEE Computer Society.

[118] Toh Ne Win and Michael D. Ernst. Verifying distributed algorithms via dynamic analysis and theorem proving. Technical Report 841, MIT Lab for Computer Science, May 2002.

[119] Toh Ne Win, Michael D. Ernst, Stephen J. Garland, Dilsun Kırlı, and Nancy Lynch. Using simulated execution in verifying distributed algorithms. *Software Tools for Technology Transfer*, 6(1):67–76, July 2004.

[120] Piotr Zieliński. Latency-optimal Uniform Atomic Broadcast algorithm. Technical Report UCAM-CL-TR-582, Computer Laboratory, University of Cambridge, February 2004.
Available at `http://www.cl.cam.ac.uk/TechReports/`.

[121] Piotr Zieliński. Paxos at war. Technical Report UCAM-CL-TR-593, Computer Laboratory, University of Cambridge, June 2004. Available at `http://www.cl.cam.ac.uk/TechReports/`.

[122] Piotr Zieliński. Optimistic Generic Broadcast. In *Proceedings of the 19th International Symposium on Distributed Computing*, pages 369–383, Kraków, Poland, September 2005.

Appendix A

Optimistically Terminating Consensus

A.1 A time metric for asynchronous systems

Let t be real time. We will show that any asynchronous run r can be assigned a continuous time metric $t_r(t)$ such that (i) $t_r(t) \to \infty$ as $t \to \infty$, and (ii) messages between correct processes have transmission times of at most 2, which means that $d \leq 2$. Here, $t_r(t)$ is the time-metric time corresponding to all events that occur at real time t.

Consider a sequence of real-time values $t_1, t_2, \ldots$ defined in the following way. Time t_1 is arbitrary but finite, for example, one second after the start of the algorithm. Let t'_{i+1} be the time when all messages sent by correct processes to correct processes at time t_i or before have been received. We define $t_{i+1} = \max\{t'_{i+1}, t_i + \Delta\}$, where $\Delta > 0$ is an arbitrary but finite period of time, for example, one second. The purpose of Δ is to ensure that the sequence $t_1, t_2, \ldots$ is strictly increasing and tends to infinity.

We define the time metric $t_r(t)$ as any continuous, strictly increasing function of t that satisfies $t_r(t_i) = i$ for all $i = 1, 2, \ldots$, and $t_r(t) \to \pm\infty$ as $t \to \pm\infty$.

Theorem A.1.1. *In any such time metric t_r, all messages between correct processes have transmission times shorter than 2.*

Proof. Extend the sequence $t_1, t_2, \ldots$ with $t_0 = -\infty$, so that every real time t belongs to some interval $(t_i, t_{i+1}]$. Consider a message m sent by a correct process to a correct process at real time $t \in (t_i, t_{i+1}]$. By definition, m reaches its destination by time $t'_{i+2} \leq t_{i+2}$. Therefore, in time metric t_r, message m was sent after time i and received no later than $i + 2$, which proves the assertion. □

A.2 Onecast

Lemma A.2.1 (Integrity). *No learner onedelivers two different messages.*

```
1   initially variables sent and received are both empty (⊥)
2   when the owner executes onecast(x) do                    { assume x ≠ ⊥ }
3     if sent = ⊥ then
4       sent ← x
5     broadcast "onecast sent" to all learners
6   initially previous = ⊥ (at learners)
7   when a learner receives "onecast x" with x ≠ ⊥ from the owner do
8     if received = ⊥ then
9       received ← x
10    onedeliver(received)
```

Figure A.1: Implementation of onecast.

Proof. Since learners accept only messages "onecast x" with $x \neq \perp$, the variable *received* at a particular learner can be set only once. Learners onedeliver only the contents of *received*, which proves the assertion. □

Lemma A.2.2 (Validity). *If the owner is honest and a learner onedelivers x, then the owner must have onecast x.*

Proof. Since learners onedeliver only the contents of their variables *received*, the learner must have executed $received \leftarrow x$. Therefore, the learner received "onecast x" from the owner, so the owner broadcast "onecast x". Honest learners broadcast only "onecast *sent*", which means that $x = sent$ at the time of broadcasting this messages, which is only possible if the owner had previously executed $onecast(x)$. □

Lemma A.2.3 (Agreement). *If the owner is honest, then no two learners onedeliver different messages.*

Proof. Since honest owners do not onecast $\perp$, the variable *sent* can be set only once. The proof of the Validity property showed that every value x onedelivered by a learner was at some point equal to the contents of the variable *sent* at the owner, which implies the assertion. □

Lemma A.2.4 (Termination). *If the owner is correct and executes onecast, then all correct learners will execute onedeliver in one communication step.*

Proof. If the correct owner executes onecast, it also broadcasts "onecast *sent*" with $sent \neq \perp$. This message reaches all correct learners within one communication step and triggers onedelivery of *received*. □

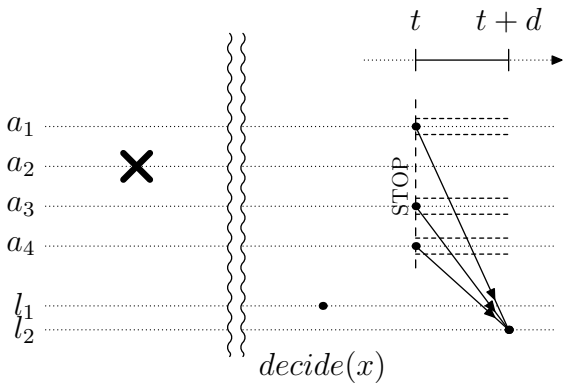


Figure A.2: Example of a run r' examined in Theorem A.3.1

A.3 OTC

Theorem A.3.1. *If an algorithm satisfies Permanent Validity, Possibility, and Integrity, then it also satisfies Standard Validity.*

Proof. Consider a run r in which some learner l_1 decides on x. To show Standard Validity, we have to prove that some honest acceptor executed *propose*(x).

Consider a run r' which is identical to r except that, at some time t after learner l_1 decided, all correct acceptors execute *stop* and freeze for one communication step (Figure A.2). Runs r and r' are identical until time t, so learner l_1 decides on x in run r' as well.

In run r, all correct acceptors execute *stop* at time t, so some correct learner l_2 will enter a complete state by time $t+d$. At that time, predicate *possible*(x) must hold at l_2 because learner l_1 decided on x (Possibility). Permanent Validity implies that *valid*(x) must hold as well. Then, the Integrity property implies that an honest acceptor executed *propose*(x) in run r' before time t. Since runs r and r' differ only in the *stop* action executed at time t, an honest acceptor executed *propose*(x) in run r as well, which implies the assertion.□

Theorem A.3.2. *If an algorithm satisfies Permanent Agreement and Possibility, then it also satisfies Standard Agreement.*

Proof. Consider a run r in which some learner l_1 decides on x_1 and another learner l_2 decides on x_2. To show Standard Agreement, we have to prove that $x_1 = x_2$.

Consider a run r' which is identical to r except that, at some time t after both learners decided, all correct acceptors execute *stop* (Figure A.3). Runs r and r' are identical until time t, so learners l_1 and l_2 decide on x_1 and x_2, respectively, in run r' as well.

All correct acceptors executed *stop* at time t, so a correct learner l_3 will eventually enter a complete state. At that time, predicates *possible*(x_1) and *possible*(x_2) must hold at l_3 because learners l_1 and l_2 decided on x_1 and x_2, respectively (Possibility). Permanent

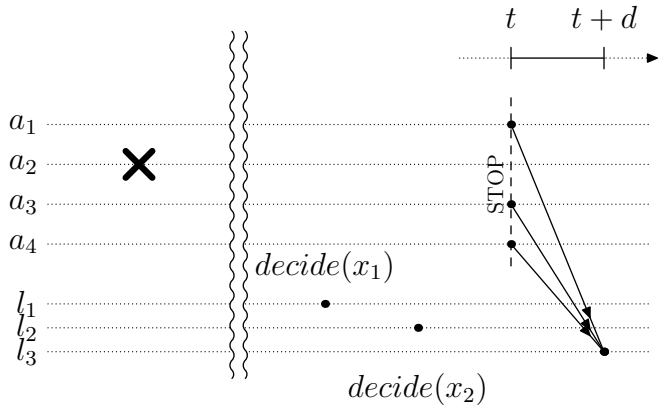


Figure A.3: Example of a run r' examined in Theorem A.3.2

---

1 **when** acceptor a_i executes *propose*(x) **do**
2 onecast x using $onecast_i$
3 **when** acceptor a_i executes *stop* **do**
4 onecast $\top$ using $onecast_i$
5 **predicate** *decision*(x) at a learner **is**
6 at least $n - q$ instances $onecast_i$ delivered x
7 **predicate** *possible*(x) at a learner **is**
8 at most $q + m$ instances of $onecast_i$ delivered a non-x
9 **predicate** *valid*(x) at a learner **is**
10 more than m instances of $onecast_i$ delivered x

Figure A.4: Generic Agreement algorithm.

Agreement requires that *possible*(x) can hold for at most one x, which implies the assertion $(x_1 = x_2)$. □

A.4 Generic Agreement

Theorem A.4.1 (Strong Standard Validity). *Assume that $n > f+m+q$. If decision(x) holds at a learner, then valid(x) holds at all complete learners.*

Proof. Every execution of *stop* involves onecasting, so all onecast instances owned by correct acceptors have executed *onedeliver* at all complete learners. The assumption implies that at least $n - q - f > m$ of those instances have onedelivered x, which implies the assertion. □

Theorem A.4.2 (Weak Permanent Validity). *Assume that $n > f+m+q$. If possible(x) holds at a complete learner, then an honest acceptor executed propose(x).*

Proof. Every execution of *stop* involves onecasting, so the assumption implies all $n - f$ onecast instances owned by correct acceptors have executed *onedeliver*. If *possible*(x) holds, then at most $q + m$ onecast instances onedelivered a non-x. This means that at least $n-f-q-m > 0$ instances owned by correct acceptors onedelivered x, which implies the assertion. □

Lemma A.4.3 (Standard Agreement). *Assume $n > m + 2q$. There is at most one x for which decision(x) holds at some learner; in other words, there is at most one decision.*

Proof. Assume *decision*(x) holds at some learner. This means that at least $n-q$ instances $onecast_i$ delivered x, which implies that at least $n - q - m$ honest acceptors proposed x. If the assertion does not hold, then *decision*(x) holds, possibly at different learners, for at least two different x. Since no honest acceptor proposes two different values, this means that $2(n - q - m) > n - m$ honest acceptors proposed something. This contradicts the fact that there are only $n - m$ honest acceptors. □

A.5 Two-step OTC

Lemma A.5.1. *Assume $n > f + m + q$ and consider a chain $A_1 \to \cdots \to A_k$. If $possible_{A_k}(x)$ holds at a complete learner, then an honest acceptor proposed x in A_1.*

Proof. By induction on k. The base case $k = 1$ follows directly from Theorem A.4.2. If $possible_{A_k}(x)$ holds at a complete learner, then the inductive assumption for the subchain $A_2 \to \cdots \to A_k$ implies that an honest acceptor proposed x to A_2. Therefore, some learner in A_1 decided on x, which by Theorem A.4.1 implies *valid*(x) at our complete learner. Integrity of A_1 implies the assertion. □

Theorem A.5.2 (Permanent Validity). *Assume $n > f + m + q$ and consider a chain $A_1 \to \cdots \to A_k$ with $k \geq 2$. For any complete learner, possible$(x) \implies$ valid(x) for all x.*

Proof. Predicate *possible*$(x) \stackrel{\text{def}}{=} possible_{A_k}(x)$, so Lemma A.5.1 applied to the subchain $A_2 \to \cdots \to A_k$ implies that an honest acceptor proposed x to A_2. Therefore, some learner in A_1 decided on x, which by Theorem A.4.1 implies the assertion. □

Theorem A.5.3 (Permanent Agreement). *Assume $n > f + m + q$ and consider a chain $A_1 \to \cdots \to A_k$ with $k \geq 2$. For any complete learner, possible(x) holds for at most one x.*

Proof. Predicate *possible*$(x) \stackrel{\text{def}}{=} possible_{A_k}(x)$, so Theorem A.4.2 applied to A_k implies that some honest acceptor proposed x in A_k. The construction of the chain $A_1 \to \cdots \to A_k$ implies that some learner decided on x in A_{k-1}. Since $q \leq f$, we have $n > m + 2q$, and Lemma A.4.3 applied to A_{k-1} implies the assertion. □

A.6 Multi-step OTC

The multi-step OTC algorithm consists of three OTC chains from Section 2.4 executed in parallel:

$$\begin{array}{lllll} A_1 & & & & \text{with } q = q_1, \\ B_1 & \to & B_2 & & \text{with } q = q_2, \\ C_1 & \to & C_2 & \to \quad C_3 & \text{with } q = q_3. \end{array}$$

Instances A_1, B_1, and C_1 share onecast instances; each proposed value is proposed to all three chains at the same time. In other words, $propose(x)$ consists of $propose_{A_1}(x)$, $propose_{B_1}(x)$, and $propose_{C_1}(x)$. Stopping the algorithm involves stopping all six Generic Agreement instances.

Theorem A.6.1 (Permanent Agreement). *Assume that*

$$\begin{aligned} n &> f + 2m + 2q_1, \\ n &> f + \ \ m + q_2 + \min\{m, q_1\}, \\ n &> f + \ \ m + q_3. \end{aligned}$$

For any complete learner, $possible(x)$ holds for at most one x.

Proof. Predicate $possible(x)$ is defined as

$$possible(x) \stackrel{\text{def}}{=} \big(possible_{A_1}(x) \wedge \neg\exists\, x' \neq x : valid_{C_2}(x')\big) \vee possible_{B_2}(x) \vee possible_{C_3}(x)$$

The assumption $n > f + 2m + 2q_1$ implies Permanent Agreement of A_1, whereas the other two assumptions imply Permanent Agreement of chains $B_1 \to B_2$ and $C_1 \to C_2 \to C_3$ (Theorem A.5.2). Therefore, predicates $possible_{A_1}(x)$, $possible_{B_2}(x)$, and $possible_{C_3}(x)$ can each hold for at most one x. To complete the proof, we consider three values x, y, z, which – if they exist – satisfy

$$possible_{A_1}(x) \wedge \neg\exists\, x' \neq x : valid_{C_2}(x'), \qquad possible_{B_2}(y), \qquad possible_{C_3}(z).$$

We need to prove that all existing x, y, z must be the same. In other words, we have to show that $x = y$, $x = z$, and $y = z$.

- *Equality $x = z$.* Since $n > f + m + q_3$, Theorem A.5.2 states that the subchain $C_2 \to C_3$ satisfies Permanent Validity. Therefore, $possible_{C_3}(z) \implies valid_{C_2}(z)$, which implies $x = z$.

- *Equality $y = z$.* If $possible_{C_3}(z)$, then Lemma A.5.1 used for the subchain $C_2 \to C_3$ implies that an honest acceptor proposed z to C_2, which implies that z was a decision

in C_1. Similarly, Theorem A.4.2 applied to B_2 implies that y was a decision in B_1. The assumption $q_3 \geq q_2$ implies that $decision_{B_1}(y) \implies decision_{C_1}(y)$. Since $n > f + m + q_3 \geq m + 2q_3$, Lemma A.4.3 implies that $y = z$.

- *Equality* $x = y$. Showing $x = y$ requires considering two cases of the assumption $n > f + m + q_2 + \min\{m, q_1\}$:
  - *Case* $n > f+m+q_2+m$. In this case, instance B_2 satisfies Permanent Validity. As a result, $possible_{B_2}(y) \implies valid_{B_2}(y) \iff valid_{C_2}(y)$, so $x = y$.
  - *Case* $n > f + m + q_2 + q_1$. Theorem A.4.2 applied to B_2 implies that y was a decision in B_1. Figure A.4 shows that this implies that at least $n - q_2 - f$ correct acceptors proposed y to B_1. On the other hand, for complete learners, $possible_{A_1}(x)$ implies that at most $q_1 + m$ correct acceptors proposed a non-x to A_1. Since honest acceptors propose the same to A_1 and B_1, this implies $n - q_2 - f \leq q_1 + m$, which contradicts the assumption $n > f + m + q_1 + q_2$.□

Appendix B

Agreement abstractions

B.1 Coordinated Consensus with malicious processes

B.1.1 Function *choose*

Lemma B.1.1. *Any signed state S_i is a learner state in OTC_i.*

Proof. In other words, we have to prove that a new learner with the state S_i can be introduced to our system without creating contradictions. Recall that the state of a learner consists of all messages received from the acceptors. Malicious acceptors can send arbitrary messages so we can make them send the messages from S_i to our learner. All messages in S_i are signed by their senders, so honest acceptors did indeed broadcast all messages attributed to them in S_i. Therefore, we can make our learner receive them as well. Finally, messages sent by acceptors but not received by our learner might have been lost by the network; for that we just need to assume our learner to be non-maliciously faulty. □

Lemma B.1.2. *Assume $x = choose(\langle S_j \rangle_{j<i}, x_i)$, where each S_j is a semi-complete learner state in OTC_j and x_i is a proposal, both received from c_i by an honest acceptor. Then,*

1. *An honest acceptor received x as a proposal x_j from some coordinator c_j with $j \leq i$.*
2. *No decision different than x was made in rounds $j < i$.*

Proof. Induction on i. If $possible_{S_j}(x)$ does not hold for any x and $j < i$, then $x = x_i$ and both assertions hold (the first one for $j = i$). Otherwise, assume that $j < i$ is the highest round for which $possible_{S_j}(x)$ holds for some x.

Semi-completeness of S_j means that $possible_j(x) \implies valid_j(x)$, so some honest acceptor a in round j executed $OTC_j.propose(x)$. Value x equals $choose(\langle S'_k \rangle_{k<j}, x_j)$,

```
1  function choose(⟨S_j⟩_{j<i}, x_i)
2    if ¬possible_{S_j}(x) for all x and all j < i then
3      return x_i
4    else
5      let j < i be the largest round number for which possible_{S_j}(x) holds for some x
6      return the x for which possible_{S_j}(x) holds
```

Figure B.1: Function *choose*.

where x_j is the proposal a received from c_j, and each S'_k is a semi-complete learner state in OTC_k (Lemma B.1.1), possibly different from S_k. The inductive assumption for $i = j$ gives the first assertion.

The same assumption shows that no decision other than x was made in any OTC_k with $k < j$. Then, semi-completeness of S_j implies that $possible_j$ holds only for x, so no other decision was made in OTC_j. Finally, by the definition of j, no decisions were made in any OTC_k with $j < k < i$. This paragraph showed that the second assertion holds as well. □

Lemma B.1.3. *If $OTC_i.decision(x)$ holds at some learner, then $x = choose(\langle S_j \rangle_{j<i}, x_i)$, where each S_j is a semi-complete learner state in OTC_j and x_i is a proposal, both received from c_i by an honest acceptor.*

Proof. Standard Validity of OTC_i (Theorem A.3.1) implies that some honest acceptor a executed $OTC_i.propose(x)$. As a result, $x = choose(\langle S_j \rangle_{j<i}, x_i)$, where $\langle S_j \rangle_{j<i}$ is a semi-complete collection of states and x_i is a proposal, both received from c_i by a. All states $\langle S_j \rangle_{j<i}$ are semi-complete and signed, so Lemma B.1.1 implies the first assertion. Honesty of a implies the second assertion. □

Corollary B.1.4. *If $OTC_i.decision(x)$ holds at some learner, then (i) an honest acceptor received x as a proposal x_j from some coordinator c_j with $j \leq i$. (ii) no decision different than x was made in rounds $j < i$.*

Proof. Directly from Lemmas B.1.3 and B.1.2. □

B.1.2 Validity and Agreement

Lemma B.1.5. *If a learner decided on x, then $OTC_i.decision(x)$ holds for some i at some (possibly different) learner.*

Proof. Consider the first learner to decide on x. Lines 16–22 show that it decided either because $OTC_i.decision(x)$ held or because it received "decide on x" from more than m

```
1   when coordinator c_i executes propose(x_i) do
2     for all j < i do
3       wait until the state S_j of c_i as a secure learner in OTC_j is semi-complete
4     broadcast x_i and ⟨S_j⟩_{j<i} to all acceptors
5   when an acceptor receives x_i and ⟨S_j⟩_{j<i} from c_i for the first time do
6     if all signatures in ⟨S_j⟩_{j<i} are correct and all states S_j are semi-complete then
7       x ← choose(⟨S_j⟩_{j<i}, x_i)                    { select the proposal }
8       OTC_i.propose(x)
9   when for each j < i, an acceptor received
10          "stop round j" from more than m + f acceptors do
11    start round j timer
12  when acceptor has not decided in OTC_i and the round i timeout expired, or
13          received message "stop round i" from more than m acceptors do
14    OTC_i.stop
15    broadcast "stop round i"
16  when a learner has OTC_i.decision(x) or
17          received "decide on x" from more than m acceptors do
18    decide(x)
19    if the learner is also an acceptor then
20      broadcast "decide on x"
21    wait until received "decide on x" from more than m + f acceptors
22    halt
```

Figure B.2: Coordinated Consensus with malicious participants.

acceptors. The former case is the assertion. The latter case is impossible because one of these message must have been sent by an honest acceptor. This acceptor had decided on x, which contradicts the choice of our learner to be the first to do so. □

Theorem B.1.6 (Validity). *If all coordinators are honest and decision(x) holds at some learner, then some coordinator proposed x.*

Proof. Lemma B.1.5 states that $OTC_i.decision(x)$ holds for some i at some learner. Then, Corollary B.1.4 states that an honest acceptor received x as the proposal x_j from some coordinator c_j. The assumption of c_j's honesty implies the assertion. □

Theorem B.1.7 (Agreement). *There is at most one x for which decision(x) holds at some learner.*

Proof. Assume *decision*(x) and *decision*(y) each holds at some learner. We will show that $x = y$. Lemma B.1.5 implies that there are rounds i and j so that $OTC_i.decision(x)$ and $OTC_j.decision(y)$ each holds at some learner. There are two cases to consider: $i = j$ and $i \neq j$.

If $i = j$, then Standard Agreement of $OTC_i = OTC_j$ (Theorem A.3.2) implies $x = y$. If $i \neq j$, then without loss of generality, we can assume $j < i$. Corollary B.1.4 applied to $OTC_i.decision(x)$ implies that decisions made in all rounds before i must equal x. In particular, $OTC_j.decision(y)$ implies $x = y$. □

B.1.3 Termination

To prove Termination, we assume $n > 2f + m$.

Lemma B.1.8. *If an honest acceptor halts, then all correct learners will eventually decide.*

Proof. Lines 16–22 shown that an honest acceptor halts only after receiving "decide on x" from at least $m + f$ acceptors. Therefore, more than m correct acceptors have indeed broadcast "decide on x". As a result, all correct learners will eventually receive this message from more than m acceptors, and decide x. □

Lemma B.1.9. *If all correct acceptors decide on x, then all correct learners will eventually decide on x and halt.*

Proof. The assumption implies that all $n - f > m + f$ correct acceptors have sent "decide on x". This means that all correct learners will eventually receive these messages, decide on x, and halt (lines 16–22). □

If an honest acceptor halts, then Lemma B.1.8 implies Termination. Since the purpose of this section is to prove Termination, from now on we assume that no honest acceptor has halted. At the same time, we assume that all correct acceptors have started the algorithm, which is required by Termination.

Lemma B.1.10. *If all correct acceptors stop all rounds $j < i$, then either they will all stop round i or all correct learners will decide.*

Proof. The assumption implies that every correct acceptor will eventually receive "stop round j" from all $n - f > f + m$ correct acceptors, for each $j < i$ (lines 12–15). Since we assume that all correct acceptors have started executing the algorithm, they will all start their round i timers and eventually either decide or stop round i (lines 9–11).

If more than m correct acceptors decide in round i, then lines 16–22 ensure that all correct learners will eventually decide in that round. Similarly, if more than m correct acceptors stop round i, then lines 12–15 ensure that all correct acceptors will eventually do so. Therefore, the assertion can only be false if the number of correct acceptors $n - f \leq m + m$, which contradicts the assumption $n > 2f + m \geq f + 2m$. □

Lemma B.1.11. *If there is a correct learner that never decides, then all correct acceptors will stop all rounds.*

Proof. To obtain a contradiction, let i be the first round which is never stopped by all correct acceptors. The choice of i means that rounds $j < i$ have been stopped by all correct acceptors. Lemma B.1.10 implies that either all correct acceptors will stop round i (a contradiction with the definition of i) or all correct learners will decide (a contradiction with the lemma assumption). □

Lemma B.1.12. *If an honest acceptor starts its round i timer, then all correct acceptors will stop all rounds $j < i$ in one communication step.*

Proof. Consider any round $j < i$. The assumption implies that some acceptor received more than $m + f$ messages "stop round j" (lines 9–11). More than m of them must have been sent by correct acceptors, so all correct acceptors will receive them in one step. All of them will stop round j, which implies the assertion (lines 12–15). □

Lemma B.1.13. *Assume that at most q acceptors are faulty. Consider a round i with a correct coordinator and OTC_i satisfying Optimistic Termination (q, k). Assume that, by time t, coordinator c_i proposed and all its states S_j with $j < i$ are complete. If no correct acceptor executes $OTC_i.stop$ before time $t + (k + 1)d$, then all correct learners in OTC_i will have decided by then.*

Proof. To show that all correct learners will decide by time $t + (k + 1)d$, we assume that no correct acceptor ever executes $OTC_i.stop$. No learner can distinguish this and the original run by time $t + (k + 1)d$, so the assertion still holds in the original run.

Consider the coordinator c_i at time t. We assume that it has executed $propose(x_i)$, and all S_j with $j < i$ are complete, so by Permanent Validity and Permanent Agreement of OTC_j, also semi-complete. Therefore c_i broadcasts its proposal x_i and the collection of states $\langle S_j \rangle_{j<i}$ by time t.

These messages reach all correct acceptors by time $t + d$. All correct acceptors receive the same x_i and $\langle S_j \rangle_{j<i}$, so they propose the same $x = choose(\langle S_j \rangle_{j<i}, x_i)$ to OTC_i. Since at most q acceptors are faulty and no correct acceptor executes *stop*, Optimistic Termination (q, k) implies that all correct learners will have $decision(x)$ by time $t + (k + 1)d$.□

Theorem B.1.14 (Termination). *If infinitely many coordinators are correct, all of them proposed by some time t_1, and all correct acceptors start the algorithm, then all correct learners will eventually decide.*

Proof. To obtain a contradiction, assume that there is a correct learner that never decides. Lemma B.1.8 implies that no acceptor ever halts. Then, Lemma B.1.11 states that all correct acceptors stop all rounds.

We assume that all rounds $i > i_0$ satisfy Optimistic Termination (f, k) for some k. Assume the algorithm started at time t_0. Since the sequence of timeout periods for successive rounds tends to infinity, there is a round i_1 such that all rounds $i \geq i_1$ have timeouts longer than $\max\{(k+3)d, t_1 - t_0\}$. There are infinitely many correct coordinators c_i, so there is one with $i > \max\{i_0, i_1\}$.

Let t be the time at which all rounds $j < i$ were stopped by all correct acceptors. Since $i_1 < i$ and the timeout period for round i_1 is longer than $t_1 - t_0$, we deduce that $t > t_1$. In other words, all correct coordinators, c_i in particular, proposed by time t.

If all correct acceptors have all rounds $j < i$ stopped at time t, then all states S_j of c_i will be complete at time $t + d$. Also, Lemma B.1.12 implies that no correct acceptor started round i timer before time $t - d$. Since $i > i_1$, its timeout period is longer than $(k + 3)d$, so no correct acceptor will stop round i before $(t + d) + (k + 1)d$. Finally, since $i > i_0$, instance OTC_i satisfies Optimistic Termination (f, k), the assumption of Lemma B.1.13 holds, so all correct learners will eventually decide. □

Theorem B.1.15 (Latency). *If the run is timely, coordinator c_1 is correct, at most q acceptors are faulty, and OTC_1 satisfies Optimistic Termination (q, k) for some k, then all correct learners will decide on the value x_1 proposed by c_1 in $k + 1$ communication steps.*

Proof. In the eventual synchrony model, a run is timely if the maximum message transmission time d is "sufficiently small" and all correct coordinators propose within one step of the start of the algorithm. We assume that, in the context of this algorithm, "sufficiently small d" means that $d \cdot (k + 3)$ is smaller than the timeout period of the first round. This implies that no correct acceptor will stop the first round earlier than $k + 1$ communication steps after c_1 proposed.

Given these assumptions, Lemma B.1.13 proves that $OTC_1.decision(x)$ will hold at all correct learners in $k+1$ steps. Corollary B.1.4 shows that an honest acceptor has received x as the proposal from some c_i with $i \leq 1$. Since c_1 is the only such coordinator and it is correct (therefore also honest), the assertion holds. □

B.2 Consensus

Theorem B.2.1 (Validity). *If all acceptors are honest and $decision(x)$ holds at some learner, then some acceptor proposed x.*

Proof. Predicate $decision(x)$ holds in Consensus only if the analogous predicate holds for the underlying Coordinated Consensus. Since all coordinators are honest acceptors, Validity of $Coord$ (Theorem B.1.6) implies that some acceptor executed $Coord.propose(x)$. This in turn implies the assertion. □

```
1  let c_1, c_2, ... = a_1, a_2, ..., a_n, a_1, a_2, ...
2  when acceptor a_i executes propose(x) do
3    explicitly start instance Coord as acceptor a_i
4    execute Coord.propose(x) as coordinators c_i, c_{i+n}, c_{i+2n}, ...
5  when Coord.decision(x) at a learner do
6    decide(x)
```

Figure B.3: Implementing Consensus with Coordinated Consensus.

Theorem B.2.2 (Agreement). *There is at most one x for which $decision(x)$ holds at some learner.*

Proof. Predicate $decision(x)$ holds in Consensus only if the analogous predicate holds for the underlying Coordinated Consensus. The Agreement property of $Coord$ implies the assertion. □

Theorem B.2.3 (Termination). *If all correct acceptors have executed propose, then all correct learners will eventually decide.*

Proof. The Consensus algorithm decides immediately after the underlying Coordinated Consensus does so. Therefore, it is sufficient to show that all conditions required to ensure Termination of the latter algorithm hold (Theorem B.1.14).

The assumption implies that all correct acceptors started $Coord$. We assume that at least one acceptor a_i is correct. In the rotating coordinator scheme, acceptor a_i plays coordinators c_i, c_{i+n}, c_{i+2n}, ..., so infinitely many coordinators are correct. The number of correct acceptors is finite, so all correct coordinators eventually propose. Therefore, Theorem B.1.14 implies the assertion. □

Theorem B.2.4 (Latency). *If the run is timely, acceptor a_1 is correct, at most q acceptors are faulty, and OTC_1 satisfies Optimistic Termination (q, k) for some k, then all correct learners will decide in $k + 1$ communication steps.*

Proof. Straightforward from the The Latency property of $Coord$. □

B.3 Individual Consensus

Theorem B.3.1 (Sensitive Validity). *If the owner is honest and $decision(x)$ holds at some learner, then x has either been proposed by the owner or equals* ABORT. *If the owner is correct and the run is timely, the former case must hold.*

```
1   c_1 = p
2   c_2, c_3, ... = a_1, a_2, ..., a_n, a_1, a_2, ...
3   when the owner p executes propose(x) do
4     execute Coord.propose(x) as coordinator c_1
5   task at acceptor a_i is
6     explicitly start Coord as acceptor a_i
7     execute Coord.propose(ABORT) as coordinators c_{i+1}, c_{i+n+1}, c_{i+2n+1}, ...
8   when an acceptor received proposal x_i from c_i with i > 1 do
9     ignore the actual x_i and behave as if x_i = ABORT was received
10  when Coord.decision(x) at a learner do
11    decide(x)
```

Figure B.4: Implementing Individual Consensus with Coordinated Consensus.

Proof. Predicate *decision*(x) holds in Individual Consensus only if the analogous predicate holds for the underlying Coordinated Consensus. Lemma B.1.5 and Corollary B.1.4 imply that some honest acceptor received x from some coordinator c_i. If $i > 1$ then $x = \text{ABORT}$. If $i = 1$, then x must have been proposed by the (honest) owner $p = c_1 = c_i$.

We assume that all OTC instances satisfy Optimistic Termination (q, f). If the run is timely, then the Latency property of $Coord$ implies that all correct learners have decided on the value proposed by the owner c_1. The Agreement property of Coordinated Consensus implies the assertion. □

Theorem B.3.2 (Agreement). *There is at most one x for which decision(x) holds at some learner.*

Proof. Predicate *decision*(x) holds in Consensus only if the analogous predicate holds for the underlying Coordinated Consensus. The Agreement property of $Coord$ implies the assertion. □

Theorem B.3.3 (Termination). *All correct learners will eventually decide.*

Proof. The Consensus algorithm decides immediately after the underlying Coordinated Consensus does so. Therefore, it is sufficient to show that all conditions required by Termination of the latter algorithm hold (Theorem B.1.14).

The assumption implies that all correct acceptors started $Coord$. We assume that at least one acceptor a_i is correct. In the rotating coordinator scheme, acceptor a_i plays coordinators c_i, c_{i+n}, c_{i+2n}, ..., so infinitely many coordinators are correct. The number of correct acceptors is finite, so all correct coordinators eventually propose. Therefore, Theorem B.1.14 implies the assertion. □

```
1  when the owner executes propose(x) do
2    Ind.propose(x)
3    broadcast "propose x" to all learners
4  when a learner receives "propose ABORT" from the owner do
5    decide(ABORT)
6  when Ind.decision(x) at a learner do
7    decide(x)
```

Figure B.5: Implementing Fast Individual Consensus with Individual Consensus in the crash-stop model.

Theorem B.3.4 (Latency). *If the run is timely, the owner is correct, at most q acceptors are faulty, and OTC_1 satisfies Optimistic Termination (q, k) for some k, then all correct learners will decide in $k + 1$ communication steps.*

Proof. Straightforward from the The Latency property of *Coord*. □

B.4 Fast Individual Consensus

In this section, we assume that the owner is honest.

Theorem B.4.1 (Sensitive Validity). *If decision(x) holds at some learner, then x has either been proposed by the owner or equals* ABORT. *If the owner is correct and the run is timely, the former case must hold.*

Proof. Predicate *decision*(x) implies either $Ind.decision(x)$ or that $x =$ ABORT and the learner has received "propose x" from the owner. In the former case, Sensitive Validity of Ind implies the assertion. In the latter case, the owner must have proposed ABORT, which also implies the assertion. □

Theorem B.4.2 (Agreement). *There is at most one x for which decision(x) holds at some learner.*

Proof. If no learner received "propose ABORT", then the assertion follows from the Agreement property of Ind. Otherwise, the owner must have proposed ABORT, so Validity of Ind implies that $Ind.decision(x)$ can only hold for $x =$ ABORT, which implies the assertion. □

Theorem B.4.3 (Termination). *All correct learners will eventually decide.*

Proof. Straightforward from the Termination property of Individual Consensus. □

```
1  when acceptor a_i executes propose(x) do
2    broadcast "propose x"
3    explicitly start Ind as acceptor a_i
4  when Ind.decision(x) at a learner do
5    decide(x)
6  predicate the virtual owner executed propose(x) is
7    x = f(x_1, ..., x_n), where x_i is the proposal of a_i
8  predicate received proposal x from the virtual owner is
9    x = f(x_1, ..., x_n), where x_i is the proposal received from a_i
```

Figure B.6: Implementing Atomic Commitment using Individual Consensus with a virtual owner.

Theorem B.4.4 (Latency). *If the run is timely and the owner is correct, then all correct learners decide in two communication steps. If in addition, the owner proposed* ABORT*, then the decision is made in one communication step.*

Proof. We assume that OTC_1 is implemented as single-value one-step OTC from Section 2.3 with $q = f$. The first part of the assertion follows from the Latency property of *Ind*. If the owner proposes ABORT, then all correct learners will receive "propose ABORT" in one step, which implies the second part of the assertion. □

B.5 Atomic Commitment

Theorem B.5.1 (Sensitive Validity of Distributed Function Computation). *If all acceptors are honest and decision(x) holds at some learner, then* $x = f(x_1, \ldots, x_n)$ *or* $x =$ ABORT*. If all acceptors are correct and the run is timely, the former case must hold.*

Proof. By definition, if the virtual owner proposes $x = f(x_1, \ldots, x_n)$ iff each (honest) acceptor a_i proposes x_i. The assumption implies that the virtual owner is honest, which by Validity of Individual Consensus (Theorem B.3.1) implies the assertion. □

Theorem B.5.2 (Sensitive Validity of Atomic Commitment). *If all acceptors are honest, then*

1. *If the run is timely, and all acceptors are correct and proposed* COMMIT*, then* COMMIT *is the only possible decision.*

2. *If at least one acceptor proposed* ABORT*, then* ABORT *is the only possible decision.*

Proof. In the first case, the virtual owner is correct and proposes COMMIT. Since the run is timely, Theorem B.5.1 implies the assertion. In the second case, the virtual owner proposed ABORT or nothing. The assertion follows again from Theorem B.5.1. □

```
1  when acceptor a_i executes propose(x) do
2    explicitly start parallel instances Ind_1, Ind_2, ..., Ind_n
3    Ind_i.propose(x)
4  when Ind_i.decision(x_i) for all i = 1, ..., n at a learner do
5    decide([x_1, ..., x_n])
```

Figure B.7: Implementing Interactive Consistency with n instances of Individual Consensus.

Theorem B.5.3 (Agreement). *There is at most one x for which decision(x) holds at some learner.*

Proof. Straightforward from the Agreement property of Individual Consensus. □

Theorem B.5.4 (Termination). *If all correct acceptors proposed, then all correct learners will eventually decide.*

Proof. The assumption implies that all correct acceptors started the instance Ind, so the assertion follows from Termination of Individual Consensus (Theorem B.3.3). □

Theorem B.5.5 (Latency). *If the run is timely, the virtual owner is correct, at most q acceptors are faulty, and OTC_1 satisfies Optimistic Termination (q, k) for some k, then all correct learners will decide in $k + 1$ communication steps.*

Proof. Follows from the Latency property of Individual Consensus (Theorem B.3.4). □

B.6 Interactive Consistency

Theorem B.6.1 (Sensitive Validity). *If a learner decides on $[v_1, \ldots, v_n]$ and acceptor a_i is honest, then v_i has either been proposed by a_i or equals* ABORT. *If a_i is correct and the run is timely, the former case must hold.*

Proof. The assumption implies that Ind_i decided on v_i. The assertion follows from Sensitive Validity of Ind_i (Theorem B.3.1). □

Theorem B.6.2 (Agreement). *There is at most one vector $[v_1, \ldots, v_n]$ for which the predicate decision($[v_1, \ldots, v_n]$) holds at some learner.*

Proof. The Agreement property of Ind_i (Theorem B.3.2) states that there is at most one v_i for which $Ind_i.decision(v_i)$ holds at some learners, which implies the assertion. □

Theorem B.6.3 (Termination). *If all correct acceptors proposed, then all correct learners will eventually decide.*

Proof. The assumption implies that all correct acceptors started all instances Ind_i, where $i = 1, 2, \ldots, n$. Therefore, the Termination property of Individual Consensus (Theorem B.3.3) implies that all Ind_i will eventually decide, which implies the assertion. □

Theorem B.6.4 (Latency). *If the run is timely, the owner is correct, at most q acceptors are faulty, and OTC_1 satisfies Optimistic Termination (q, k) for some k, then all correct learners will decide in $k + 1$ communication steps.*

Proof. Follows from the Latency property of Individual Consensus (Theorem B.3.4). □

Appendix C

Atomic Broadcast

C.1 Atomic Broadcast

Lemma C.1.1. *Consider any (Uniform) Consensus algorithm. If a learner decides on x, then at least one correct acceptor has seen x.*

Proof. To obtain a contradiction, assume that there is a run r_1, in which some learner l_1 decides on x at time t, and no correct acceptor ever sees x. Consider a run r_2, which is identical to r_1, except that all faulty acceptors that have not yet failed by time t in run r_1 fail at time t. Also assume that, in run r_2, all messages from faulty acceptors that have not yet reached their destination before time t are lost. Finally, assume that, in run r_2, every correct acceptor that did not propose by time t, proposes at time t some $x' \neq x$.

Runs r_1 and r_2 are identical until time t, so in r_2 no correct acceptor sees x by time t. After time t, correct acceptors receive only messages from correct acceptors. Therefore, no correct acceptor ever sees x in run r_2.

Consider a correct learner l_2, and assume that all messages from faulty acceptors to l_2 are lost. Since all correct acceptors propose in run r_2, the Termination property implies that learner l_2 will eventually decide. It cannot decide on x, because it has only received messages from correct acceptors, who have never seen this value. On the other hand, Agreement implies that l_2 must decide on x because l_1 has already done so. This contradiction proves the assertion. □

Theorem C.1.2 (Validity). *For any message m, every learner delivers m at most once, and only if m was abcast by a proposer.*

Proof. The first part of the assertion follows from the fact that learners deliver only undelivered messages. For the second part, assume that m is delivered in the k-th iteration of the loop. This implies that $m \in B_k$ and, by Validity of $batch_k$, at least one acceptor executed $batch_k.propose(\mathcal{M})$ with $m \in \mathcal{M} = B_k$. Hence, this acceptor must have received m, which implies the assertion. □

```
1  M is the set of received messages, initially ∅
2  when a proposer executes abcast(m) do
3    broadcast m to the acceptors
4  when an acceptor sees m eventually do
5    abcast(m)
6  task broadcasting at acceptors is
7    for k = 1, 2, ... do
8      wait for some message m ∉ M
9      insert m into M
10     batch_k.propose(M)
11 task delivery at learners is
12   for k = 1, 2, ... do
13     wait until batch_k.decision(B_k)
14     deliver all undelivered messages from B_k in some deterministic order
```

Figure C.1: Atomic Broadcast.

Theorem C.1.3 (Agreement). *For any two different messages m and m', it is impossible that one learner l delivers m without having previously delivered m', and another learner l' delivers m' without having previously delivered m.*

Proof. To obtain a contradiction, assume that the assertion does not hold. Let k be the first instance $batch_k$ that decides on a set of messages containing m. Similarly, let k' be the first instance $batch_{k'}$ that decides on a set of messages containing m'. These definitions are unambiguous thanks to the Agreement property of instances $batch_i$.

Learner l delivers m without having previously delivered m', which implies $k \leq k'$. Learner l' delivers m' without having previously delivered m, which implies $k' \leq k$. As a result, $k = k'$, so both learners deliver m and m' while delivering the same batch of messages B_k. All these messages are delivered in the same deterministic order, which implies the assertion. □

Theorem C.1.4 (Termination Validity). *If a correct proposer abcasts m, then all correct learners will eventually adeliver m.*

Proof. Consider the set $M \ni m$ of all messages ever received by a correct acceptor. Correct acceptors see all messages they receive, and (eventually) re-abcast all of them, therefore each correct acceptor receives all messages in M.

If M is infinite, then correct acceptors propose to infinitely many instances $batch_k$. Eventually, there will be an instance $batch_k$ in which no faulty acceptors participate and to which all correct acceptors propose some $\mathcal{M} \ni m$. By Validity, this instance decides on some $B_k \ni m$, which implies the assertion.

Now, consider the case in which M is finite and has k elements. All correct acceptors propose to instances $batch_1, \ldots, batch_k$, so all of these instances will eventually decide (Termination). Consider the decision B_k of $batch_k$. All proposals $\mathcal{M}$ in $batch_k$ have k elements, so B_k has k elements as well. Every message in B_k has been seen by a correct acceptor (Lemma C.1.1), which re-abcasts it so that all correct acceptors eventually receive it. As a consequence, $B_k \subseteq M$ and since $|B_k| = |M| = k$, we have $B_k = M \ni m$, which implies the assertion. □

Theorem C.1.5 (Termination Agreement). *If a learner delivers a message m, then eventually all correct learners will deliver that message.*

Proof. Any delivered message m belongs to the decision B_k of some instance $batch_k$. Lemma C.1.1 implies that m is seen by a correct acceptor, who eventually abcasts it. The assertion follows from Theorem C.1.4 (Termination Validity). □

Theorem C.1.6 (Latency C2). *Assume the underlying Consensus algorithm satisfies Property C2. Then, in stable runs, a message abcast by a correct proposer is delivered by all correct learners in three communication steps.*

Proof. Assume that a correct proposer abcasts message m at time t. The leader will receive m by time $t + d$ and will propose $\mathcal{M} \ni m$ to some instance $batch_k$. The leader proposed in all instances $batch_1, \ldots, batch_k$ by time $t + d$, therefore, by Property C2, all these instances will decide on values proposed by the leader by time $t + 3d$. In particular, instance $batch_k$ will decide on $B_k = \mathcal{M} \ni m$, so all correct learners will deliver m by time $t + 3d$. □

Theorem C.1.7 (Latency C1). *Assume the underlying Consensus algorithm satisfies Property C1. In ordered stable runs, a message abcast by a correct proposer is delivered by all correct learners in two communication steps.*

Proof. Assume that a correct proposer abcast message m at time t. Since correct acceptors receive all proposer messages in the same order, they all receive m by time $t + d$, as, say, the k-the message. Therefore they propose to all instances $batch_1, \ldots, batch_k$ by time $t + d$. Since, in each of these instances $batch_i$, all correct acceptors propose the same set $\mathcal{M}_i$, they will all decide by $t + 2d$.

In instance $batch_k$, all correct acceptors proposed the same $\mathcal{M} \ni m$. Therefore, this instance will decide on $\mathcal{M} \ni m$. As a result, message m will be delivered by time $t + 2d$.□

C.2 Optimistic Generic Broadcast

We will first make some definitions. Each learner l builds its own relation "$\rightarrow$", which we also denote as "$\rightarrow_l$" if l is not obvious from the context. This relation changes over time,

```
1   when a proposer executes gbcast(m) do
2     broadcast m to the acceptors
3   when an acceptor sees m eventually do
4     gbcast(m)
5   when an acceptor receives m for the first time do
6     for all possible non-received messages m' conflicting with m do
7       first_{m,m'}.propose(m)
8     abcast(m)
9   when first_{m,m'}.decision(m) at a learner do
10    set m → m'
11  when a learner has not gdelivered m,
12        and has m → m' for all undelivered messages m' conflicting with m do
13    gdeliver_1(m)
14  task cycle resolution at a learner is
15    repeat forever
16      wait until adeliver(m)
17      wait until m has been gdelivered or
18                  all undelivered messages conflicting with m are blocked
19      if m has not been gdelivered yet then
20        gdeliver_2(m)
```

Figure C.2: Optimistic Generic Broadcast.

which might lead to confusion in proofs. To avoid this, we assume that, unless explicitly said otherwise, the symbol "$\rightarrow$" represents the ultimate form of the relation, that is, the union of the relations $\rightarrow$ taken at all moments in time. If $m \rightarrow m'$, then we say that m is a *predecessor* of m' and m' is a *successor* of m. A finite *path* $m \rightsquigarrow m'$ is a sequence of messages $m = m_1 \rightarrow m_2 \rightarrow \cdots \rightarrow m_k = m'$.

Lemma C.2.1. *If $m \rightarrow m'$, then m' has been seen by a correct acceptor.*

Proof. Relation $m \rightarrow m'$ requires that Consensus instance $first_{m,m'}$ decided on m, so Lemma C.1.1 implies the assertion. □

Lemma C.2.2. *Any message m seen by a correct acceptor has a finite number of predecessors.*

Proof. The Validity property of the underlying Consensus algorithm implies that for any predecessor m', at least one acceptor must have executed $first_{m,m'}.propose(m')$. Therefore, at least one acceptor received m' before m. We have to prove that there are only finitely many such messages m'.

Message m has been seen by a correct acceptor, so all correct acceptors eventually receive m. Therefore, any correct acceptor receives only finitely many messages m' before

m. Incorrect acceptors receive finitely many messages before they crash. Therefore, the total number of messages m' that precede m is finite. □

Lemma C.2.3. *Let m be a message seen by a correct acceptor. Eventually, $m \rightarrow m'$ or $m' \rightarrow m$ for any m' conflicting with m.*

Proof. The assumption implies that all correct acceptors will eventually receive m. Therefore, all correct acceptors will eventually propose in all instances $first_{m,m'}$ (this happens when the acceptor receives its first message in $\{m, m'\}$). Eventually, all such instances will decide, which implies the assertion. □

Lemma C.2.4. *A never-delivered message m seen by a correct acceptor has a never-delivered predecessor.*

Proof. To obtain a contradiction, assume that every predecessor of m will eventually be delivered. Message m has a finite number of predecessors (Lemma C.2.2), so eventually all predecessors of m will be delivered. Lemma C.2.3 implies that eventually every message m' conflicting with m will either be its predecessor or successor. As a result, eventually all undelivered messages conflicting with m will be its successors, so m will be 1-delivered. This contradicts the assumption of m never being delivered. □

Lemma C.2.5. *All paths $m_1 \leftarrow m_2 \leftarrow m_3 \leftarrow \cdots$ are finite.*

Proof. Assume that the path $m_1 \leftarrow m_2 \leftarrow m_3 \leftarrow \cdots$ is infinite. Properties of the leader elector Ω ensure that eventually all acceptors will either crash or output a single acceptor a as the leader. Let M be the (finite) set of messages received by any acceptor before this happens. Consider an (infinite) tail $m_k \leftarrow m_{k+1} \leftarrow \cdots$ of the original path that does not contain any messages from M. Since all messages m_k, m_{k+1}, ... were received after the output of the leader elector stabilized, all relations $m_k \leftarrow m_{k+1} \leftarrow \cdots$ are consistent with the linear order of message reception at the eventual leader (Consensus Property **C2**) However, this means that the eventual leader received infinitely many messages before m_k, which is impossible. □

Lemma C.2.6. *If a correct learner executes adeliver(m), then it will deliver m.*

Proof. To obtain a contradiction, assume that a correct learner adelivered m but will never deliver it. Message m has been adelivered so it must have been seen by a correct acceptor. As a result, Lemma C.2.4 implies that m has a never-delivered predecessor.

We will prove that m has a never-delivered predecessor m' that is never blocked. To obtain a contradiction, assume that all never-delivered predecessors m' will eventually be blocked. This implies (previous paragraph) that at least one of the predecessors of m is blocked, which implies that m itself is blocked. As a result, all successors of m are

blocked as well. Therefore, Lemma C.2.3 implies that eventually all undelivered messages conflicting with m will be blocked, so m will be 2-delivered. This contradicts the assumption of m never being delivered, and proves that m has a never-delivered, never-blocked predecessor m'.

Consider the set of all paths $m'' \rightsquigarrow m'$ consisting only of never-delivered messages. There is at least one such path ($m' \rightsquigarrow m'$). Lemma C.2.5 implies that all such paths are finite, so there is a maximal path $m'' \rightsquigarrow m'$ (otherwise we could keep extending any path ad infinitum).

Both messages m' and m'' have been seen by a correct learner because they have a successor in the path $m'' \rightsquigarrow m' \rightarrow m$ (Lemma C.2.1). As a result, Lemma C.2.4 implies that m'' has a never-delivered predecessor m'''.

If $m''' \in m'' \rightsquigarrow m'$, then m''' is blocked because $m''' \rightarrow m'' \rightsquigarrow m'''$ forms a cycle, and as a result m' is blocked as well, which contradicts the assumption that m' is never blocked. On the other hand, if $m''' \notin m'' \rightsquigarrow m'$, then the path $m''' \rightarrow m'' \rightsquigarrow m'$ contains only never-delivered messages and extends $m'' \rightsquigarrow m'$, which contradicts the maximality of $m'' \rightsquigarrow m'$. These contradictions prove the assertion. □

Theorem C.2.7 (Termination Validity). *If a correct proposer gbcasts a message m, then all correct learners will eventually deliver it.*

Proof. The assumption implies that all correct acceptors will eventually receive m and execute $abcast(m)$. Therefore, all correct learners will eventually execute $adeliver(m)$, so Lemma C.2.6 implies the assertion. □

Theorem C.2.8 (Termination Agreement). *If a learner delivers m, then all correct learners will eventually deliver m.*

Proof. Any delivered message must have been seen by a correct acceptor, which will eventually gbcast it. Termination Validity implies the assertion. □

Theorem C.2.9 (Validity). *A learner delivers m only once and only if m was gbcast by some proposer.*

Proof. No message can be delivered twice because delivery of a message requires it not to have been delivered before. If m is 1-delivered, then $first_{m,\perp}$ decides on m, which implies that some acceptor proposed m, which implies the assertion. If m is 2-delivered, then it must have been abcast by some acceptor, which also implies the assertion. □

C.2.1 Partial Order

Definition C.2.10. *A learner "(1-,2-)delivers message m before m'" iff it (1-,2-)delivers m without having previously delivered m'. (Message m' can be delivered later in any way or not be delivered at all.)*

Lemma C.2.11. *Assume that learner l 2-delivers m before m' and learner l' 2-delivers m' before m. This is impossible.*

Proof. At the time of 2-delivery of m at l, message m' is not delivered. Therefore, learner l adelivers m before m'. By a similar argument, learner l' adelivers m' before m. This violates the Agreement property of the underlying Atomic Broadcast protocol. □

Lemma C.2.12. *Let m and m' be conflicting messages. Assume that learner l 1-delivers m before m' and learner l' 1-delivers m' before m. This is impossible.*

Proof. In order to 1-deliver m before m' at learner l, we must have $m \rightarrow m'$. An analogous argument for learner l', leads to $m' \rightarrow m$, which violates Agreement of the underlying Consensus algorithm. □

Lemma C.2.13. *Let m and m' be conflicting messages. Assume that some learner l 2-delivers m before m', and learner l' delivers m' before m. This is impossible.*

Proof. Consider the moment when learner l executes $deliver_2(m)$. Let B be the set of undelivered messages blocked at l. This set contains all undelivered messages conflicting with m. We will prove that, at any learner l', no message from B will be delivered before m.

To obtain a contradiction, assume that B is not empty and that l' delivers the first message $m' \neq m$ from B before m. We shall now obtain a contradiction by proving that learner l' can neither 2-deliver nor 1-deliver m' before m. Note that m and m' do not necessarily conflict.

Learner l' 2-delivering m' before m violates Lemma C.2.11 because learner l 2-delivers m before m'.

Message $m' \in B$ is blocked at l, therefore it has an undelivered, blocked predecessor m'' at l. In other words, there is $m'' \in B$ such that $m'' \rightarrow_l m'$. Note that m' is the first message in B delivered by l'; thus, at the moment of 1-delivery of m', message $m'' \in B$ is still undelivered at l'. This leads to a contradiction: 1-delivery of m' requires $m' \rightarrow_{l'} m''$, which is impossible because $m'' \rightarrow_l m'$. □

Theorem C.2.14 (Generic Agreement). *For any two* conflicting *messages m and m', it is impossible that one learner delivers m without having previously delivered m', and another learner delivers m' without having previously delivered m.*

Proof. To obtain a contradiction, assume that learner l delivers m before m', and learner l' delivers m' before m. If learner l 2-delivers m before m', then Lemma C.2.13 prevents learner l' from delivering m' before m. Therefore, learner l 1-delivers m before m'. By an analogous argument, learner l' 1-delivers m' before m. However, this is impossible by Lemma C.2.12. □

C.2.2 Latency

Lemma C.2.15. *In stable runs, if the leader receives a message at time t, then all correct learners will deliver it by time $t + 2d$.*

Proof. To obtain a contradiction, assume this is not true. Let m be the first message received by the leader for which the assertion does not hold.

Let m' be any message conflicting with m that was not received by the leader before m (at time t). When the leader receives m, it executes $\mathit{first}_{m,m'}.\mathit{propose}(m)$ at time t. Therefore, by Property **C2** of the underlying Consensus, by time $t+2d$, all correct learners have $\mathit{first}_{m,m'}.\mathit{decision}(m)$ and set $m \to m'$.

By assumption, all messages m' received by the leader before m were delivered before time $t + 2d$, therefore, at time $t + 2d$, the 1-delivery condition for m is met. □

Lemma C.2.16. *In conflict-ordered stable runs, any message received by all correct acceptors by time t will be delivered by time $t + d$.*

Proof. To obtain a contradiction, assume this is not true. Let m be the first message received by the leader for which the assertion does not hold.

Let m' be any message conflicting with m that was not received by the leader before m. By assumption, no correct acceptor receives m' before m. Therefore, all correct acceptors execute $\mathit{first}_{m,m'}.\mathit{propose}(m)$ by time t. As a result, Property **C1** of the underlying Consensus implies that, by time $t + d$, all correct learners have $\mathit{first}_{m,m'}.\mathit{decision}(m)$ and set $m \to m'$.

Let m' be any message conflicting with m received by the leader before m. By assumption, all correct acceptors receive m' before m and therefore before time t. By another assumption, m' will be delivered by time $t + d$. Therefore, at time $t + d$, the 1-delivery condition for m is met. □

Theorem C.2.17. *In stable runs, a message gbcast by a correct proposer is delivered by all correct learners within two steps if the run is conflict-ordered, and three steps otherwise.*

Proof. Straightforward from Lemmas C.2.15 and C.2.16. □

C.3 One-Two Consensus

Theorem C.3.1 (Agreement). *There is at most one x for which $\mathit{decision}(x)$ holds at some learner.*

Proof. In most cases, this follows from the same property of auxiliary instances of Consensus. This property can be violated only if some learner decides in condition 2, whereas

```
1   when acceptor a executes propose(x) do
2     broadcast(x, a)
3     propose_1(x)
4     propose_2(x, a)
5     propose_L(l) where l is the output of Ω
6   task decide at learners is
7     wait until decision_L(l)
8     wait until one of the conditions is true and decide on x
9       condition 1: decision_1(x) and receive(x, l)
10      condition 2: decision_2(x, l)
11      condition 3: decision_1(x) and decision_2(y, q) with q ≠ l
```

Figure C.3: The One-Two Consensus algorithm.

another does so in condition 1 or 3. Decisions in conditions 1 and 2 must be the same because they are both values proposed by the leader l. Conditions 2 and 3 cannot be used in the same execution. Condition 2 is used only if instance 2 decides on a value proposed by the leader, whereas condition 3 is used only if instance 2 decides on a value proposed by another acceptor. □

Theorem C.3.2 (Validity). *If decision(x) holds at some learner, then some acceptor proposed x.*

Proof. Follows from analogous properties of Consensus instances 1 and 2. □

Theorem C.3.3 (Termination). *If all correct acceptors propose, then eventually all correct learners will decide.*

Proof. Termination properties of the underlying instances of Consensus imply that eventually every correct learner will have $decision_L(l)$, $decision_1(x)$ and $decision_2(y, q)$. If $q = l$, then condition 2 will decide on y. Otherwise, condition 3 will decide on x. □

Theorem C.3.4 (Property C1). *In stable runs in which all correct acceptors proposed the same value, all correct learners decide on that value in one communication step.*

Proof. The assumption ensures that all correct acceptors, including the leader, will propose the same value to instances 1 and L. As a result, the leader l is known and condition 1 holds in one communication step [13]. □

Theorem C.3.5 (Property C2). *In stable runs, all correct learners decide on the value proposed by the leader in two communication steps after the leader proposed.*

Proof. The assumption ensures that all correct acceptors will propose the same leader to instance L, so all correct learners will decide on the leader l in one communication step. The leader l is correct, so its proposal (x, l) will become the decision in two communication steps [73]. As a result, condition 2 will hold. □

```
1   when an acceptor executes abcast(m) do
2     insert m into M
3     broadcast "active t" with t = current-time
4   task broadcasting at acceptors is
5     t_old ← t_0
6     repeat forever
7       M ← ∅
8       wait until received "active t" with some t > t_old
9       wait until current-time ≥ t
10      for all t' ∈ (t_old, t) do
11        messages_t'.propose(∅)
12      messages_t.propose(M)
13      t_old ← t
14      increase current-time
15  task delivery at learners is
16    t_old ← t_0
17    repeat forever
18      wait until messages_t'.decision([∅, ..., ∅]) for all t' ∈ (t_old, t) for some t > t_old
19      wait until messages_t.decision([M_1, ..., M_n])
20      for i = 1, 2, ..., n do
21        adeliver all undelivered messages in M_i in some deterministic order
22      t_old ← t
23  task retransmission at an acceptor is
24    periodically do
25      for all seen messages m do
26        abcast(m)
```

Figure C.4: Atomic Broadcast in closed groups.

C.4 Atomic Broadcast in closed groups

Theorem C.4.1 (Validity). *For any message m, every learner delivers m at most once, and only if m was abcast by a proposer.*

Proof. Learners deliver only undelivered messages, so the first part of the assertion holds. For the other part, note that the learner has $messages_t.decision([\mathcal{M}_1, \ldots, \mathcal{M}_n])$ for some t and $\mathcal{M}_k \ni m$. This means that $\mathcal{M}_k \neq \emptyset = \text{ABORT}$, so acceptor a_k executed $messages_t.propose(\mathcal{M}_k)$. This means that a_k must have executed $abcast(m)$. □

Theorem C.4.2 (Agreement). *For any two different messages m and m', it is impossible that one learner l delivers m without having previously delivered m', and another learner l' delivers m' without having previously delivered m.*

Proof. To obtain a contradiction, assume that the assertion does not hold. Let t be the first instance $messages_t$ which decides on a vector $[\mathcal{M}_1, \ldots, \mathcal{M}_n]$ with one of the sets $\mathcal{M}_k$

containing m. Similarly, let t' be the first instance $messages_{t'}$ which decides on a vector of sets $[\mathcal{M}'_1, \ldots, \mathcal{M}'_n]$ with one of the sets $\mathcal{M}'_{k'}$ containing m'. The Agreement property of Interactive Consistency implies that t and t' are defined in an unambiguous way.

Learner l delivers m without having previously delivered m', which implies $t \leq t'$. Learner l' delivers m' without having previously delivered m, which implies $t' \leq t$. As a result, $t = t'$, so both learners deliver m and m' while delivering the same batch of messages in $[\mathcal{M}_1, \ldots, \mathcal{M}_n]$. All these messages are delivered in the same deterministic order, which implies the assertion. □

Lemma C.4.3. *If a correct acceptor broadcasts "active t", then all correct acceptors will eventually have $t_{old} \geq t$.*

Proof. To obtain a contradiction, assume that at some correct acceptor a, the statement $t_{\text{old}} < t$ holds forever. The value of t_{old} can be updated to $t' < t$ only after receiving message "active t'" with $t' < t$. The number of messages broadcast before time t is finite, therefore, so is the number of messages "active t'" with $t' < t$. As a result, t_{old} is updated only finitely many times, and eventually it reaches some value $t_{\text{old}} < t$, which never changes again. However, we assume that a correct acceptor broadcasts "active t", therefore acceptor a will eventually receive "active t". Since, $t > t_{\text{old}}$, this will eventually lead to updating the value of t_{old}, which contradicts the assumption that t_{old} is never updated. □

Theorem C.4.4 (Termination Validity). *If a correct proposer abcasts message m, then all correct learners will eventually deliver it.*

Proof. Three events will eventually occur. Firstly, all faulty acceptors will crash. Secondly, the Eventual Weak Accuracy of $\Diamond S$ implies that there is a correct acceptor a_k that eventually will never be suspected by any correct acceptor. Thirdly, the assumption implies that a_k will eventually receive m, and therefore see it. The *retransmission* task ensures that acceptor a_k will execute $abcast(m)$ at least once after the three above events occurred.

This execution of $abcast(m)$, which happens, say, at time t_m, results in "active t_m" being broadcast by a_k. Lemma C.4.3 states that all correct acceptors, will eventually have $t_{\text{old}} \geq t_m$, which implies two facts. Firstly, all correct acceptors propose in all instances $messages_t$ with $t \in (t_0, t_m]$. Secondly, by executing $abcast(m)$, acceptor a_k inserts m into $\mathcal{M}$. At that time $t_{\text{old}} < \textit{current-time} = t_m$, so a_k will indeed propose $\mathcal{M} \ni m$ in some instance $messages_{t'}$ with $t' \in (t_{\text{old}}, t_m]$.

We have just shown that all correct acceptors propose in all instances $messages_t$ with $t \in (t_0, t_m]$ and that acceptor a_k proposed an $\mathcal{M} \ni m$ in one of them ($messages_{t'}$). As a result, the "delivery" task will eventually decide in all these instances and deliver the messages it decided on. Since no correct acceptor suspects a_k, the Sensitive Validity

condition of $messages_{t'}$ guarantees that its decision vector $[\mathcal{M}_1, \ldots, \mathcal{M}_n]$ satisfies $m \in \mathcal{M}_k$. Therefore m will be delivered by all correct learners. □

Theorem C.4.5 (Termination Agreement). *If a learner delivers m, then all correct learners will eventually deliver m.*

Proof. By assumption, some learner has executed $message_t.decision([\mathcal{M}_1, \ldots, \mathcal{M}_n])$ with $m \in \mathcal{M}_k$ for some t and k, so Lemma C.1.1 implies that a correct acceptor has seen m. This acceptor will eventually execute $abcast(m)$ in line 26, so Termination Validity (Theorem C.4.4) implies the assertion. □

Theorem C.4.6 (Latency). *In good runs with synchronized clocks, every abcast message is delivered by all correct learners within two communication steps.*

Proof. Assume that acceptor a_k abcasts message m at time t_m. Let $\mathcal{A}$ be the set of acceptors that abcast any messages at time t_m. Note that $\mathcal{A} \neq \emptyset$ because $a_k \in \mathcal{A}$. Each acceptor in $\mathcal{A}$ broadcasts an "active t_m" message at time t_m. As a result, all acceptors in $\mathcal{A}$ receive "active t_m" at time t_m, and all the others do so by time $t_m + d$.

Consider any acceptor $a \in \mathcal{A}$. Local messages incur no delay, so a_k receives its own message "active t_m" still at time t_m, and sets $t_{\text{old}} \geq t_m$. The assumption of clock synchronization implies that no acceptor has sent any "active t" with $t > t_m$ yet, so in fact $t_{\text{old}} = t_m$. Before abcasting m, acceptor a had $t_{\text{old}} < \textit{current-time} = t_m$, which implies that a_k executes line $messages_t.propose(\mathcal{M})$ in line 12 at least once after abcasting m. Consider the first such execution. Since $\mathcal{M}$ has not been emptied since a_k abcast m, we have $m \in \mathcal{M}$. We have shown that a_k executes $messages_t.propose(\mathcal{M} \ni m)$ with some $t \leq t_m$ at time t_m.

Consider now any acceptor a, not necessarily in $\mathcal{A}$. Since the set $\mathcal{M}$ is emptied after being proposed, the previous paragraph also proves that if an acceptor a proposes a non-empty $\mathcal{M}$ to $messages_t$, then it must have executed $abcast(m)$ at time t. As a result, any execution $messages_t.propose(\mathcal{M} \neq \emptyset)$ happens at time t. In other words, all executions of $messages_t(\mathcal{M} \neq \emptyset)$ with $t \leq t_m$ happen at time t_m or before.

Since acceptors in $\mathcal{A}$ broadcast "active t_m" at time t_m, all acceptors receive this message by time $t_m + d$. As a result, by time $t_m + d$, all acceptors propose in all instances $message_t$ with $t \leq t_m$.

Consider any instance $messages_t$ with $t \leq t_m$. The previous two paragraphs proved that all proposals are issued by time $t_m + d$, and non-empty ones are issued by time t_m. Since the run is good, the Latency property of Interactive Consistency implies that all instances $messages_t$ with $t \leq t_m$ will decide by time $t_m + 2d$. As a consequence, all messages in the decision vector in instances $messages_t$ will be delivered by time $t_m + 2d$.

We already showed that every message m abcast at time t_m was proposed to some instance $messages_t$ with $t \leq t_m$. Since the run is good, the Validity property of Interactive

Consistency implies that m will be in the decision vector of $messages_t$. Therefore, as the previous paragraph showed, m will be delivered by time $t_m + 2d$. □

Glossary

active timeframe

A timeframe in which at least one acceptor executes *abcast*.

anti-stable predicate

A time-dependent predicate is *anti-stable* if, once it is false, it will remain false forever.

blocked message

A message is *blocked* if, in the "$\rightarrow$" relation, it belongs to a cycle or it is a successor of a blocked message

communication step

The unit of time equal the maximum (supremum) message transmission time d between correct processes, in a given *time metric*.

complete state

The (state of a) learner is *complete* if all *correct* acceptors have executed *stop* and the learner has received all messages sent by these acceptors before or by their (first) *stop* action.

conflict-free run

A run is *conflict-free* iff no conflicting messages are gbcast.

conflict-ordered run

A run is *conflict-ordered* iff correct acceptors receive all conflicting proposer messages in the same order.

decide

A learner *decides on* x if *decision*(x) holds at that learner. A learner *decides* if it decides on some x. An algorithm *decides* or *terminates* when all correct learners have decided.

good run

A run is *good* iff it is timely and all acceptors are correct.

halt

A process that *halts* stops participating in the algorithm, usually after making a decision.

latency

The time that passes from the beginning of an algorithm run to its end, usually measured in communication steps d.

leader

A process with the following property: if the run is timely and the leader is correct, then learners decide on the value proposed by the leader. In OTC-based algorithms, this is the coordinator c_1 of the first round.

liveness property

A property is a *liveness property* iff, at any point in time, no matter what happened up to that point, it is still possible for that property property to hold.

ordered run

A run is *ordered* iff correct acceptors receive all proposer messages in the same order.

run

An execution of a particular algorithm.

safety property

A property is a *safety property* iff, at any point in time, if that property does not hold, then it will never hold, no matter what happens.

seen message

An acceptor *sees* a message m if it receives any proposal containing m (not necessarily directly). A message is *seen* iff a correct acceptor has seen it.

semi-complete state

The (state of a) learner is *semi-complete* if $\mathit{possible}(x) \implies \mathit{valid}(x)$ for all x, and $\mathit{possible}(x)$ holds for at most one x.

stable predicate

A time-dependent predicate is *stable* if, once it is true, it will remain true forever.

stable run

A run is *stable* iff it is timely, and all acceptor perceive the same leader, which is correct and never change.

time metric

A function t from events to real numbers, such that if an event e causally precedes e', then $t(e) \leq t(e')$. Examples of time metrics include real time and the logical time introduced by Lamport [78].

timeframe

Section 5.4 divides the continuous real time into discrete *timeframes* of length δ each. Timeframes are denoted by the times at which they begin, that is, timeframe t starts at time t and ends at time $t + \delta$. The first timeframe is t_0, the next is $t_0 + \delta$, then $t_0 + 2\delta$, etc.

timely run

In the eventual synchrony model, a run is *timely* iff all messages between correct processes have "sufficiently small latencies". In the failure detector model, a run is *timely* iff no correct acceptors are ever suspected.

Index