A Quick Consensus

by

Andrew Lewis-Pye

April 2026

Contents

Chapter 1

Introduction

1.1 The aim of this book

When I started working in distributed computing for blockchain, I found it difficult to get into the area. This was not because the proofs are especially complicated; most of the arguments in this book require only elementary reasoning. The difficulty is rather that there is a labyrinth of different possible setups and assumptions, and that results are extremely sensitive to the precise details of these setups. How reliable is communication between participants? What kinds of misbehaviour must the protocol tolerate? What exactly are the participants trying to agree on, and how many of them might be faulty? Changing the answer to any one of these questions can flip a problem from solvable to unsolvable, or vice versa. To make matters worse, different papers in the literature often use incompatible terminology and subtly different definitions, making it hard to see how results relate to one another.

What I wanted, as a newcomer to the area, was a clear narrative to walk me through all of these results and the relationships between them. My aim in writing this book has been to construct such a narrative: one that will take the reader quickly and easily through the essentials of distributed computing for blockchain, building up the theory in a logical order, with each new concept motivated by what has come before. The emphasis throughout is on clarity and accessibility rather than encyclopaedic coverage.

1.2 Who this book is for

The book does not assume any background knowledge beyond the occasional use of basic mathematical techniques, such as proof by induction, proof by contradiction, elementary counting arguments, and basic probability theory. It will undoubtedly be easier to read, however, for those with some background in computer science. Reasoning carefully about the step-by-step execution of a precisely defined procedure is a skill that can take some getting used to. The book is primarily aimed at undergraduates or postgraduates with some background in computer science or mathematics.

That said, I hope even experienced researchers and practitioners will find aspects of the presentation useful. Part of the aim has been to tidy up certain aspects of the

literature that are not always treated with full rigour. For example, the distinction we draw between State Machine Replication and Total Order Broadcast (Chapter 7), the formal treatment of Extractable SMR (Chapter 13), and the use of oracles to cleanly model cryptographic primitives (Chapter 12) are intended to clarify issues that can be a source of confusion, even for those familiar with the area.

1.3 What is missing

This is a short book, and there is inevitably a great deal that has been left out. To keep the exposition focused, I have not attempted to survey the vast literature on distributed computing, nor to describe every important protocol or result. Instead, I have tried to select a core set of ideas that, taken together, provide a solid foundation for understanding modern consensus protocols for blockchain.

One large and deliberate omission is that, outside Chapter 22, the book focuses on *permissioned* systems, in which one has a fixed and known set of participants. *Permissionless* systems, such as Bitcoin's proof-of-work protocol or Ethereum's proof-of-stake mechanism, allow arbitrary and unknown participants to join and leave. These systems raise a host of additional challenges, including Sybil resistance, incentive compatibility, and the analysis of protocols under economic assumptions.

The justification for focusing on the permissioned setting is that understanding the computer science behind permissioned systems suffices to understand most (though certainly not all) of the techniques required for the analysis of permissionless systems. The core consensus mechanisms at the heart of permissionless protocols are, in essence, permissioned protocols augmented with additional machinery. At the very least, a thorough understanding of permissioned consensus is a necessary first step before tackling the permissionless case.

1.4 Structure and a fast path

The book can be divided into four parts.

The first part (Chapters 2–6) works in the simplest possible communication model, in which messages between processors are delivered with a known and fixed delay. We use this simple setting to introduce the basic problems of consensus (such as getting all processors to agree on a single value despite the presence of faulty participants), to set up the formal framework, and to establish the fundamental impossibility and possibility results that underpin the rest of the book. We also describe two protocols in this setting: the Dolev-Strong protocol (Chapter 5) and the Phase-King protocol (Chapter 6).

The second part (Chapters 7–10) introduces the problem that blockchain protocols must actually solve, called State Machine Replication (SMR). Roughly, this asks processors to agree on a shared, growing sequence of transactions, even when some processors are faulty. We also move from the simple communication model used in the first part to more realistic models, in which message delays are not known in advance, and establish what is and is not solvable in these settings. We then describe and analyse Tendermint

(Chapter 9), one of the simplest protocols for this more realistic setting, and define formal metrics for comparing the performance of SMR protocols (Chapter 10).

The third part (Chapters 11–16) describes and analyses a series of increasingly sophisticated consensus protocols. These include PBFT (Chapter 11), which introduces an important alternative approach to leader changes; Hotstuff (Chapter 13), which achieves improved communication efficiency; and Simplex (Chapter 15), whose design underpins a number of recent protocols. Chapter 12 introduces a formal technique for cleanly modelling cryptographic tools within the framework. Chapter 14 addresses the problem of view synchronisation, which is needed to coordinate leader changes across processors. Chapter 16 develops protocols for the fully asynchronous setting, in which no bound on message delays is assumed.

The fourth part (Chapters 17–24) covers a selection of more advanced topics: techniques for reducing communication costs using erasure codes (Chapter 17), payment systems (Chapter 18), DAG-based protocols (Chapter 19), accountability (Chapter 20), recovery from faults (Chapter 21), changing the set of participants over time (Chapter 22), protocols achieving two-round finality (Chapter 23), and a model for formally analysing throughput and latency (Chapter 24).

A summary of the notation used throughout the book can be found in Appendix A.

A fast path for the protocol-minded reader. A reader whose primary interest is in understanding how consensus protocols work, rather than in the details of impossibility proofs, may wish to take the following path. In Chapters 2–6, read all definitions and the descriptions and correctness proofs of the two protocols (the Dolev-Strong protocol in Chapter 5 and the Phase-King protocol in Chapter 6), while simply noting the statements of the impossibility results without studying their proofs. Then read Chapter 7 in full, followed by Chapter 8, again noting the statements of impossibility results without dwelling on the proofs. From Chapter 9 onwards, the focus shifts predominantly to protocol design, and the reader can proceed through the remaining chapters in order.

Chapter 2

A simple consensus problem

Many modern computing systems involve multiple independent computers that must work together. A blockchain network, for example, consists of thousands of computers (often called *nodes*, or *validators*) that must agree on a shared record of transactions, even though some nodes might be controlled by attackers attempting to corrupt that record or halt the system. A fleet of autonomous vehicles might need to coordinate their movements, even though some vehicles' computers might be compromised. In settings like these, we need protocols (precisely defined communication procedures) that can maintain agreement despite the presence of participants that behave in arbitrary and potentially malicious ways. To understand how such protocols work, we begin by introducing a classic problem that captures the essence of this challenge: Byzantine Agreement.

The Byzantine Agreement problem (informal version). As described by Lamport, Shostak and Pease [1, 2], who introduced the problem in the early 1980s, the problem to be solved is as follows. Several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. After observing the enemy, each general has their own private opinion as to the best plan of action, either 'retreat' or 'attack'. The generals can communicate with one another only by messenger and have to carry out a protocol (a precisely defined communication procedure) to decide on a common plan of action. The protocol should work whatever their initial opinions. The difficulty is that some unknown subset of the generals may be dishonest traitors. While honest generals will carry out the protocol exactly as prescribed, the dishonest traitors may deviate from the protocol. We have to design the protocol to meet the following conditions:

1. **Termination.** Each honest general must eventually reach a decision (retreat or attack).
2. **Agreement.** No two honest generals can reach different decisions.
3. **Validity**. If all honest generals start with the same opinion, then that common opinion must be the same as their final decision.

Some simple observations. The description of the Byzantine Agreement problem above is informal. To reason about it formally, we'll need to give precise answers to questions such as:

- How exactly does messaging work? More specifically, do messages always arrive, and how long does delivery take? Can generals be sure who messages are from?
- How exactly can dishonest generals behave?

In fact, we will see that the circumstances in which it is possible to define a successful protocol are very sensitive to the precise way in which we answer these questions. Even with this informal version of the problem, though, there are certain observations that can be made. We enumerate some below.

(i) **It's easy without the Validity requirement**. Without the requirement for Validity, all honest generals could just ignore their initial opinion and always decide to attack.

(ii) **It's easy if all generals are honest.** To understand a problem better, one can start by testing boundary cases. Suppose that there are n generals of which *at most* f are dishonest. For clarity, let us suppose the honest generals know n and f (but not which generals are dishonest, or the exact number). Note that, if $f = 0$ and messages always arrive as intended, then the problem is trivial. For example, we can just have each general communicate their initial opinion to the others and then decide according to majority vote (with ties broken in favour of 'retreat', say). So, it's clear that one of the basic questions we should be interested in is, "for what values of n and f is a successful protocol possible?".

(iii) **The case $f \geq n/2$ is impossible.** It's also easy to see that no protocol can work if $n \geq 2$ and $f \geq n/2$. To see this, consider the case $n = 2$. Then the reader can extend the argument to the general case (see Exercise 2.1). Suppose we have a protocol which works when $f = 1$. Without loss of generality, let us suppose that, when the first general has initial opinion 'attack' and the second has initial opinion 'retreat', the protocol causes both generals to decide to attack if they behave honestly (a similar argument will apply if they both decide to retreat). Now consider what happens when both generals have initial opinion 'retreat' but the first is dishonest and behaves during the protocol exactly like they should if honest with initial opinion 'attack', while the second general behaves honestly. In this case, the protocol must cause both generals to decide to attack, just as in the first case considered, even though all honest generals started with the initial opinion 'retreat'. This contradicts the Validity requirement.

Is the problem trivial when $f < n/2$? We saw above that the problem is trivially *not* solvable when $f \geq n/2$. It's also tempting to think that the problem might be trivially solvable when $f < n/2$. For example, can we not just implement a majority vote argument of the sort described in paragraph (ii) above? The problem with this approach stems from the allowed form of communication between generals, which is intended to accurately reflect communication between computing devices in real-world scenarios. If the generals were standing in a circle and shouting out their votes – so that everybody can see who is shouting out a vote and any vote heard by a single honest general is immediately heard by all – then a simple majority vote approach could work. However, in the setting described in the problem formulation, communication occurs only by messenger between one pair of generals at a time. The problem now is that, even if message delivery is perfectly reliable, dishonest generals can tell different things

to different generals. For example, suppose that $n = 3$, call the generals A, B, and C, and suppose that A and B are honest, while C is dishonest. Suppose A initially wants to attack, while B wants to retreat. If we carry out a simple majority vote protocol as described above, and if C sends an 'attack' vote to A and a 'retreat' vote to B, then the honest generals will see different majority votes: while A sees a majority of votes to attack (including their own), B sees a majority of votes to retreat. This means the honest generals will decide differently, violating Agreement.

In fact, we will see that (under a natural formalisation of the informal problem above) the Byzantine Agreement problem is not solvable when $f \geq n/3$ unless we endow the generals with certain extra abilities. So, the problem is not as trivial as it might initially seem. However, to establish impossibility results of this kind, we certainly need a formal framework. So, that is what we will set up in the next chapter.

2.1 Exercises

Exercise 2.1. *In (iii) above, we gave a simple 'proof' (for this informal setting) that no protocol will work if $f \geq n/2$, but only for the case $n = 2$. Extend the argument to handle all $n \geq 2$. What happens for the case $n = 1$?*

Chapter 3

The formal framework

The decisions we have to make in formally defining the setup include such things as how long messages take to arrive (and whether they are certain to arrive), and what sort of behaviour the dishonest generals are capable of. We will see that whether it is possible to define a working consensus protocol is very sensitive to these choices. In fact, the main difficulty when studying consensus protocols is not that the proofs are particularly tricky. The difficulty is rather that the results are very sensitive to the precise details of the setup and the particular problem posed, be it the Byzantine Agreement problem or some variant. This means one has to keep track of a large number of different possible setups, problems, and the different corresponding results.

We will formalise protocol participants as computing devices, referred to as *processors*.[1] One aspect of the setup which won't impact our results, so long as one restricts to 'reasonable' options, is the choice of computational model. Since it is probably the simplest option, we will take the standard approach of modelling processors using state-transition-diagrams (specified below). If you prefer to think in terms of interactive Turing machines, or another standard model of computation, that's fine. You can make the appropriate modifications to the model we describe below: all of the proofs will go through essentially unchanged.

3.1 Processors

The processors. We consider a set of n *processors* $\Pi = \{p_0, \ldots, p_{n-1}\}$.

Time. Time is divided into discrete time-slots $t = 0, 1, 2, \ldots$.

Authenticated channels. Messages are finite binary strings. There exists a two-way communication channel, called 'channel $\{i, j\}$', for each pair of distinct processors p_i and p_j. The communication channel $\{i, j\}$ is *authenticated* in the sense that:

- Only p_i can send messages to p_j and only p_j can send messages to p_i on the channel $\{i, j\}$, and;

[1] The use of the term 'processor' is fairly standard. Many papers use the terms 'replica', 'process', or 'node' instead.

- When p_i receives messages it is aware of the name of the channel by which the messages were sent, i.e., as we'll make precise below, the instructions for p_i can depend not only on the messages received at any given time-slot but also which channels the messages arrived on.

Processors are specified by state-transition-diagrams. A state-transition-diagram is simply a precise description of how a processor should behave. The idea is as follows. At any given moment, a processor is in some *state*. We can think of the state as encoding everything the processor currently 'knows' or 'remembers'. At each time-slot t, the processor begins in some state x, and then receives a set of messages on each of its communication channels $\{i, j\}$. Given x and a specification of which set of messages has arrived on each channel $\{i, j\}$ at t, the state-transition-diagram then determines: (a) the message (if any) that p_i sends along each channel $\{i, j\}$ at t, and (b) the state in which p_i begins time-slot $t + 1$. So, according to this model, the state-transition-diagram for each processor specifies the protocol. There is no requirement that the state-transition-diagram be finite, i.e., it may contain an infinite number of states.

Example 1: a simple echo. Suppose $\Pi = \{p_0, p_1\}$, and that the state-transition-diagram for p_0 has two states, A and B. While in state A, if p_0 receives the message 1 from p_1, then it sends 1 back to p_1 and moves to state B. Otherwise, it sends no messages and remains in state A. While in state B, it sends no messages and remains in state B, regardless of what it receives. If p_0 begins in state A at time-slot 0, the result is that p_0 will echo the message 1 back to p_1 if p_1 ever sends it, and will do nothing thereafter.

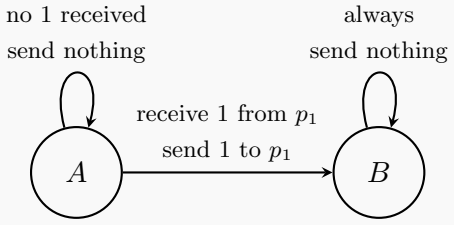


FIGURE 3.1: Example 1, a simple echo.

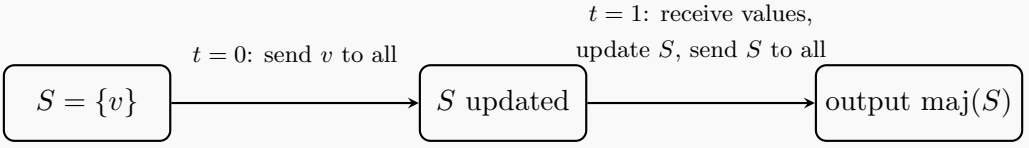
We note that, at a single time-slot, p_i can send different messages along different channels, but can only send one message along each individual channel. The latter requirement is not a significant restriction, since we have not limited the length of messages, so that multiple messages can be combined to form a single message.

The specification above defines *deterministic* processors: at each time-slot, a processor's state and the messages it receives (together with information as to which channels they arrived on) suffice to entirely specify which messages the processor sends at that time-slot and the state they are in for the next time-slot. We will assume that processors are deterministic, unless explicitly stated otherwise. If processors are *probabilistic*, then the present state and the messages received at a given time-slot specify a *distribution* on the messages sent at that time-slot and the state for the next time-slot.

Inputs and outputs. Certain states of the state-transition-diagram for p_i are labelled as 'input states', and p_i begins time-slot 0 in one of these input states. The input to p_i determines which input state it starts in. Both the number of processors n and also p_i's name 'p_i' are given to p_i as part of its input. Certain states are also labelled as 'output'

states. When processor p_i reaches such a state x at time t, its output is determined by both the state x and the messages it receives at that time-slot.

Example 2: collecting and forwarding. Suppose $\Pi = \{p_0, p_1, p_2\}$, and consider a simple protocol in which each processor sends its own input value to both other processors at time-slot 0. At time-slot 1, each processor receives the values sent by the others, and forwards all values it now holds (including its own) to both other processors. At time-slot 2, each processor receives these forwarded values and outputs the majority of all values it has seen (we can assume ties are broken according to some arbitrary fixed rule). The diagram below shows the key transitions for p_0. We write S for the multiset of values that p_0 has collected.



The key point illustrated by this example is that the state of a processor encodes its memory: everything it needs to determine its future behaviour. Recall that p_0 also receives n and its own name as part of its input; we suppress these in the diagram for clarity.

FIGURE 3.2: Example 2, collecting and forwarding.

Now that we've defined how processors compute and how they communicate, we need to specify the timing assumptions - how long do messages take to arrive, and are they guaranteed to arrive at all?

3.2 Message delays

There are a number of different possible assumptions regarding the reliability of message delivery. To keep things simple, we start by considering two straightforward variants:

- In the *lock-step* model, messages always arrive at the next time-slot: if p_i sends p_j a message at time-slot t then p_j receives that message from p_i at time-slot $t + 1$.
- In the *synchronous* model, there is some known and finite bound $\Delta \geq 1$, given as input to all processors, such that any message sent at any time-slot t arrives by time-slot $t + \Delta$ at the latest, i.e., the message arrives at some time-slot in the interval $[t + 1, t + \Delta]$.

A point of clarification is required with respect to the synchronous model: a message m should be received precisely once for each time it is sent. For example, if the only messages p_i sends are m to p_j at time-slot t and then (again) m to p_j at time-slot $t' > t$, then p_j must receive m on the channel $\{i, j\}$ (exactly) twice by time-slot $t' + \Delta$, and should not receive m at later time-slots. In particular, p_j may receive m from p_i twice *at the same time-slot*, so the messages received by p_j on a given channel at a given time-slot are really a *multi-set* (and the instructions specified by the state-transition-diagram depend on the multi-set of messages arriving on each channel at a given time-slot).

When we say that a protocol achieves a certain functionality in the synchronous model, we mean that this is the case for any value of $\Delta \geq 1$. In the first few chapters of this book (until Chapter 8), we will focus on the lock-step model. However:

> *All of the results we prove for the lock-step model will also hold for the synchronous model.*

This is the case because:

(a) Any impossibility result for the lock-step model necessarily holds for the synchronous model, since the lock-step model is a special case of the synchronous model with $\Delta = 1$, and;

(b) All of the protocols we consider for the lock-step model can straightforwardly be modified to handle the weaker assumption that messages arrive within time Δ by executing the same instructions only at time-slots that are multiples of Δ.

Since the lock-step and synchronous models may be unrealistic in many real-world scenarios, in Chapter 8 we will also formally define the *partially synchronous* and *asynchronous* models. Roughly, the partially synchronous model formalises the idea that message delivery will sometimes be reliable (with a bound on delivery times, as in the synchronous model), but that there may be periods of unbounded length during which message delivery is not reliable. In the asynchronous model, messages are always delivered,[2] but there is no bound on how long delivery may take. The good news, for those who like things simple, is that almost all papers in the literature use either the lock-step/synchronous, partially synchronous or asynchronous models.[3]

Our focus: If there is one of these four models that is most important in the context of direct real-world applications to 'blockchains', it is the partially synchronous model. Generally, one wants protocols that can handle unbounded periods of unreliable message delivery. On the other hand, the assumptions of the asynchronous model may be seen as unnecessarily weak in most real-world scenarios of interest. For that reason, when it comes to possibility results (describing protocols that achieve a certain functionality) this book will focus (somewhat) on protocols for the partially synchronous model. However, to develop the theory it is important to consider the lock-step model first. We'll describe a number of impossibility results for the lock-step model that carry over directly to the other models.

Some conventions and terminology for message sending. In many of the protocols we describe in this book, processors will be instructed to send certain messages 'to all processors'. As shorthand, we will use the term *disseminate* to mean 'send to all processors'. Formally, processors cannot send messages to themselves: if p_i disseminates m, this means that it sends m to all other processors, and also regards m as immediately received when interpreting future instructions. For example, consider a voting process. Suppose p_i disseminates a 'vote for x' and subsequently receives votes for x

[2]The assumption that messages are always delivered may seem strong. Roughly, protocols such as TCP achieve something close to this functionality, by using a 'lower level' protocol that repeatedly sends messages until delivery is acknowledged.

[3]Often these 'models' are also referred to as 'settings'. So, one may also refer to the 'synchronous setting', or the 'partially synchronous setting'.

from $n - f - 1$ other processors. In interpreting future instructions, p_i will regard itself as having received $n - f$ votes for x (counting its own). This simple convention is just used to simplify and streamline protocol descriptions.

Having established our basic model of computation and communication, next we specify precisely how some processors may deviate from correct behaviour.

3.3 Faults

Processors can either be *correct* or *faulty*. Correct processors have a state-transition-diagram as specified by the protocol, and carry out instructions exactly as it dictates. On the other hand, there are a number of different possibilities when it comes to allowable faulty behaviours. Generally, in the context of the 'blockchain' literature, we are most interested in analysing settings where some participants may be trying to 'attack' the protocol, potentially causing it to fail in various ways. Since we can't be sure what form these attacks will take, we therefore suppose that faulty processors can behave in an *arbitrary* fashion. As specified below, faults of this kind are called *Byzantine* faults.

The Byzantine fault-model. In the Byzantine fault model, faulty processors may follow any state-transition-diagram, not necessarily the one specified by the protocol. To model the fact that faulty processors may also *collude*, we also allow that the state-transitions and messages sent by faulty processors can depend on the messages received by other faulty processors: if p_i is faulty, then the messages that p_i sends at time-slot t and the state that it enters for time-slot $t + 1$ are a function of p_i's state at t and the multi-set of messages received by each faulty processor on each of their channels at time-slot t. When considering Byzantine faults, it is common to refer to correct processors as *honest*, and faulty processors as *Byzantine*.

In this book, we will be most concerned with the Byzantine fault-model. We will sometimes consider other fault-models when proving impossibility results (when impossibility results are just as easily proved for a more restrictive fault-model, we may as well do so). The two other kinds of faults most commonly considered are *crash-faults* and *omission-faults*. Roughly, processors with crash-faults must behave correctly, except that they may stop executing instructions at any point of the execution. Processors with omission-faults may fail to send or receive some messages. While Byzantine faults model malicious attacks, crash-faults and omission-faults capture more common failure modes: processors that simply stop working (e.g., due to power failures) or network failures that cause message loss.

The crash-fault model. In this model, faulty processors have the state-transition-diagram specified by the protocol and execute instructions correctly, except that they may *crash* at some time-slot t. If p_i crashes at t, then it may send an arbitrary subset of the messages that it is instructed to send at t, and p_i then performs no further action at later time-slots (remaining in the state that it was in at time-slot t thereafter, while sending and receiving no further messages).

The omission-fault model. In this model, faulty processors have the state-transition-diagram specified by the protocol and execute instructions correctly, except that they may fail to send or receive messages. Specifically, at each time-slot, a processor with omission-faults may take an arbitrary subset of the messages that have arrived on each

channel, and will then act (at that time-slot) exactly like a correct processor that has received this (potentially reduced) set of messages, except that it may send an arbitrary subset of the messages that the instructions require it to send.

The adversary. Especially when considering Byzantine faults, it is common in the literature to talk in terms of an *adversary* who controls and co-ordinates all faulty processors so as to cause problems. For now, you can just see this as a linguistic device, which facilitates a useful way of thinking. Since protocols have to deal with arbitrary message delays within the constraints of the model considered (the synchronous, partially synchronous and asynchronous models give some leeway as to exactly when messages arrive), it is common also to think of the adversary as controlling message delivery (within the constraints of the model).

Static, adaptive, and mobile adversaries. Unless explicitly stated otherwise, we will consider *static* adversaries, and for now we only formally consider adversaries of this type. If the adversary is static, this means an unknown set of at most f processors are faulty from the start of the protocol execution, where f is a known bound. Roughly, *adaptive* and *mobile* adversaries work as follows:

- An adaptive adversary can choose up to f processors to *corrupt* (make faulty) over the course of the protocol execution. Processors behave correctly until corrupted, and then have behaviour dictated by the fault-model (Byzantine, crash or omission-faults). The adversary may be able to choose which processors to corrupt based on the inputs of faulty processors and the messages received by all faulty processors thus far, or we may allow the adversary extra information (such as all inputs and the messages sent and received by correct processors).
- A mobile adversary can choose any number of processors to make faulty as the execution progresses, but at most f processors can be faulty at any given time-slot.

Generally, since we may think of the adversary as controlling message delivery, the distinction between static and adaptive adversaries is not interesting for deterministic protocols, but can be important for probabilistic protocols. All of our impossibility results for static adversaries necessarily carry over to the more general case of adaptive or mobile adversaries. All of our possibility results for static adversaries carry over to adaptive adversaries, unless explicitly stated otherwise.

Exercise 3.1. *In what sense is the distinction between static and adaptive adversaries not interesting for deterministic protocols?*

3.4 Protocols and executions

A *protocol* is just a specification of the state-transition-diagram that should be carried out by each p_i, if correct. An *execution* is a complete description of:

- The set of processors, their state-transition-diagrams, and their inputs;
- The state of each processor at each time-slot;

- The messages sent and received by each processor on each of their channels at each time-slot, and;
- In the context of the crash-fault-model, which processors crash at each time-slot.

The definition above is for the static adversary model; in the context of an adaptive or mobile adversary, it may sometimes be convenient to stipulate that executions specify further information.

Exercise 3.2. *(Some parts of this question are quite open-ended.) According to the framework we have described, a protocol may specify a different state-transition-diagram for each processor. Is this important? Is it important that p_i is given its name as part of its input? Would this change if each correct processor had to have the same state-transition-diagram? How about if processors were not aware of the name of the channel on which messages arrive?*

One final modelling choice significantly affects what protocols can achieve: whether processors can produce unforgeable signatures on their messages.

3.5 Signatures

Another important distinction is whether or not processors are able to produce unforgeable signatures for the messages they send, i.e., whether p_i is able to attach a signature to any given message that suffices to prove the message was produced by p_i. The use of authenticated channels might initially seem to make such a signature scheme redundant: each processor can already see who sent the messages it receives. The crucial point is that the availability of a signature scheme means no processor will be able to deceive others about what messages it has received from other processors.

In this book, we focus on the case that a signature scheme is available, since it is the case of most practical relevance. However, we will not assume a signature scheme is available unless explicitly stated otherwise. When we do make use of signatures, we entirely blackbox the necessary cryptography. Formally, when we consider a model *with signatures* this simply means that:

- For each $p_i \in \Pi$ and each message m, there exists a special message, denoted $\langle m \rangle_i$, to be thought of as the 'message m signed by p_i'.
- For each $p_i \in \Pi$ and m, processors other than p_i cannot send any message containing the sequence $\langle m \rangle_i$ until any time-slot at which they have received a message containing this sequence.[4] This means we restrict the set of allowable state-transition-diagrams (even for Byzantine processors) to those which obey this requirement.

[4]To model a fully colluding adversary, we can also allow a Byzantine processor p_i to send a message containing $\langle m \rangle_j$ before receiving any message containing this sequence in the case that p_j is also Byzantine, or if another Byzantine processor has already received such a message. Whether or not we make such an allowance will not impact any of our results here.

To see the significance of signatures, suppose p_1 wants to tell p_2 that it received message 'attack' from p_0. Without signatures, p_1 could lie. With signatures, p_1 must forward $\langle \text{attack} \rangle_0$, which only p_0 could have created.

An aside for those familiar with cryptography. As described above, we consider idealised signature schemes. Those familiar with how signature schemes are established in practice will know that, in reality, there are a number of subtleties. For example, a signature scheme will generally require some form of public-key-infrastructure to achieve proper functionality. Signatures must be efficiently produceable *and* verifiable (no special considerations concerning verifiability are necessary in our formulation, because there is nothing to stop state-transition-diagrams being able to recognise whether a signature is valid). Signature schemes require certain cryptographic assumptions, and one must also restrict to the case of polynomial-time-bounded adversaries. Even under all these restrictions, one has to accept a negligible chance of error in any given execution. We take the approach of using idealised signature schemes to avoid these complications. While one *could* recast all our arguments in the standard formal frameworks of cryptography, to do so would be a distraction here.

3.6 The research programme

Given the formal framework outlined above, the questions we now want to answer include:

- For which n and f do there exist protocols to solve Byzantine Agreement and other variants of the problem?
- How does this depend on our assumptions regarding message reliability, i.e., whether we are working in the lock-step/synchronous, the partially synchronous or the asynchronous model?
- Does the answer depend on whether we consider a model with signatures?

The order in which we address these issues is as follows. We begin in the simplest setting: the lock-step model with Byzantine faults. In the next three chapters, we determine for which n and f Byzantine Agreement is solvable in this model, first with signatures (Chapter 5) and then without (Chapter 6). In Chapter 7, we introduce State Machine Replication, which captures the problem blockchain protocols must actually solve. We then move to the partially synchronous and asynchronous models (Chapter 8), and determine what is solvable in each. The remainder of the book is devoted to describing and analysing increasingly sophisticated protocols, and to exploring further topics such as message complexity, responsiveness, and accountability.

Chapter 4

Byzantine Agreement and Byzantine Broadcast

In Chapter 2, we considered a version of the Byzantine Agreement (BA) problem which was binary in the sense that processors had to decide between two output options. While binary agreement captures the essence of consensus, many practical applications require agreement over larger domains. For example, nodes in a distributed database might need to agree on which of many possible updates to process next. Here is the more general version of the problem:

- We consider a set of n processors, of which at most f display Byzantine faults.
- For some set V, each processor is given an input in V (different processors potentially receiving different inputs). The set V is told to the processors and could be of any finite size ≥ 2.
- The protocol must satisfy the following conditions:
  - **Termination**. All correct processors must give an output in V.
  - **Agreement**. No two correct processors can give different outputs.
  - **Validity**. If all correct processors have the same input v, then v must be their common output.

On seeing the more general version of the problem specified above, it is natural to ask about the relationship to the previous binary version. Of course, any protocol which solves the more general form of the problem also gives a solution for the binary case. There are also reductions, such as that by Turpin and Coan [3], which show that *in certain settings* a protocol for the binary version of the problem can be used to solve the more general case. In this book, it will not be necessary to treat the two versions of the problem separately because:

- All impossibility results will apply to the binary version of the problem (as well as the general form).
- All protocols described for solving Byzantine Agreement will solve the general form (and so also the binary form).

An aside on Validity. A difficulty when interacting with the literature on consensus is that there is a lack of consensus on terminology. The condition for Validity we have stated above is sometimes referred to as 'Weak Validity', while the latter term also sometimes refers to the following requirement: if all processors are correct and have the same input value, then this common input must be the value output by each processor. There are also many other variants considered, with a range of names. For example, a version sometimes called 'Strong Validity' requires that any value output by a correct processor must have been input to at least one correct processor.

Byzantine Broadcast. In the original papers in which Lamport, Shostak and Pease introduced the Byzantine Agreement problem, they actually focussed on a variant of the problem which is now most frequently referred to as *Byzantine Broadcast* (BB). Byzantine Broadcast models scenarios where a designated party called the *broadcaster* (like a commanding officer or system coordinator) is required to reliably disseminate information to all participants, even if some participants (perhaps including the broadcaster) are malicious:

- We consider a set of n processors, of which at most f display Byzantine faults.
- One processor is designated the 'broadcaster'. All processors are told the name of the broadcaster.
- The broadcaster is given an input in some set V. The set V is told to all processors.
- The protocol must satisfy the following conditions:
  - **Termination**. All correct processors must give an output in V.
  - **Agreement**. No two correct processors can give different outputs.
  - **Validity**. If the broadcaster is correct and has input v, then all correct processors must output v.

The relationship between BA and BB. What is the relationship between the Byzantine Agreement and Byzantine Broadcast problems? In Chapter 2, we noted that BA cannot be solved when $f \geq n/2$. However, it is easy to see that the argument given there does not apply to BB (you are invited to check). In Chapter 5 we will see that, if signatures are available and we work in the lock-step model, then BB can actually be solved for any number of faulty processors. So, there are certainly scenarios in which BB can be solved although BA cannot be.

On the other hand, if we work in the lock-step model, and if $f < n/2$, then the two problems reduce to each other quite easily:

- If we can solve BB, then to solve BA we have all processors broadcast their inputs using the protocol for BB (meaning that we carry out n simultaneous executions of BB). Once a value is decided corresponding to each processor, processors then decide by majority vote, breaking ties in some previously arranged but arbitrary fashion.
- If we can solve BA, then to solve BB we have the broadcaster send their input to all other processors at time-slot 0. Each processor then takes the value received at time-slot 1 as their input value, choosing some arbitrary value in V if no value is received from the broadcaster. We then have the processors carry out the protocol for BA on those input values.

So far, it might seem that BB is strictly easier than BA. As a word of caution, we will later see that, in the partially synchronous and asynchronous models, it is not possible to solve BB if $f \geq 1$, although protocols do exist to solve BA. So, the two problems are not strictly comparable in a general sense.

Other fault models. When restricting to the crash-fault model, we'll call the corresponding version of BB, 'Crash-fault Broadcast'. We'll call the version with omission-faults, 'Omission-fault Broadcast'. Similarly, we will sometimes consider a version of BA with crash-faults, which we'll call 'Crash-fault Agreement', and we'll call the version with omission faults, 'Omission-fault Agreement'.

Chapter 5

The lock-step model with signatures

In this chapter, we consider the lock-step model with signatures. First, in Section 5.1, we prove a positive result, by introducing the elegant protocol for Byzantine Broadcast by Dolev and Strong [4]. This protocol dates back to the early 80s but remains an important tool for those developing the theory of consensus protocols, because it is really the only protocol that can deal with an arbitrary number of (Byzantine) faults. In Sections 5.2 and 5.3, we then prove two impossibility results, both of which establish some sense in which the Dolev-Strong protocol is optimal. A reader wishing to focus on techniques required for positive results may wish to omit the proofs in Sections 5.2 and 5.3, and just memorise these two results before skipping to Chapter 6.

5.1 The Dolev-Strong protocol

In this section, we'll prove the following theorem.

Theorem 5.1. *Consider the lock-step model with signatures. There exists a protocol that solves the Byzantine Broadcast problem for any number of faulty processors, i.e., for any $f \leq n$.*

Note that Theorem 5.1 also suffices to establish precisely when BA is possible for the lock-step model with signatures: by the reductions discussed in Chapter 4, and since we know BA is not possible when $f \geq n/2$ (see Chapter 2), BA is solvable iff $f < n/2$.

5.1.1 Informal discussion: why isn't it trivial?

Before describing the proof, let us explore why a trivial solution does not work. When signatures are available, one obvious way to try solving BB would be to have the broadcaster send out signed values of their input to each of the other processors. The processors could then repeatedly share all of the signed values they have seen produced by the broadcaster. If they only ever see a single value produced, then they output that value. If they ever see two different values produced (or no values), then they realise the

broadcaster is faulty, so they give some 'default' value as output. Of course, the idea behind this approach is that if the broadcaster is correct, then they will only produce a single signed value and all correct processors will output that value. If the broadcaster produces two different signed values (or no values) and shows them to correct processors then everyone will eventually see or 'recognise' that those signed values have been sent out by the broadcaster, and will give the default output.

Hopefully, the problem with this approach is quite clear. At what point should the processors stop sharing values and output? If they share until time-slot t, then the adversary can choose to show one signed value to all correct processors prior to time-slot t, and then show some subset of the correct processors a second signed value at time-slot t (when it is too late to share anymore), causing the Agreement requirement of BB to be violated. We could require processors to ignore (not to 'recognise') new values seen at the last time-slot t, but this only takes things one step back: now the adversary just has to show some subset of the correct processors a second signed value at time-slot $t-1$. It's a common reaction to feel that *some* sort of approach along these lines must work, and it's a good exercise to specify a few simple approaches and work out exactly why they don't work (or else realise that they do and that you have reinvented the Dolev-Strong protocol for yourself!).

The trick (informal). What we need is a clever mechanism to ensure that if any correct processor 'recognises' a certain signed value as being produced by the broadcaster, then all correct processors will also 'recognise' that value. That way, either they all recognise a single value and give that as output, or they all recognise (no values or) multiple values and so give the default output. The mechanism that Dolev and Strong used to achieve this works as follows:[1]

(i) At time-slot 0 the broadcaster disseminates[2] a signed version of their input.

(ii) At time-slot 1, the processors look to see whether they have received a signed value from the broadcaster, and if so then they 'recognise' that value. Now, though, rather than just passing on that signed value, they attach their own signature to the message, so that now it has been signed twice – first by the broadcaster and then secondly by them. Then they disseminate this new version of the message.

(iii) Then we stipulate that, if a processor is to 'recognise' a new value at any time-slot t, the message must have been signed by t distinct processors (with the broadcaster signing first). If they recognise a new value at time-slot t, they add their signature to the list, and disseminate that message (now with $t+1$ distinct signatures).

(iv) At time-slot $f+1$ we give the processors a last chance to recognise new values (but not to share again), before either outputting the single value they have recognised or else the default value.

Why does this approach work? We have to show that if any correct processor recognises a certain value $v \in V$, then all correct processors will also recognise that value. There are two cases to consider:

[1] The result was originally proved by Lamport, Shostak and Pease [1], but the proof described here was given a little later by Dolev and Strong [4].

[2] Recall from Section 3.2 that 'disseminate' means 'send to all processors'.

- **Case 1**. Suppose that some correct p_i first recognises v at a time-slot $t < f + 1$. In this case, p_i receives a message relaying the value v at time-slot t that has t distinct signatures attached. Processor p_i then adds their signature to form a message with $t + 1$ distinct signatures and sends this message to all processors. This means all correct processors will recognise v by time-slot $t + 1$ ($\leq f + 1$).
- **Case 2**. Suppose next that some correct p_i first recognises v at time-slot $f + 1$. In this case, p_i receives a message relaying the value v at time-slot $f + 1$ that has $f + 1$ distinct signatures attached. At least one of those signatures must be from a correct processor p_j (since there are at most f faulty processors), meaning that Case 1 applies w.r.t. p_j, i.e., p_j must have recognised and forwarded v to all processors at a previous time-slot.

In summary, the approach works because either a correct processor recognises v at some time-slot $< f + 1$, in which case they pass it on and all correct processors recognise v, or else some correct processor recognises v at time-slot $f + 1$. In the latter case, the fact that $f + 1$ processors have already signed the message means that some correct processor has already recognised (and passed on) v at a previous time-slot.

In the next subsection, we give a more formal version of the protocol.

5.1.2 A formal description of the protocol

The proof described above was quite simple, but was described in somewhat informal language. Here is a more formal version. Recall that $\langle m \rangle_i$ is the message m signed by p_i. For $v \in V$, and for distinct processors $p_{i_1}, \ldots, p_{i_t}$, we let $\langle v \rangle_{i_1,\ldots,i_t}$ be v signed by $p_{i_1}, \ldots, p_{i_t}$ in order, i.e., for each $k \in (1, t]$, $\langle v \rangle_{i_1,\ldots,i_k}$ is $\langle v \rangle_{i_1,\ldots,i_{k-1}}$ signed by p_{i_k}. For $t \in \mathbb{N}_{\geq 1}$, let M_t be the set of all messages of the form $\langle v \rangle_{i_1,\ldots,i_t}$ such that $v \in V$, $p_{i_1}, \ldots, p_{i_t}$ are distinct processors, and p_{i_1} is the broadcaster. Each processor p_i maintains a set O_i, which can be thought of as the set of values that p_i recognises, and which is initially empty. We let $\perp$ denote a 'default' element of V.

The pseudocode is shown in Figure 5.1

The instructions for processor p_i are as follows.

Time-slot 0. If p_i is the broadcaster and if p_i's input is v, then p_i disseminates $\langle v \rangle_i$ and adds v into O_i.

Time-slot t **with** $1 \leq t \leq f + 1$. Consider the set of messages $m \in M_t$ that p_i receives at time-slot t. For each such message $m = \langle v \rangle_{i_1,\ldots,i_t}$, if $v \notin O_i$, proceed as follows: add v into O_i and if $t < f + 1$ disseminate $\langle m \rangle_i$.

The output for processor p_i. After executing all other instructions at time-slot $f + 1$, p_i outputs v if O_i contains the single value v, and otherwise p_i outputs $\perp$.

FIGURE 5.1: Pseudocode for the Dolev-Strong protocol.

5.1.3 The verification

We have to verify that the protocol satisfies Termination, Agreement, and Validity:

- *Termination.* Satisfaction of this condition is obvious, because correct processors give an output at time-slot $f + 1$.
- *Agreement.* It suffices to show that if any correct processor p_i recognises $v \in V$ (adds v into O_i), then all correct processors do so. This follows from the proof already given above.
- *Validity.* If the broadcaster, p_i say, is correct and has input v, then all correct processors will recognise v by time-slot 1. Since p_i does not send any messages $\langle u \rangle_i$ for $u \neq v$, no correct processor can recognise any value other than v. All correct processors therefore output v.

5.1.4 Making the protocol more efficient

In the protocol as we have described it so far, processors can recognise (and pass on) any number of values in V. To make the protocol more efficient in terms of the number of messages sent by correct processors, note that it suffices for each processor to recognise and pass on at most *two* values. That way, either all correct processors recognise no values at all and give the default output, or they all recognise a single value and give that as their output, or they all recognise two values and give the default output. In any case, correct processors output correctly. Exercise 5.1 asks you to write down a formal description of the modified protocol and prove that it solves BB.

5.1.5 Why do we need other protocols for BB/BA?

Given that the Dolev-Strong protocol can handle any number of faults, do we need to consider any other protocols for BB? The main drawbacks of the protocol are:

- It requires either the lock-step model or (when suitably modified) the synchronous model, and;
- It is slow, requiring $f + 1$ rounds of communication.

In some real-world contexts, f could be on the order of one hundred or more, so the requirement for $f + 1$ rounds of communication is significant. Actually, we will show in the next section that any deterministic protocol requires $f+1$ rounds of communication, but we will also see later (in Chapters 9 and 12) that, in certain suitably formalised settings, some probabilistic protocols can terminate in an expected constant number of rounds. Even among deterministic protocols, Exercise 5.3 investigates an important sense in which the Dolev-Strong protocol is slow.

5.1.6 Exercises

Exercise 5.1. *Write down a formal description of the more efficient version of the Dolev-Strong protocol described in Section 5.1.4. Prove that it solves BB. Show that, for this modified protocol, the total number of messages sent by all correct processors combined is $O(n^2)$. Note that a single processor sending a single message to all others means sending $n - 1$ messages.*

Exercise 5.2. *In Chapter 2 we observed that, for the Byzantine fault-model, no protocol can solve BA when $f \geq n/2$. Use the Dolev-Strong protocol to show that this is not true for the crash-fault model if we only require the following version of Validity: if all processors receive the same input v, then all correct processors must output v. Hint: consider the approach we described in Chapter 4, in which one uses a protocol for BB to solve BA. Note that the Dolev-Strong protocol can be adapted to the crash-fault model so that, when the broadcaster is faulty, either all correct processors output the broadcaster's input value, or else they all output the special value 'faulty'. Then, apply a majority vote argument to the values output by each instance of BB.*

Exercise 5.3. *This question explores an important sense in which the Dolev-Strong protocol is slow when compared to some other deterministic protocols.*

Some protocols satisfy useful 'early stopping' properties, whereby processors can output in a smaller number of time-slots in many executions (e.g., if it happens to be the case that the actual number of faulty processors is less than the known bound f). Consider modifications of the Dolev-Strong protocol in which the messages sent by correct processors are unchanged, but where correct processors may output before time-slot $f+1$ under some conditions. Does there exist any such modification (still solving BB for any $f \leq n$) for which correct processors can sometimes/always output before time-slot $f+1$ if all processors act correctly?

Exercise 5.4. *This question explores a context that sometimes arises when building blockchains, in which some known subset of the processors are able to receive and pass on messages, but cannot create new messages themselves.*

Suppose $f < n$ and that some subset of $m < n - f$ of the correct processors are required to be 'mute'. Mute processors are not allowed to create new messages (including signed versions of messages they have received), but can send messages they have received to others. Specify a version of the protocol that works for this new setting, and prove that your protocol works as required. Hint: consider allowing mute processors to pass on messages at odd time-slots, while moving other instructions to even time-slots.

5.2 Proving $f+1$ rounds of communication are necessary

When considering the lock-step model, we'll say a protocol *terminates in x rounds* if it holds in every execution that all correct processors output by time-slot x, i.e., after x rounds of communication. Note that the Dolev-Strong protocol solves BB and terminates in $f+1$ rounds. In the same paper in which they introduced the protocol, Dolev and Strong showed that no deterministic protocol for solving BB terminates in f rounds. In fact, they showed that this is true even when we restrict to the crash-fault model.

Theorem 5.2. *Consider the lock-step model (with or without signatures) and suppose $f < n - 1$. No deterministic protocol for (binary) Crash-fault Agreement terminates in f rounds.*

In this section, we'll prove Theorem 5.2, giving our first (non-trivial) impossibility result. While the original proof was by Dolev and Strong, we'll give a simplified version of a proof by Aguilera and Toueg [5].

Corollaries of Theorem 5.2. Of course, the result for crash-faults immediately gives the corresponding result for BA. By the reduction of Chapter 4, the result for BA immediately suffices to give the result for BB when $f < n/2$. Exercise 5.8 asks you to modify the proof to give the result for BB and all $f < n - 1$.

Exercise 5.5. *Suppose $f = 0$ and $n \geq 2$. Show that no protocol for Crash-fault Agreement (or for Agreement under any other fault model) terminates in 0 rounds, so that it suffices to consider the case $f > 0$ in what follows.*

5.2.1 Two new techniques

Our proof of Theorem 5.2 combines two fundamental techniques that appear throughout distributed computing. The first technique is the use of 'indistinguishability arguments'. The core idea is that processors can only make decisions based on what they observe during an execution. If a processor p_i receives the same input and the same sequence of messages in two different executions, it must produce the same output in both. In this case, we say the two executions are *indistinguishable* for p_i. A typical indistinguishability argument derives a contradiction by describing one execution in which a processor p_i gives a particular output, and then describing another in which p_i should not have the same output, but must do because the two executions are indistinguishable for p_i.

Example: A simple indistinguishability argument. As a simple example, consider Crash-fault Agreement and $n = 2$ processors, where p_0 has input 0 and p_1 has input 1. Recall that executions begin at time-slot 0. If both processors output 0 at time-slot 1 in an execution E in which both processors are correct, then we can use an indistinguishability argument to derive a contradiction to Validity.
Consider a crash-fault execution E' which is identical to E, except that p_0 crashes at time-slot 0, but still sends the same message to p_1 at time-slot 0 as in E. Since E and E' are indistinguishable for p_1 up to the end of time-slot 1, and since p_1 outputs 0 at time-slot 1 in E, it must also output 0 at time-slot 1 in E'. This contradicts Validity, since all correct processors have input 1 in E'.

FIGURE 5.2: A simple indistinguishability argument.

The second new technique is the use of 'bivalency arguments'. Roughly, a partial execution (up to some time-slot t, say) is *bivalent* if it has not yet been decided how correct processors will output. Our proof of Theorem 5.2 will suppose that there exists a protocol that always terminates in f rounds, and then derive a contradiction by showing that there exist partial executions in which processors are supposed to have terminated, but which must also be bivalent.

Indistinguishability and bivalency will be introduced formally in Sections 5.2.3 and 5.2.4. First, we need to define *runs*, which are just partial executions.

5.2.2 Defining runs as partial executions

Fix n and f with $0 < f < n - 1$, and any protocol $\mathcal{P}$. Since we consider the crash-fault model, all processors have the state-transition-diagram specified by $\mathcal{P}$, and at most f processors may crash during any execution. In fact, it will be convenient to further restrict to executions in which *at most one processor crashes at each time-slot*: this gives us a tidy space to work in, and proving that the impossibility result holds even when we restrict to this specific class of executions clearly suffices to establish the claim of Theorem 5.2. So, let $\mathcal{E}$ be the set of executions of $\mathcal{P}$ of this form (and in the lock-step model). We make the following definitions:

- By a 0-run (think of a 'run of length 0'), we mean a specification of the input (0 or 1) to each processor.
- Recall that we start from time-slot 0. If $k \geq 1$, by a k-run we mean a specification of:
  - (i) The input to each processor;
  - (ii) The messages sent and received (on each channel) by each processor at each time-slot $< k$, and;
  - (iii) Which processors crashed at each time-slot $< k$.

By a *run*, we mean a k-run, for some k. We define whether one run *extends* another in the obvious way. If r is a k-run and r' is a k'-run with $k' \geq k$, we say r' extends r if: (i) in r', all processors receive the same inputs as in r; (ii) at each time-slot $< k$, the messages sent and received (on each channel) by each processor are the same in r' as in r, and; (iii) at each time-slot $< k$, the same processors crash in r' and r. We also extend this terminology in the obvious way to executions. So, if $E \in \mathcal{E}$, and r is a k-run, we say E extends (or is an *extension* of) r if the three conditions above hold when r' is replaced by E. Finally, we let $\mathcal{R}_k$ be the set of k-runs that have an extension in $\mathcal{E}$.

5.2.3 Indistinguishable runs and executions

We say two executions $E, E' \in \mathcal{E}$ are *indistinguishable* for p_i if:

- (i) p_i receives the same input in both executions;
- (ii) At each time-slot, the messages sent and received (on each channel) by p_i are the same in E as in E'.
- (iii) p_i crashes at the same time-slot (if any) in both executions.

Similarly, we say two k-runs are indistinguishable for some p_i if (i)-(iii) above hold with respect to those runs when we restrict to time-slots $< k$. The crucial point is this: *If two executions/runs are indistinguishable for correct p_i, and if p_i has some output in one, then it must have the same output in the other.* This basic fact will be a useful tool in our impossibility argument, and is also central to most impossibility arguments in distributed computing.

5.2.4 Univalent and bivalent runs

We make the following definitions:

- A run r is *0-valent* [*1-valent*] if all correct processors output 0 [1] in every execution in $\mathcal{E}$ extending r.
- A run is *univalent* if it is 0-valent or 1-valent, otherwise it is *bivalent*.

Intuitively, a bivalent run is one in which the protocol hasn't committed to either output yet.

Motivating bivalency. With these definitions in place, the basic form of our argument will be quite simple. First, we'll give a short proof showing that some 0-run is bivalent. This proof will be a simple indistinguishability argument. Then we'll use a proof by induction (which also includes multiple indistinguishability arguments) to show that, if $\mathcal{P}$ terminates in f rounds, some f-run is bivalent. This gives a contradiction, since, according to the framework as specified in Chapter 3, each f-run suffices to specify the state of each correct processor p_i at time-slot f, as well as the messages received by p_i at time-slot f. If $\mathcal{P}$ terminates in f rounds, any f-run therefore specifies the output of each correct processor.

The formal proof appears in Section 5.2.5.

5.2.5 The proof of Theorem 5.2

Fix n and f with $0 < f < n - 1$. Suppose $\mathcal{P}$ is a deterministic protocol that solves Crash-fault Agreement and terminates in f rounds. As noted already in Section 5.2.4, our basic aim is to derive a contradiction by showing that some f-run is bivalent. Since the proof is by induction, first we consider the base case.

Lemma 5.3. *There exists a bivalent* 0*-run.*

Proof. Towards a contradiction, suppose all 0-runs are univalent. Recall that the set of processors is $\Pi = \{p_0, \ldots, p_{n-1}\}$. For each $i \in [0, n]$, let r_i be the 0-run in which all processors p_j for $j < i$ receive input 1, while all other processors receive input 0. Since $\mathcal{P}$ satisfies Validity, it follows that r_0 is 0-valent, while r_n is 1-valent, and from this it follows that there exists $i \in [0, n)$ such that r_i is 0-valent, while r_{i+1} is 1-valent. This is illustrated in Figure 5.3.

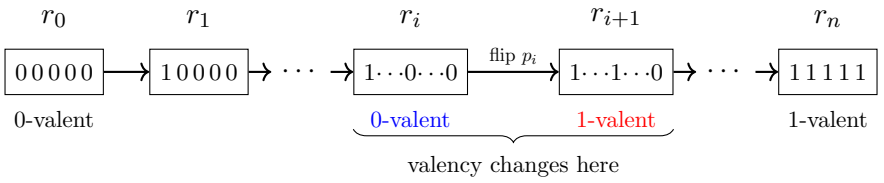


FIGURE 5.3: The sequence of 0-runs $r_0, r_1, \ldots, r_n$. Each box represents the inputs to the n processors. In r_0, all inputs are 0, so by Validity the run is 0-valent. In r_n, all inputs are 1, so by Validity the run is 1-valent. If all runs in the sequence are univalent, there must exist adjacent runs r_i and r_{i+1} of different valency.

Now consider the two executions in $\mathcal{E}$ extending r_i and r_{i+1}, in which all processors are correct, except that p_i crashes at time-slot 0, without sending any messages. These two executions are indistinguishable for all processors other than p_i, meaning that all other processors must give the same output in both executions. This gives the required contradiction. □

Analysis of Lemma 5.3. The argument proceeded by constructing a sequence of runs, where each differs from the next only in what is received by a *single* processor, and where the first and last runs are of differing valency. If all runs in the sequence are univalent, there must exist two *adjacent* runs of differing valency, and which only differ in what is received by a single processor p_i. Then we reach a contradiction by considering what happens when p_i crashes. We will use the same idea to prove the induction step.

Now we complete the argument. We prove by induction that, for all $k \leq f$, $\mathcal{R}_k$ contains a bivalent k-run.

Base case. This is Lemma 5.3.

Induction Step. For $k < f$, let r be a bivalent k-run in $\mathcal{R}_k$ and note that, by the definition of $\mathcal{R}_k$, at most k processors crash in r. Towards a contradiction, suppose that all extensions of r in $\mathcal{R}_{k+1}$ are univalent. Let $r^* \in \mathcal{R}_{k+1}$ be the extension of r in which no processors crash at time-slot k. By assumption, r^* is univalent. W.l.o.g. suppose it is 1-valent. Since r is bivalent, it has another extension in $\mathcal{R}_{k+1}$ that is 0-valent, r_0 say. A single processor p must crash at time-slot k in r_0. Suppose p fails to send messages to $p_{i_1}, \ldots, p_{i_d}$ at time-slot k. For each $j \in [1, d]$, let r_j be the run in $\mathcal{R}_{k+1}$ that is identical to r_0, except that p does send messages to $p_{i_1}, \ldots, p_{i_j}$. This is illustrated in Figure 5.4.

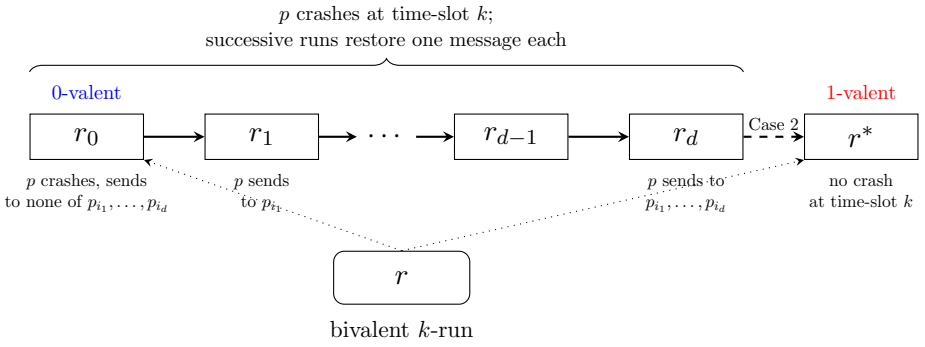


FIGURE 5.4: The induction step. The bivalent k-run r has a 0-valent extension r_0 (in which p crashes at time-slot k and fails to send messages to $p_{i_1}, \ldots, p_{i_d}$) and a 1-valent extension r^* (in which no new processor crashes). The intermediate runs $r_1, \ldots, r_d$ progressively restore the missing messages: in r_j, processor p crashes but does send messages to $p_{i_1}, \ldots, p_{i_j}$. In Case 1, the valency changes between consecutive runs r_{j-1} and r_j in this chain. In Case 2, r_d is still 0-valent, and the valency change occurs between r_d and r^*.

There are two cases to consider.

Case 1. For some $j \in [1, d]$, r_{j-1} is 0-valent, while r_j is 1-valent. The only difference between r_{j-1} and r_j is that p_{i_j} is sent a message from p at time-slot k in r_j. There exists at least one correct processor not equal to p_{i_j}, since $f < n - 1$. If $k + 1 = f$, we get an immediate contradiction to the claim that r_{j-1} is 0-valent while r_j is 1-valent, since all correct processors output at time-slot $k + 1$, and all correct processors other than p_{i_j} receive the same messages at time-slot $k + 1$ in all extensions of r_{j-1} or r_j. If $k < f - 1$, consider the two executions, extending r_{j-1} and r_j respectively, in which p_{i_j} crashes at time-slot $k + 1$ without sending any messages at that time-slot, and in which no processor crashes at any later time-slot. These two executions are indistinguishable for all correct processors, again contradicting the claim that r_{j-1} is 0-valent while r_j is 1-valent.

Analysis of Case 1. The argument is almost identical to that for Lemma 5.3. We consider a sequence of runs. If all members of the sequence are univalent, and if there exist two adjacent runs of differing valency, these two runs differ only in what is received by a single processor. Then we reach a contradiction by considering what happens when that processor crashes.

Case 2. Otherwise, r_d is 0-valent and r^* is 1-valent. The only difference between the two runs is that p crashes at time-slot k in r_d. If $k+1 = f$, we get an immediate contradiction to the claim that r_d is 0-valent while r^* is 1-valent, since all correct processors output at time-slot $k + 1$, and all processors other than p receive the same messages at time-slot $k + 1$ in all extensions of r_d or r^*: p can send messages at time-slot $k + 1 = f$ in extensions of r^*, but those messages will not be delivered until after all correct processors have outputted. If $k + 1 < f$, then consider executions:

- $E_1 \in \mathcal{E}$ extending r_d, in which no processor crashes at time-slots $> k$, and;
- $E_2 \in \mathcal{E}$ extending r^*, in which p crashes at time-slot $k + 1$ without sending any messages at that time-slot, and in which no processor crashes at any later time-slot.

These two executions are indistinguishable for all correct processors, contradicting the claim that r_d is 0-valent, while r^* is 1-valent. □

5.2.6 Exercises

Exercise 5.6. *This question investigates some differences between the proof given by Aguilera and Toueg [5] and the proof presented here. In the proof above, we argued that, if $\mathcal{P}$ terminates in f rounds, there exists an f-run that is bivalent.*

(i) Where is the fact that $\mathcal{P}$ terminates in f rounds used in the proof?

(ii) The proof of Aguilera and Toueg begins by showing that, if $\mathcal{P}$ terminates in f rounds, then every run in $\mathcal{R}_{f-1}$ must already be univalent. Prove their claim. Hint: towards a contradiction, suppose there is a bivalent run in $\mathcal{R}_{f-1}$. Consider two extensions of different valency: one in which no processors crash at time-slot $f-1$ and another in which one more processor crashes. Use an indistinguishability argument similar to those we described above to contradict the idea that these runs have different valencies.

Exercise 5.7. *The statement of Theorem 5.2 assumes $f < n - 1$. Show that, for $f \geq n - 1$, there exists a protocol that solves BB and terminates in f rounds.*

Exercise 5.8. *Show how to modify the proof of Theorem 5.2 to give the result for BB (for all $f < n - 1$). Hint: one only has to modify the part of the proof establishing that there exists a bivalent 0-run. For BB, define a 0-run to be a specification of which processor is the broadcaster and their input.*

5.3 Quadratic communication is necessary

We are generally interested in protocols for BB and BA when f is at least a constant fraction of n. In this section, we show that deterministic protocols solving BB for such adversaries require correct processors to send a number of messages that is at least quadratic in n. By the reduction of Chapter 4, the result also holds for BA. This lower bound has often been seen as a fundamental barrier to describing consensus protocols that *scale*, i.e., remain practical when the number of processors is large.

The result was originally proved by Dolev and Reischuk [6]. We present a proof that uses an indistinguishability argument very similar to theirs, but with some minor simplifications. While they stated the result for Byzantine faults, it is easily observed that it suffices to consider omission-faults.

Theorem 5.4. *Consider the lock-step model (with or without signatures). Suppose $f < n - 1$. Any deterministic protocol for Omission-fault Broadcast has executions in which correct processors send at least max $\{(n-1)/2, (f/2)^2\}$ messages.*

Proof. The key idea is to show that if processors send too few messages, we can construct an execution in which some processor doesn't receive any messages, but needs to receive messages to output correctly.

Fix a broadcaster. As noted above, the basic idea is that we'll consider executions in which processors receive few messages and derive a contradiction. So, let us start by considering how processors output when they receive no messages at all. Note that some $d \in \{0, 1\}$ must satisfy the condition that there exists a set of at least $(n-1)/2$ processors Q other than the broadcaster that *do not output* d in an execution in which they are correct and do not receive any messages.[3] W.l.o.g., suppose this is true for $d = 0$. Then, in the execution in which all processors are correct and the broadcaster has input 0, all processors in Q must receive a message. So, this is an execution in which correct processors send at least $(n-1)/2$ messages.

Now suppose that the maximum is not achieved by the first term in the statement of the lemma, so that $f > 0$. Next, we take an arbitrary subset $P_1 \subseteq Q$ of size $\lceil f/2 \rceil$, setting P_2 to be all processors outside P_1.[4] Why do we take P_1 to be of size $\lceil f/2 \rceil$? The plan is as follows:

- We consider the execution E_1, in which the broadcaster has input 0 and only the processors in P_1 are faulty. Processors in P_1 act correctly except that they do

[3] We phrase this in the negative to incorporate the possibility that they do not output at all.

[4] Recall that $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x, while $\lceil x \rceil$ denotes the least integer greater than or equal to x.

not send or receive messages to or from each other, and ignore messages from processors in P_2.

- Correct processors must output 0 in E_1, since the broadcaster is correct and has input 0.
- If each processor in P_1 receives at least $\lceil f/2 \rceil$ messages from processors in P_2 in E_1, then we are done, since the correct processors in P_2 must then send at least $|P_1| \cdot f/2 \geq (f/2)^2$ messages (combined). Otherwise, there exists $p \in P_1$ that receives at most $\lfloor f/2 \rfloor$ messages from processors in P_2.
- Now we use the fact that, because P_1 is of size $\lceil f/2 \rceil$, we have $\lfloor f/2 \rfloor$ processors left to be faulty. We consider an execution E_2 in which p is correct, the remaining processors in P_1 act as in E_1, and all processors in P_2 act correctly except that they do not send messages to p (this requires at most $\lfloor f/2 \rfloor$ of them to be faulty).
- Since p ignored messages from P_2 in E_1, does not receive messages from P_2 in E_2, and does not receive messages from P_1 in either execution, it sends the same messages to processors in P_2 in both executions. In fact, E_1 and E_2 are indistinguishable for all processors in P_2 that are correct in E_2. All such processors must therefore output 0 in E_2, while p does not output 0, since it belongs to Q and receives no messages. This contradicts Agreement (or Termination).

The above describes the whole argument, but now let's flesh out the details. As noted above, correct processors must output 0 in E_1, and we are done if each processor in P_1 receives at least $\lceil f/2 \rceil$ messages from processors in P_2 in E_1. So, suppose that, in E_1, $p \in P_1$ receives at most $\lfloor f/2 \rfloor$ messages from processors in P_2. Let A be the set of all processors in P_2 that send messages to p in E_1, noting that $|A| \leq \lfloor f/2 \rfloor$. We specify E_2 as follows:

- Processors in $\{p\} \cup P_2 \setminus A$ are correct.
- Processors in $P_1 \setminus \{p\}$ act correctly, except that they do not send or receive messages to or from processors in P_1, and ignore messages from processors in P_2.
- Processors in A act correctly, except that they do not send messages to p.

Then it follows directly by induction on time-slots that the only possible differences between the messages sent at each time-slot t in E_1 and E_2 are:

1. In E_1, a processor in A may send a message to p that is not sent in E_2, but is anyway ignored by p in E_1.
2. In E_2, p may send a message to $p' \in P_1$ that is not sent in E_1, but is anyway ignored by p' in E_2.

In particular, E_1 and E_2 are indistinguishable for processors in $P_2 \setminus A$. Since $f < n - 1$, the set $P_2 \setminus A$ is non-empty. Since E_1 and E_2 are indistinguishable for processors in this set, they must output 0 in E_2 (as in E_1). However, p receives no messages in E_2 and therefore does not output 0, since it belongs to Q. This contradicts satisfaction of Agreement (or Termination). □

Further discussion. Theorem 5.4 lower bounds *message complexity*, i.e., the number of messages sent by correct processors. To give a more fine-grained analysis, it is also interesting to consider *communication complexity*, i.e., the number of *bits* sent by correct processors. To what extent is the theorem tight with respect to these two measures?

Further results of Dolev and Reischuk. As well as Theorem 5.4, Dolev and Reischuk showed how to transform any protocol solving BB or BA with message complexity $O(n^2)$ into a protocol using signatures with message complexity $O(f^2 + n)$. Additionally, they established that any protocol solving BB requires communication complexity $\Omega(nf)$ if signatures are available, and message complexity $\Omega(nf)$ if signatures are not available. To our knowledge, it remains open whether Omission-fault Broadcast or Omission-fault Agreement can be solved with communication complexity $O(n + f^2)$.

Achieving the quadratic bound. The positive result of Dolev and Reischuk, achieving $O(f^2+n)$ message complexity, assumes a protocol for BB or BA with message complexity $O(n^2)$. For BB with signatures, we have already seen that there is a version of the Dolev-Strong protocol in which each correct processor except the broadcaster sends at most two messages to all other processors, and with message complexity $O(n^2)$. When $f < n/3$, Berman, Garay and Perry [7] showed that BA (and so BB) can be solved with communication complexity $O(n^2)$ without signatures. The proof uses a *recursive* form of the Phase King protocol: the Phase King protocol will be explained in Chapter 6, and the recursive form is discussed in Appendix C. In Chapter 6, we also show that BA and BB are not solvable without signatures when $f \geq n/3$. When $f < n(1/2 - \epsilon)$ for some $\epsilon > 0$, and when signatures are available, Momose and Ren [8] show that BA can be solved with communication complexity $O(n^2)$. Exercise 5.10 walks you through a simple way to show that any protocol solving BA with communication complexity $O(n^2)$ can be converted into a protocol solving BA with communication complexity $O(nf)$.

Crash faults. Galil, Mayer and Yung [9] show that Crash-fault Broadcast can be solved with message complexity $O(n)$. In Appendix B, we describe a much simpler (but less time-efficient) protocol solving binary Crash-Fault Broadcast with communication complexity $n + f$.

5.3.1 Exercises

Exercise 5.9. *Dolev and Reischuk [6] also produce a lower bound on the number of signatures (signed messages) sent by correct processors in the case of Byzantine faults. Since some protocols solve BB without signatures (see Chapter 6), the lower bound must somehow count the messages that do not contain signatures. Dolev and Reischuk therefore make the technical assumption that every message carries at least the signature of its sender. Alternatively, the lower bound can count the number of signatures together with the number of messages without signatures. The claim is that, if $0 < f < n - 1$, then any protocol for BB in the lock-step model must have an execution in which correct processors send at least $n(f + 1)/4$ signatures. To establish the claim:*

(a) Fix a broadcaster. For $d \in \{0, 1\}$, let E_d be the execution in which all processors are correct and the broadcaster has input d. For each p, let $A(p)$ be all processors $p' \neq p$ such that either of the following holds in at least one of E_0 or E_1:

(i) p' *receives a message signed by* p *(perhaps as part of a larger message), or;*

(ii) p *receives a message signed by* p'*.*

Argue that if $A(p) \geq f + 1$ *for all* p*, then the claimed bound holds.*

(b) Suppose $A(p) \leq f$*. Argue that there exists an execution* E_2 *in which only the processors in* $A(p)$ *are faulty and in which these processors behave towards* p *as in* E_0 *(sending* p *the same messages at the same time-slots) and towards all other processors as in* E_1*.*

(c) Argue that p *must output 0 in* E_2*, while the other correct processors are a non-empty set and must all output 1.*

Exercise 5.10. *Show how to convert a protocol solving BA with communication complexity* $O(n^2)$ *into a protocol for BA with communication complexity* $O(fn)$*. Hint: Consider running the given protocol on a set of processors of size* $O(f)$*, and then having each processor in this set send its output to all other processors.*

Chapter 6

The lock-step model without signatures

In this chapter, we consider the lock-step model without signatures. Note that the impossibility results of the last chapter still hold for this setting. First, we show that solving BB or BA is not possible in this setting when $f \geq n/3$. Then we show that it is possible when $f < n/3$. The proof of the negative result is beautiful, while the proof of the positive result is instructive. On the other hand, it's okay just to remember the results of this chapter and then to skip to the next one if your aim is just to quickly get up to date with modern protocols, and if you don't intend to do research in the area.

6.1 The proof of Fischer, Lynch and Merritt

In this section, we prove Theorem 6.1 below. The result was originally proved by Lamport, Shostak and Pease [2], but the more elegant proof we present here is due to Fischer, Lynch and Merritt [10].

Theorem 6.1. *Consider the lock-step model without signatures. No protocol can solve BA or BB when $f \geq n/3$.*

By the reductions discussed in Chapter 4, it suffices to prove the theorem for BA. To present a proof for which it is entirely clear why all elements of the proof are necessary, it is convenient to momentarily consider (in Sections 6.1.1 and 6.1.2) an alternative to our standard setup regarding communication channels.

6.1.1 A revised communication model

According to the setup described in Chapter 3, each processor p_i knows which processor is at the other end of each communication channel $\{i, j\}$, i.e., if p_i receives a message on communication channel $\{i, j\}$ then p_i knows this message must be from p_j. However, to present a proof of Theorem 6.1 for which it is entirely clear why all elements of the proof are necessary, it is convenient to momentarily consider an alternative setup in which there is still a two-way channel corresponding to each pair of processors, and

where each processor is still aware of which channel each message arrives on, but where processors no longer begin with knowledge as to which processor is at the end of each of their channels. For example, in a given execution, p_i might have two channels to p_j and p_k. At each time-slot, it might receive messages from p_j on the first of these channels, and messages from p_k on the other. The instructions for p_i may also depend on which messages have arrived at each channel, but, in this revised model, p_i is not told which processor is at the other end of each channel.

Recall that the notion of 'indistinguishable' executions was introduced in Chapter 5 (there in the context of crash-faults): when we say that two executions are indistinguishable for a certain processor, we mean that the processor has the same inputs and sees precisely the same messages arriving on their communication channels at each time-slot in the two executions. To prove Theorem 6.1, we will ultimately describe an indistinguishability argument involving six processors. The reason we initially consider the revised communication model is so that we can first give a simpler proof involving four processors. Then we will come back to the standard setup, and explain why six processors are necessary for the full proof.

6.1.2 The case $n = 3$, $f = 1$ for the revised communication model

For the revised communication model, we can describe a simple proof by contradiction. The approach taken might initially seem a bit unusual: we suppose a working protocol exists and then we use indistinguishability arguments to deduce certain things about how the protocol must behave in scenarios which are different than the intended application. This might seem an odd thing to do, but the analysis allows us to deduce that there exist valid executions of the protocol in which it does not behave correctly.

We start by considering the case $n = 3$, $f = 1$. We suppose there exists a working protocol, and we consider what happens when *four* processors all execute the protocol for $n = 3$ and $f = 1$ correctly, when the communication channels are arranged as in Figure 6.1. In the figure, each node represents a correct processor. We refer to them by the names p_0, p_1, p'_0, and p'_1 and suppose that each processor p_i or p'_i for $i \in \{0, 1\}$ is told their name is p_i. Next to each processor is also indicated its input, either a or b. There are communication channels between processors p_0 and p_1, between processors p_0 and p'_1, between processors p'_1 and p'_0 and between processors p'_0 and p_1. Each processor behaves precisely as dictated by the protocol, according to its input and the messages that it receives on each communication channel at each time-slot. Of course, this is not a configuration in which the protocol was intended to operate, but the way in which it behaves here will allow us to deduce things about how the protocol behaves in valid executions (with the right number of nodes and with communication channels arranged in the standard fashion).

Deriving the contradiction. What can we deduce about the outputs of the processors in this setup?

1. Processors p_0 and p_1 must both output a, because this four processor execution is indistinguishable as far as they are concerned from a valid execution with three processors, in which p_0 and p_1 both have input a, and in which the third processor is

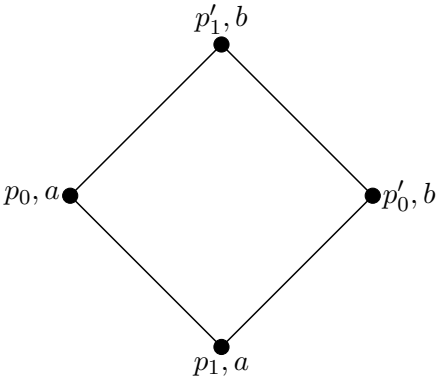


FIGURE 6.1

faulty and simulates p_0' and p_1' as arranged as in the figure, i.e, the faulty processor sends to processor p_1 at each time-slot what p_0' sends to p_1 in the four processor execution, and sends to p_0 what p_1' sends to p_0 in the four processor execution. This is illustrated in Figure 6.2.

2. Similarly (and symmetrically), processors p_0' and p_1' must output b.

3. Now we get to our contradiction. We have already deduced that p_0 must output a and that p_1' must output b. The problem is that this execution is indistinguishable as far as processors p_0 and p_1' are concerned from a valid execution in which the third processor is faulty and simulates p_0' and p_1 as arranged in the figure. The fact that p_0 and p_1' give different outputs therefore violates Agreement. This is illustrated in Figure 6.3.

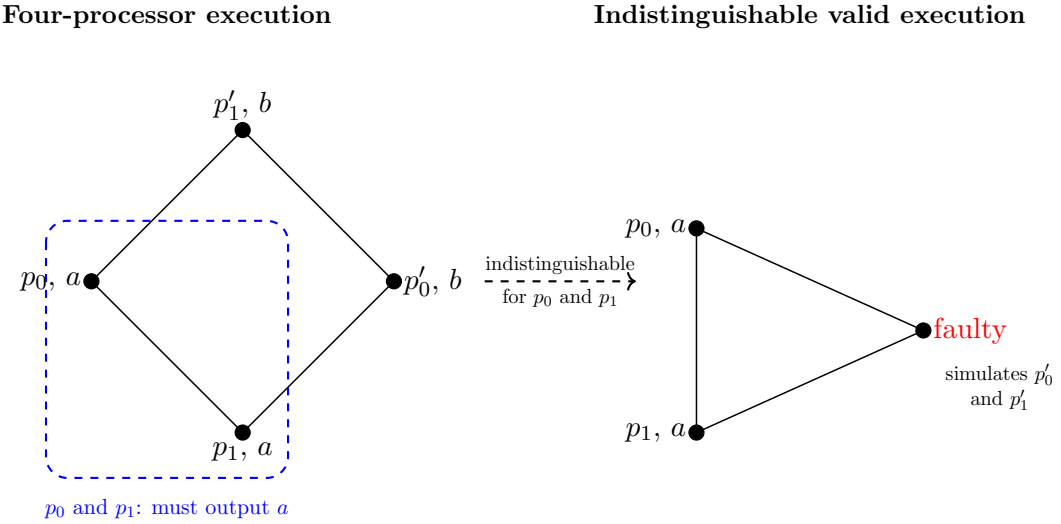


FIGURE 6.2: Step 1 of the argument. On the left is the four-processor execution from Figure 6.1. On the right is a valid three-processor execution in which p_0 and p_1 both have input a and the third processor is faulty, simulating the roles of p_0' and p_1'. Since these two executions are indistinguishable for p_0 and p_1, and since Validity requires output a in the valid execution, processors p_0 and p_1 must output a in the four-processor execution.

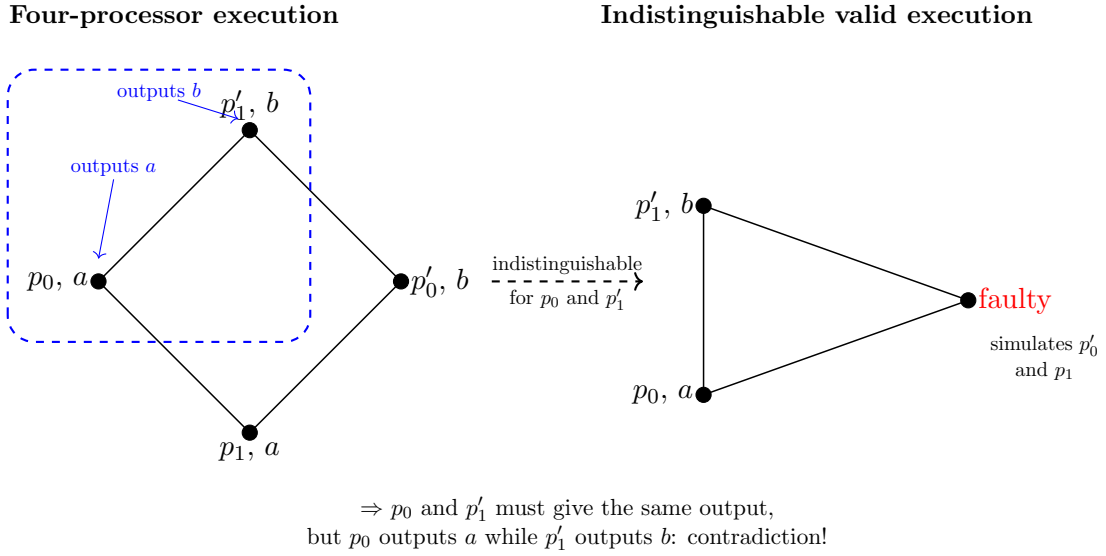


FIGURE 6.3: Step 3: the contradiction. From steps 1 and 2, we know p_0 outputs a and p'_1 outputs b in the four-processor execution. But this execution is indistinguishable for p_0 and p'_1 from a valid three-processor execution (shown on the right) in which the third processor is faulty and simulates p'_0 and p_1. Since p_0 and p'_1 are both correct in this valid execution, Agreement requires them to give the same output. This is a contradiction.

6.1.3 Back to the standard setup

What happens when we move back to the standard setup in which processors know who is at the end of each channel? The problem now is that we have to be more careful when deciding how to treat each channel during indistinguishability arguments. To see this, consider first processors p_0 and p_1 in Figure 6.1. If these processors both treat the channel between them as the channel $\{0, 1\}$, and if each processor $p_i \in \{0, 1\}$ treats the channel between p_i and p'_{1-i} as the channel $\{i, 2\}$, then we can again deduce that processors p_0 and p_1 must output a. If the same conditions hold when we reverse the role of i and i' for $i \in \{0, 1\}$, then it must also be the case that processors p'_0 and p'_1 output b. The problem now is that this execution does not look indistinguishable from any valid execution as far as processors p_0 and p'_1 are concerned, because in any valid execution they would both treat the communication channel between them as the channel $\{0, 1\}$ rather than $\{0, 2\}$.

Adding two more processors. There is a simple way to remedy this problem. We just introduce two further processors, as in Figure 6.4. We suppose that each processor with the name p_i or p'_i in the figure is told that its name is p_i, and that it treats the channel in the figure to any processor p_j or p'_j as the channel $\{i, j\}$. The argument now works much as before. Processors p_0 and p_1 must output a because for them the execution is indistinguishable from a three processor execution in which the processor p_2 is faulty and simulates processors p'_2, p'_0, p'_1 and p_2 in the figure. Processors p'_2 and p'_0 must output b because for them the execution is indistinguishable from a three processor execution in which the processor p_1 is faulty and simulates processors p_1, p_0, p_2 and p'_1 in

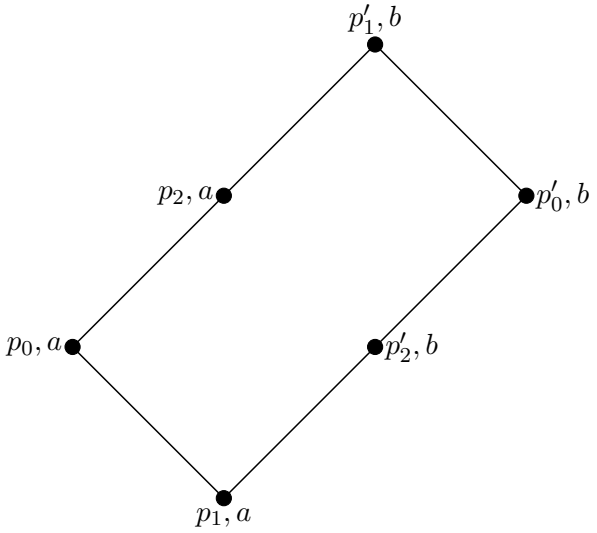


FIGURE 6.4

the figure. Now we get to the contradiction because processors p_1 and p_2' give different outputs, but the execution is indistinguishable for them from a three node execution in which processor p_0 is faulty and simulates processors p_0, p_2, p_1' and p_0' as arranged in the figure.

6.1.4 The proof for general n

So far, we have only dealt with the case $n = 3$ and $f = 1$. To deal with the general case, suppose there exists some $n > 3$ and $f \geq n/3$ for which a working protocol exists. This protocol can then be used to give a protocol that works for three processors when at most one is faulty. We divide the n processors in the n processor execution into three disjoint subsets P_0, P_1, P_2, so that each subset has at most one more element than the others. Then we have each of the three processors $p_i \in \{p_0, p_1, p_2\}$ in the three processor execution simulate the set of processors P_i in the n processor execution, giving as output the common output of all processors in P_i (which must exist, since the n processor protocol satisfies Termination and Agreement). It is straightforward to verify that, if the n processor protocol satisfies Termination, Agreement, and Validity, then the protocol for three processors also satisfies these properties, contradicting the result of Section 6.1.3.

6.2 The Phase-King protocol

In this section, we prove Theorem 6.2 below. The result was originally proved by Lamport, Shostak and Pease [2]. Here we describe (a slight modification) of a simple and more instructive proof by Berman, Garay and Perry [11], which is known as the *Phase-King protocol*.

Theorem 6.2. *Consider the lock-step model without signatures. There exist protocols solving BA and BB when $f < n/3$.*

By the reductions of Chapter 4, it suffices to consider BA. To explain the main protocol, which must satisfy Termination, Validity and Agreement, in Section 6.2.1 we first consider a simple two-step sub-protocol called *Gradecast*, which satisfies Termination and Validity, but which does not yet satisfy Agreement.

6.2.1 The Gradecast protocol

Just as for BA, each processor receives an input $v \in V$. Now, though, they output a value in V and a *grade* $\in \{0, 1, 2\}$, which indicates something about the knowledge the processor has regarding other processors' outputs. The protocol is shown in Figure 6.5 (and assumes the conventions described in Section 3.2 concerning the meaning of the instruction to 'disseminate').

Gradecast: The instructions for p_i.

Time-slot 0. Disseminate v.

Time-slot 1. If there exists b such that (at least) $n - f$ processors sent b to p_i at time-slot 0, disseminate b.

Time-slot 2. Output as follows:

(i) If there exists b such that $n - f$ processors sent b to p_i at time-slot 1, then output b with grade 2.

(ii) Otherwise, if there exists b such that $f+1$ processors sent b to p_i at time-slot 1, then output b with grade 1.

(iii) If neither of the first two cases apply, output v with grade 0.

FIGURE 6.5: The pseudocode for Gradecast.

The Gradecast protocol is certainly simple. Let us examine its properties and verify that it is well-defined.

First, it cannot be the case that one processor p_i is sent b by $n-f$ processors at time-slot 0, while another processor p_j (possibly $p_i = p_j$) is sent $b' \neq b$ by $n - f$ processors. To see this, suppose otherwise:

- Let P be the set of processors that sent b to p_i, so $|P| \geq n - f$;
- Let P' be the set of processors that sent b' to p_j, so $|P'| \geq n - f$;
- The intersection $|P \cap P'| \geq |P| + |P'| - n \geq (n - f) + (n - f) - n = n - 2f$;
- Since $f < n/3$, we have $n - 2f > f$;
- Therefore $|P \cap P'| > f$, meaning $P \cap P'$ contains at least $f + 1$ processors;
- Since at most f processors are faulty, $P \cap P'$ contains at least one correct processor;
- This correct processor sent both b and b', contradicting the fact that correct processors send only one value at time-slot 0.

To phrase this another way, the protocol satisfies '*Time-slot 1 Agreement*':

> **Time-slot 1 Agreement**. If p_i is correct and disseminates b at time-slot 1, then no correct processor disseminates $b' \neq b$ at time-slot 1.

By the same argument, when p_i goes to output at time-slot 2, it cannot be the case that (i) holds for two distinct values of b. It follows directly from Time-slot 1 Agreement that (ii) cannot hold for two distinct values of b, since if $f + 1$ processors send b to p_i at time-slot 1 then at least one is correct. This suffices to show that the protocol is well-defined. Processors are instructed to send at most one value at each time-slot, and the instructions suffice to specify a single output for each processor.

It is also easy to see that the protocol satisfies a form of Validity:

> **Validity**$^+$. If all correct processors have the same input v, then they all output v with grade 2.

Outputting v with grade 2 also implies knowledge about the outputs of other processors:

> **Knowledge of Agreement**. If any correct processor outputs a value b with grade 2, then all correct processors output b.

So, if a correct processor outputs b with grade 2, then it *knows* all correct processors have output b. To prove Knowledge of Agreement, note that, if a correct processor outputs b with grade 2, then it is sent b by at least $n - f$ processors at time-slot 1. This means all correct processors are sent b by at least $n - 2f \geq f + 1$ processors at that time-slot.

6.2.2 Introducing tie-breaking

Where has this got us? So far, we have a simple protocol which, if all correct processors start with the same input, causes all correct processors to give that same input as their output with 'grade 2'. What we haven't achieved yet is any method for tie-breaking: if the processors start with a mix of inputs, how to we get them to agree on some value? The rough idea (made precise in Section 6.2.3) is as follows:

- We run $f + 1$ *views* (or 'rounds'), each of which has a different 'leader'.
- Processors start each view with a certain value. At the start of the first view this is just their input.
- Each view starts with an instance of Gradecast, after which processors update their value to be that specified by the their output in the Gradecast instance.
- After Gradecast, the leader of the view shares their value v. Each correct processor changes their value to v *unless* they outputted the Gradecast instance for this view with grade 2, before progressing to the next view (if this is not the last).
- At the end of the last view, processors output their current value.

If we proceed in this way, then Termination will be satisfied. It is also straightforward to see Validity will be satisfied: if all correct processors start with the same value v, then it follows inductively (from Validity$^+$) that they will output each Gradecast instance with $(v, 2)$, and will not change their value upon hearing from the leader.

It is also straightforward to see that, after any view with a correct leader (and then at all subsequent views), all correct processors will have the same value. To see this, note that a correct leader will share the same value with all processors. If a correct processor does not change their value to that of the correct leader because they outputted with grade 2 in the Gradecast instance, then Knowledge of Agreement means that the leader must anyway have the same value. It follows that Agreement will be satisfied.

6.2.3 The formal specification

The instructions are shown in Figure 6.6 (the variable r should be thought of as ranging over views):

The instructions for p_i.

Time-slot $4r$ ($r \in [0, f]$). Disseminate v.

Time-slot $4r + 1$ ($r \in [0, f]$). If there exists b such that (at least) $n - f$ processors sent b to p_i at time-slot $4r$, then disseminate b.

Time-slot $4r + 2$ ($r \in [0, f]$). Set v and `grade` as follows:

(i) If there exists b such that $n - f$ processors sent b to p_i at time-slot $4r + 1$, then set $v := b$, `grade` $:= 2$.

(ii) Otherwise, if there exists b such that $f+1$ processors sent b to p_i at time-slot $4r + 1$, then set $v := b$, `grade` $:= 1$.

(iii) If neither of the first two cases apply, set `grade` $:= 0$.

If $r = i$, disseminate v.

Time-slot $4r + 3$ ($r \in [0, f]$). If `grade` < 2 and p_i has received some value b from p_r at this time-slot, set $v := b$.

Time-slot $4(f + 1)$. Output v.

FIGURE 6.6: The pseudocode for Phase-King.

6.2.4 The verification

Proof of Termination: All correct processors output at time-slot $4(f + 1)$.

Proof of Validity: This follows from the Validity$^+$ property of Gradecast. If all correct processors have the same input v then, at the end of each view, the local value `grade` for a correct processor will be 2, meaning that the leader's value will be ignored. It follows inductively that correct processors will never change their local value v, and will output this value.

Proof of Agreement: Consider the first view with a correct leader. From the Knowledge of Agreement property, all correct processors will either switch to the leader's value v, or already have that value with grade 2. From Validity$^+$ applied to later views, it follows that correct processors will never subsequently change their value.

6.2.5 Message and communication complexity

The Phase-King protocol has each correct processor send $\Theta(n)$ messages and $\Theta(n)$ bits per view, meaning that all correct processors combined send $\Theta(n^2)$ messages/bits per view. Since there are $f + 1$ views, the overall message/communication complexity is therefore $\Theta(n^2 f)$. If $f = \Theta(n)$, then this means the message/communication complexity is $\Theta(n^3)$.

A variant of Phase-King, called Recursive Phase-King was used by Berman, Garay and Perry [11] to show that BA (and so BB) can be solved with communication complexity $O(n^2)$ in the lock-step model without signatures when $f < n/3$. The latter protocol is discussed in Appendix C.

6.2.6 Important takeaways

We presented the proof above, rather than the original proof of Lamport, Shostak and Pease, because it teaches some techniques that will be useful later. In particular, we will later be considering the *partially synchronous model*, where solving BA is impossible when $f \geq n/3$, even with signatures. In this context, the counting argument that we used to prove Time-slot 1 Agreement becomes one of our central tools:

> If $f < n/3$ and processors exchange values (correct processors only sending a single value), then there cannot exist $v \neq v'$ and correct processors p_i and p_j such that p_i receives v from at least $n - f$ processors, while p_j receives v' from $n - f$ processors also.

In the coming chapters, will also see that is a very common approach to design protocols that have instructions that are divided into *views*, each with a designated leader.

Chapter 7

State Machine Replication

In this chapter, we define the problem that 'blockchain' protocols have to solve, which is called *State Machine Replication* (SMR). Roughly, special messages called *transactions* are given to the processors over time, but some processors may receive these transactions at different times and in a different order than others. The processors have to reach consensus on a total ordering for the transactions, i.e., all correct processors should agree on the same sequence of transactions. All transactions received by correct processors should be included in this agreed sequence, called the *log*. For our purposes now, it does not matter what 'transactions' really are: they could be instructions to transfer funds, or, more generally, to update some database.

An important distinction between SMR and BA (or BB) is that the latter is a 'one-shot' problem, requiring processors to give a single output. On the other hand, SMR is a 'multi-shot' problem, requiring consensus on the first transaction, then the second, and so on. The reader may note that each of these subproblems is like the Byzantine Agreement problem, without exactly the same requirement for Validity. We'll discuss the exact relationship between SMR and BA/BB in detail later in this chapter.

Unfortunately, the literature often conflates SMR with a related problem called Total Order Broadcast (also known as Atomic Broadcast), creating confusion about what's actually solvable when $f \geq n/2$. Here, we'll carefully distinguish these problems to clarify this issue. The crucial difference involves external verifiability. In SMR, the requirements are actually slightly stronger than those described above. Processors must not only agree on a transaction sequence but also be able to demonstrate to outside observers that this sequence represents genuine consensus. Total Order Broadcast does not have this additional requirement, which we'll show makes SMR strictly harder than Total Order Broadcast when $f \geq n/2$. These problems are often conflated because, in ways we'll make precise, the two problems are equivalent when $f < n/2$.

In the next section, we'll formally define Total Order Broadcast (TOB). Then, in Section 7.3, we'll define SMR. We begin with Total Order Broadcast because understanding this simpler problem will help clarify what makes SMR more challenging.

7.1 Total Order Broadcast

The setup. Since we are principally interested in the 'blockchain' context, where transactions are signed messages, we'll define TOB and SMR in a model with signatures. A *transaction* is just a signed message, belonging to a set of signed messages $\mathcal{T}$ that is known to the protocol (given as input to all processors). We allow that messages in $\mathcal{T}$ may be signed by processors outside Π. Recall, from Chapter 3, that $p_i \in \Pi$ cannot send any message signed by a processor other than themselves until any time-slot at which they have received a message containing this sequence.

For TOB (as for SMR), we suppose that, in addition to messages sent by processors in Π, each processor in Π receives an arbitrary finite set of transactions at each time-slot:[1] we say such transactions are received *from the environment* (and when we say 'p_i receives the transaction tr' this means p_i either receives tr from another processor or from the environment). Since the transactions received from the environment are arbitrary in any given execution, we can think of them as being selected by the adversary.

Sequence notation. If σ and τ are sequences, we write $\sigma \preceq \tau$ to denote that σ is a prefix of τ. We say σ and τ are *compatible* if $\sigma \preceq \tau$ or $\tau \preceq \sigma$. If two sequences are not compatible, they are *incompatible*. If σ is a sequence of transactions, we write $\mathrm{tr} \in \sigma$ to denote that the transaction tr belongs to the sequence σ.

The requirements. Each processor p_i is required to maintain an append-only log, denoted $\log_i$, which at any time-slot is a sequence of distinct transactions, each of which has been received by p_i. We also write $\log_i(t)$ to denote the value $\log_i$ at the end of time-slot t. The log being append-only means that for $t' > t$, $\log_i(t) \preceq \log_i(t')$. We require the following conditions to hold in every execution:

Consistency. If p_i and p_j are correct, then for any time-slots t and t', $\log_i(t)$ and $\log_j(t')$ are compatible.

Liveness. If p_i and p_j are correct and if p_i receives the transaction tr then, for some t, $\mathrm{tr} \in \log_j(t)$.

When a processor appends a transaction to its log, it is common to say it has *finalised* the transaction. Roughly, Consistency requires that all correct processors finalise the same sequence of transactions. Liveness requires that any transaction received by a correct processor is eventually finalised by all correct processors.

Parameterising liveness. We note that our definition of Liveness stipulates that, if p_i and p_j are correct and p_i receives the transaction tr, then for *some* t, $\mathrm{tr} \in \log_j(t)$. So, tr must *eventually* be appended to the logs of correct processors, but the requirement does not specify a time-bound. An alternative is to require the existence of some known bound ℓ: if p_j is correct and correct p_i receives the transaction tr at t, then we now require $\mathrm{tr} \in \log_j(t)$ by $t + \ell$. In this case, we call ℓ the *liveness parameter*. Exercise 7.1 explores the extent to which insisting on the existence of some known liveness parameter (no matter how large) strengthens the requirement for Liveness.

[1] If a transaction is signed by a processor outside Π, we have a small technical issue: when first received by any processor, which channel should the transaction arrive on? The details are not important, but, for technical completeness, we can suppose that each processor p_i also has an extra channel $\{i, *\}$ on which such transactions arrive.

Now that we have formally defined TOB, we can explore when it is solvable. We do this in the next section.

7.2 When is TOB solvable?

In later chapters, we will consider a number of efficient protocols for solving TOB and SMR. In this section, we do not concern ourselves with issues of efficiency. Rather, we make the simple observation that the Dolev-Strong protocol (or any protocol for BB that terminates in a known finite number of rounds) can be used to solve TOB for any $f \leq n$. To do so, we proceed as follows:

- As for the Phase-King protocol (Chapter 6), we divide the execution into *views*. In each view, we now carry out a single instance of Dolev-Strong (consisting of $f + 1$ rounds).
- We let the *leader* for view v be p_i, where $i = v \bmod n$. So, leaders are 'rotating'.
- If p_i is the leader for view v, then they act as the broadcaster for this instance of the Dolev-Strong protocol. Upon starting the view, they collect all transactions received by the start of the view and not yet appended to their log. The 'value' they broadcast is a signed list of these transactions, indexed by the view.
- During view v, all correct processors then carry out an instance of the Dolev-Strong protocol, with 'default' value $\perp$. Processors ignore any 'value' b unless:
  - b specifies a sequence of transactions in $\mathcal{T}$;
  - No transaction in b already belongs to their finalised log;
  - b is indexed by v.

  These rules prevent Byzantine leaders from proposing invalid or duplicate transactions, while view indexing prevents interference between messages from different views.
- If a correct processor p_i outputs $\perp$ in the instance of Dolev-Strong for view v, then p_i does not append any transactions to their log at this stage. Otherwise, if p_i outputs b, then p_i appends the sequence of transactions specified by b to their log.

This construction works because Dolev-Strong guarantees all correct processors agree on the same value for each view, ensuring Consistency. Liveness follows because rotating leadership ensures every correct processor is the leader of infinitely many views. Exercise 7.2 asks you to formally describe the protocol, and to give a proof that it solves TOB.

As noted earlier, the difference between TOB and SMR is that SMR also requires a form of external verifiability. We make this precise in the next section.

7.3 Defining State Machine Replication

The setup. The setup is the same as for TOB, i.e., we suppose that, in addition to messages sent by processors in Π, each processor in Π receives an arbitrary finite set of transactions at each time-slot.

To specify the extra conditions required for SMR, there are a number of possible approaches. One very interesting approach, described in a recent paper by Sridhar, Tas, Neu, Zindros, and Tse [12], has in common with many previous definitions that one introduces a new class of participant called *clients*, who are observers of the protocol execution. In the treatment of Sridhar et al., however, clients may be of different kinds. For example, they may be always *active* and observing the protocol (receiving messages sent by processors in Π), or they may only be active at certain time-slots. Clients may have the ability to pass on messages they receive or not, and so on. In this way, one can define a number of different versions of SMR, which they show are often not equivalent to each other.

Here, we take a simple approach which avoids the need to explicitly introduce clients, and which succinctly encapsulates the distinction between TOB and SMR, and the reason SMR is not solvable when $f \geq n/2$. Our approach also works tidily in more complex scenarios, such as when we come to consider *player reconfiguration* (considered in Chapter 22).

The basic idea behind the approach is as follows. Suppose an interested party arrives part way through an execution, and asks a processor p_i (that may not be correct) for the current value of the log. How can p_i prove to this party that the sequence of transactions σ has been finalised? The only information that p_i has to build such a proof is the signed messages it has received. Moreover, p_i has no way of proving the *order* in which it has received those messages. So, the *set* of messages it has received is the only ingredient from which to construct the required proof. Next, we present the formal details.

The requirements. In this section, we use the following notation when discussing any execution of a protocol:

- $M_{p_i}(t)$ denotes the set of messages received by processor p_i by time-slot t;
- M^* denotes the set of all messages received by any processor during the execution.

If $\mathcal{P}$ is a protocol for SMR, then it must specify a function $\mathcal{F}$, which may depend on Π and $\mathcal{T}$, that maps any set of messages M to a sequence of transactions, each in $\mathcal{T} \cap M$. We require the following conditions to hold in every execution (for any M_1, M_2, p_i, p_j and any transaction tr):

Consistency. If $M_1 \subseteq M_2 \subseteq M^*$, then $\mathcal{F}(M_1) \preceq \mathcal{F}(M_2)$.

Liveness. If p_i and p_j are correct and if p_i receives the transaction tr then, for some t, $\text{tr} \in \mathcal{F}(M_{p_j}(t))$.

At any time-slot t, p_i sets $\log_i(t) = \mathcal{F}(M_{p_i}(t))$.

This definition of consistency ensures that correct processors never finalise incompatible sequences: for any sets of messages $M_1, M_2 \subseteq M^*$ that two such processors might have

received, $\mathcal{F}(M_1) \preceq \mathcal{F}(M_1 \cup M_2)$ and $\mathcal{F}(M_2) \preceq \mathcal{F}(M_1 \cup M_2)$. We say a set of messages M is a *certificate* for a sequence of transactions σ if $\mathcal{F}(M) \succeq \sigma$: intuitively, any processor can present the set of messages M to an interested party that has not been observing the execution as proof that it has finalised a sequence of transactions extending σ.

It is clear that any protocol solving SMR also solves TOB. In the next section, we show that, unlike TOB, SMR is not solvable when $f \geq n/2$. We also demonstrate a precise equivalence between the two problems when $f < n/2$ (in the context of a fixed processor set Π that we focus on here).

7.4 When is SMR solvable?

First, we'll show that SMR is not solvable when $f \geq n/2$. The proof is a simple indistinguishability argument.

Theorem 7.1. *Consider the lock-step model with signatures. No protocol solves SMR when $f \geq n/2$.*

Proof. Suppose $f \geq n/2$. Partition Π into two sets of processors P_0 and P_1, so that $|P_0| = f$ and $|P_1| = n - f$. Let tr_0 and tr_1 be distinct transactions, each signed by a processor outside Π, and set $\mathcal{T} = \{\mathrm{tr}_0, \mathrm{tr}_1\}$. We consider three executions of the protocol. In all three executions of the protocol, processors in P_i ($i \in \{0, 1\}$) receive tr_i from the environment at time-slot 0, and receive no further transactions from the environment. Let $\mathcal{F}$ be the function specified by the protocol, with respect to which it satisfies Consistency and Liveness.

Execution 0. Processors in P_0 are correct, while processors in P_1 are faulty and perform no action at any time-slot. Since $|P_1| = n - f \leq f$, at most f processors are faulty. By Liveness, all processors in P_0 must finalise tr_0 as the first transaction in their log by some time-slot, t_0 say. Let M_0 be the set of all messages received by correct processors by t_0.

Execution 1. Processors in P_1 are correct, while processors in P_0 are faulty and perform no action at any time-slot. Since $|P_0| = f$, at most f processors are faulty. By Liveness, all processors in P_1 must finalise tr_1 as the first transaction in their log by some time-slot, t_1 say. Let M_1 be the set of all messages received by correct processors by t_1.

Execution 2. Processors in P_1 are correct. Processors in P_0 are faulty and 'simulate' Execution 0. This means they act correctly, except that they ignore messages from processors in P_1, and do not send messages to processors in P_1, i.e., at each time-slot they follow the instructions given by the state-transition-diagram, except that:

- Messages from processors in P_1 are dropped when determining which messages to send and which state x to transition to, and;
- They then transition to state x and send messages to processors in P_0 as specified by the state-transition-diagram, but do not send messages to processors in P_1.

Let M^* be the set of all messages received by processors in Execution 2.

Contradicting Consistency. Now let us analyse the messages in M^*. Note that $\mathcal{F}(M_0) = \text{tr}_0$ and $\mathcal{F}(M_1) = \text{tr}_1$. Since Execution 2 is indistinguishable from Execution 1 for processors in P_1, it follows that $M_1 \subseteq M^*$. Since the processors in P_0 simulate Execution 0 in Execution 2, $M_0 \subseteq M^*$. So, $M_0 \subseteq M_0 \cup M_1 \subseteq M^*$ and $M_1 \subseteq M_0 \cup M_1 \subseteq M^*$. However, it cannot be the case that both $\mathcal{F}(M_0) \preceq \mathcal{F}(M_0 \cup M_1)$ and $\mathcal{F}(M_1) \preceq \mathcal{F}(M_0 \cup M_1)$: since $\mathcal{F}(M_0) = \text{tr}_0$ and $\mathcal{F}(M_1) = \text{tr}_1$ are distinct transactions, they can't both be prefixes of the same sequence. So, Consistency is not satisfied. □

Theorem 7.1 shows that SMR and TOB are not equivalent. To show that SMR is possible when $f < n/2$ in the lock-step model with signatures, we observe that it is actually simple to convert any protocol for TOB into a protocol for SMR when $f < n/2$. To see this, consider any protocol for TOB (satisfying the corresponding Liveness and Consistency requirements). Then proceed as follows:

- Add an instruction that, at every time-slot t, p_i should send the signed message $\langle \text{finalised}, \log_i(t) \rangle_i$ to all other processors.
- To specify $\mathcal{F}$, consider any set of messages M. Let σ be the longest sequence of transactions such that, for more than $n/2$ distinct values of i, M contains a message $\langle \text{finalised}, \tau \rangle_i$ such that $\sigma \preceq \tau$.

So, processors send out signed messages testifying to their present log. If any processor receives more than $n/2$ signed messages testifying to having finalised σ, the existence of this set of signed messages is proof that a correct processor has finalised σ. It's straightforward to check that Consistency and Liveness for the SMR protocol follow from satisfaction of the corresponding properties for the TOB protocol (you are invited to check).

Since any protocol for SMR solves TOB, the reduction above gives a precise sense in which the two problems are equivalent when $f < n/2$. In fact, this equivalence does not only hold in the lock-step model with signatures. It also holds in the partially synchronous and asynchronous models, which we will define in Chapter 8.

So far, we have considered reductions between SMR and TOB. In the next section, we'll consider reductions between SMR and BB/BA.

7.5 Reductions between SMR and BA/BB

In this section, we'll briefly consider how to transform protocols for SMR into protocols for BB and BA, and vice versa. Since we've already considered reductions between BA and BB in Chapter 4, it suffices to consider BB. As in all previous sections of this chapter, we consider the lock-step model with signatures.

Reducing SMR to BB. To reduce the task of solving SMR to solving BB, we enumerate some observations:

- We already saw in Section 7.2 that any protocol for BB that terminates in a known finite number of rounds can be used to solve TOB for any $f \leq n$.

- Exercise 7.1 walks you through a proof that (so long as the set of possible values V is finite) any deterministic protocol for BB can be straightforwardly converted into one that terminates in a known finite number of rounds (where that number may depend on Π, f, and V).
- We saw in Section 7.4 that any protocol for TOB can be straightforwardly transformed into a protocol for SMR when $f < n/2$. In the lock-step model with signatures, this is precisely when SMR is solvable.

Under the assumption that only finitely many distinct sequences of transactions can be received from the environment by any fixed time-slot t, these observations provide a straightforward conversion between BB protocols and SMR protocols if and only if $f < n/2$. The restriction on sequences of transactions received from the environment is only required if the given BB protocol does not already terminate in a known finite number of rounds.

Reducing BB to SMR. In the opposite direction, suppose we have an SMR protocol and we want to solve BB. The key insight is that we can use the SMR protocol to order values received from the broadcaster. We treat values received as transactions, run the SMR protocol to establish a total order of these 'input transactions', then output according to some deterministic rule applied to this ordered sequence.

A difficulty is that the broadcaster might not send any values: in this case, we need a bound on how long we should run the SMR protocol, after which processors can output some default value if their logs are empty. If the transaction set $\mathcal{T}$ is finite, Exercise 7.1 walks you through a straightforward way of converting any deterministic SMR protocol into one with a known liveness parameter $\ell(t)$, i.e., such that any transaction received by a correct processor by time-slot t will be finalised by all correct processors by $t+\ell(t)$. Whether or not the transaction set is finite, suppose we have an SMR protocol satisfying the condition that any transaction received by a correct processor by time-slot 1 will be finalised by all correct processors by some known time-slot, t_1 say, and that any transaction received by a correct processor by time-slot $t_1 + 1$ will be finalised by all correct processors by some known time-slot t_2. To solve BB with input set V given the latter protocol, we can then proceed as follows:

- Let the transaction set $\mathcal{T}$ be values in V signed by the broadcaster, together with the 'default' value $\perp$ signed by each processor;
- At time-slot 0, the broadcaster is instructed to send their signed value to all processors;
- From time-slot 1 until time-slot t_1 all processors execute the SMR protocol;
- At time-slot t_1, each processor is instructed to sign the default value $\perp$ and send it to all processors;
- At time-slot t_2, processors inspect the first $f + 1$ finalised values in their log. If any of these are transactions signed by the broadcaster, they output the first such value in their log. Otherwise, they output $\perp$.

This works because the SMR protocol ensures all correct processors agree on the same ordered sequence of transactions, and applying the same deterministic rule to this sequence ensures they produce the same BB output. Our choice of t_1 and t_2 ensures that

all correct processors have at least $f+1$ transactions in their log by t_2. If the broadcaster is correct, their signed value must appear in the finalised logs of all correct processors by t_1, and at most f other transactions can be finalised by any correct processor at that time.

7.6 Exercises

Exercise 7.1. *Consider strings of natural numbers, and say a set A of such strings is **downward closed** if, whenever $\sigma \in A$, all initial segments of σ also belong to A. We'll say an infinite string is a **path** through A if all finite initial segments of the infinite string belong to A. Let $|\sigma|$ denote the length of the finite string σ. If σ and τ are finite strings with $\sigma \preceq \tau$ and $|\tau| = |\sigma| + 1$, we say τ is a **one-element extension** of σ. The set of finite strings A is **finitely branching** if, for every $\sigma \in A$, there are only finitely many one-element extensions of σ in A.*

(i) Show that if A is a set of finite strings of natural numbers that is infinite, downward closed and finitely branching, then there exists an infinite string which is a path through A. Hint: build a path through A by recursion. Observe that the empty string (of length 0) has infinitely many extensions in A. Then argue that, since A is finitely branching and infinite, some string of length 1 must also have infinitely many extensions in A, and so on.

(ii) For the rest of the question, consider a fixed deterministic SMR protocol, $\mathcal{P}$ say. In Sections 7.1 and 7.3, we allowed transactions (as binary strings) to be of arbitrary length. However, let us now suppose that there are finitely many possible transactions, i.e., that $\mathcal{T}$ is finite. Consider executions of $\mathcal{P}$ in which all processors are correct and show that, for each t, there are only finitely many runs of length t (recall the definition of a run from Section 5.2.2).

(iii) Use (i) and (ii) above to argue that, for executions of the protocol in which all processors act correctly, there exists some bound $\ell(t)$ such that any transaction received by a correct processor by time-slot t must be finalised by $t + \ell(t)$.

(iv) Argue that, by ignoring overly long messages, $\mathcal{P}$ can be straightforwardly converted into a protocol for which the claim of (iii) also holds in a context where some processors may be Byzantine.

(v) Modify your arguments above to show that, when V is finite, every deterministic protocol solving BB can be straightforwardly converted into one that terminates in a finite number of rounds.

Exercise 7.2. *Write down a formal description of the protocol considered in Section 7.2, and prove that it solves SMR.*

Chapter 8

The asynchronous and partially synchronous models

So far, we have described results that hold for the lock-step and synchronous models. However, real distributed systems sometimes face unpredictable network delays, meaning that the timing assumptions of these models may be unrealistic. For this reason, we'll next introduce the asynchronous and partially synchronous models. The asynchronous model, described in Section 8.1, is the most difficult to operate in. In fact, we show in Section 8.2 that no deterministic protocol can solve BA, SMR or BB in this model. In Section 8.3, we then introduce the partially synchronous model, which may be seen as a middle ground - periods of reliable communication interrupted by arbitrary delays.

8.1 The asynchronous model

In fact, we'll define two simple variants of the asynchronous model. First, we consider the *asynchronous model with synchronised clocks*.

The asynchronous model with synchronised clocks. This is the same as the synchronous model, except that there is no bound on how long messages take to arrive. The only guarantee on message delivery is that any message sent by any correct processor p_i to any correct processor p_j must eventually be received at *some* time-slot.

While guaranteed message delivery may seem strong, it captures the essential property that reliable transport protocols (like TCP) provide in practice. The point is that eventual delivery between correct processors can be guaranteed by repeated resending.

We say 'clocks are synchronised' for this model, because every processor begins the execution simultaneously at time-slot 0 and carries out one transition (as specified by their state-transition-diagram) at each time-slot.[1] However, in the standard version of the asynchronous model, described next, we do not assume this lock-step form of computation.

The asynchronous model. Roughly, this is the same as the asynchronous model with synchronised clocks, except that we now allow arbitrary deviation in the rate at which

[1] Recall the state-transition-diagram model, as described in Chapter 3.

each processor's internal clock progresses. To formalise this idea, we expand the state-transition-diagram model of Chapter 3, by further stipulating that, at each time-slot, each processor may be *active* or *inactive*. If active, then the processor receives messages and carries out one step of the instructions as specified by the state-transition-diagram. If inactive at time-slot t, then the processor does not receive messages and performs no instructions, remaining in the same state until the next time-slot. In each execution, a correct processor may be active or inactive at arbitrary time-slots, but must be active at infinitely many time-slots: one may think of the allocation of active time-slots for p_i as being chosen by the adversary subject to this constraint.

The guarantee for message delivery is that, if correct p_i sends a message to correct p_j, then there must exist some time-slot at which p_j (is active and) receives the message.

Why two models of asynchrony? We consider both variants because our main impossibility result for the asynchronous model holds even with synchronised clocks, which is particularly relevant given modern clock synchronisation capabilities. In this book, all positive results that hold for the asynchronous model with synchronised clocks will also hold for the standard asynchronous model without synchronised clocks (synchronised clocks are not very useful when message delays are arbitrary).

Having established these models, we can now ask: do the positive results for the lock-step and synchronous models extend to the asynchronous model? The answer, as we'll prove next, is surprising and fundamental to understanding the limits of distributed computation.

8.2 Deterministic consensus is not possible in asynchrony

In this section, we prove the following theorem.

Theorem 8.1. *Consider the asynchronous model with synchronised clocks, crash-faults, and with signatures. No deterministic protocol solves (binary) BA for $f \geq 1$ and $n \geq 2$.*

Theorem 8.1 is often known as the 'FLP Theorem', because a version for the asynchronous model (without synchronised clocks) was first proved by Fischer, Lynch and Paterson [13]. Here we'll give a simpler proof, which follows the approach used in a blog post by Abraham and Stern,[2] itself based on approaches developed by Gafni and Losa [14] and Volzer [15].

High level intuition. Recall that, in Section 5.2, we proved that $f + 1$ rounds are necessary for protocols solving BA in the lock-step model. To do so, we considered the notion of a *k-run*, which is just a description of the first k time-slots of an execution. We also defined the notion of a *bivalent* run, which is a run for which the output of correct processors has not yet been determined, i.e., a run r for which there exist executions E and E' extending r such that correct processors output differently in E than in E'. We started by showing that, for any protocol, there must exist a 0-run (i.e., a set of inputs) that is bivalent. Then we carried out a proof by induction to show that some f-run must be bivalent.

[2] https://decentralizedthoughts.github.io/2024-03-07-mobile-is-FLP/

At a high level, the approach we take to prove Theorem 8.1 is very similar. We start by showing that some 0-run must be bivalent. Now, however, we carry out a proof which recursively builds a bivalent *execution*, i.e., an execution in which correct processors never output.

To describe the proof, we first recall the precise definition of a k-run, and also introduce some other useful terminology.

8.2.1 k-runs and pivots

Towards a contradiction, we suppose given a protocol $\mathcal{P}$ that solves binary BA in the asynchronous model with signatures, synchronised clocks, and crash-faults, when $f = 1$ and $n \geq 2$. Fixing some $n \geq 2$, we let $\mathcal{E}$ be the set of all executions of $\mathcal{P}$ in this model, for a given set of processors Π with $|\Pi| = n$.

Defining runs. We consider a notion of k-run which is adapted slightly from that in Section 5.2.2 to incorporate the fact that we are now working in a different model:

- By a 0-run (think of a 'run of length 0'), we mean a specification of the input (0 or 1) to each processor.
- If $k \geq 1$, by a k-run we mean a specification of:
  - (i) The input to each processor;
  - (ii) The messages sent by each processor at each time-slot $< k$;
  - (iii) The messages received by each processor at each time-slot $\leq k$, and;
  - (iv) Which processors crashed at each time-slot $< k$.

By a *run*, we mean a k-run, for some k. As in Section 5.2.2, we define whether one run *extends* another in the obvious way. If r is a k-run and r' is a k'-run with $k' \geq k$, we say r' extends r if: (i) in r', all processors receive the same inputs as in r; (ii) at each time-slot $< k$, the messages sent by each processor are the same in r' as in r; (iii) at each time-slot $\leq k$, the messages received by each processor are the same in r' as in r, and; (iv) at each time-slot $< k$, the same processors crash in r' and r. If these conditions are satisfied and $k' > k$, then r' *properly extends* r. We also extend this terminology in the obvious way to executions. So, if $E \in \mathcal{E}$ is an execution and r is a k-run, we say E extends (or is an *extension* of) r if the conditions above hold when r' is replaced by E.

It will also be useful to consider the new notion of a *pivot*, which is a particular type of bivalent run. We define this next.

Pivots. To specify pivots precisely, we make the following definitions:

- A run r is *crash-free* if no processor crashes in r. We let R_k be the set of crash-free k-runs extended by some execution in $\mathcal{E}$, and set $R = \bigcup_{k \in \mathbb{N}} R_k$.
- If $r \in R_k$, then $E(r)$ is the unique execution in $\mathcal{E}$ extending r in which no processor crashes, messages sent in r but not received in r are received at time-slot $k+1$, and all messages sent at time-slots $\geq k$ are received at the next time-slot. So, roughly, $E(r)$ is the extension of r in which no processor crashes, and message delivery is 'lock-step' after time-slot k.

- If $r \in R$, we let $\text{val}(r)$ be the value output by all processors in $E(r)$. This value must be well-defined, because $E(r) \in \mathcal{E}$ and $\mathcal{P}$ satisfies Termination and Agreement.
- If $r \in R_k$ then $E(r-p)$ is the unique execution in $\mathcal{E}$ extending r in which p crashes without sending any messages at time-slot k, messages sent to processors other than p in r but not received in r are received at time-slot $k+1$, and all messages sent to processors other than p at time-slots $\geq k$ are received at the next time-slot. Since $f = 1$, this means no processor other than p crashes in $E(r-p)$. So, roughly, $E(r-p)$ is the extension of r in which only p crashes, and message delivery is otherwise 'lock-step' after time-slot k.
- If $r \in R$, we let $\text{val}(r-p)$ be the value output by all correct processors in $E(r-p)$. Again, this value must be well-defined, because $E(r-p) \in \mathcal{E}$ and $\mathcal{P}$ satisfies Termination and Agreement.
- We say r is a *p-pivot* if $r \in R$ and $\text{val}(r) \neq \text{val}(r-p)$. If r is a p-pivot for some $p \in \Pi$, we also say r is a *pivot*.

So, if $r \in R_k$ is a p-pivot, this means r is a specific form of bivalent run: if we restrict to executions in which no processor other than (perhaps) p crashes and message delivery to non-crashed processors is lock-step after k, then whether or not p crashes at time-slot k changes how correct processors output.

With these definitions in place, we are ready to describe the formal proof.

8.2.2 The proof of Theorem 8.1

Let $\mathcal{P}$ be as specified in Section 8.2.1. As hinted at previously, the basic form of the proof is simple. We start by showing that some 0-run is a pivot. Then we show that each pivot can be properly extended to give another pivot. Iterating this argument produces an execution in $\mathcal{E}$ in which correct processors do not output.

We begin by showing that some 0-run is a pivot, using a proof very similar to the proof of Lemma 5.3 in Section 5.2.5.

Lemma 8.2. *Some 0-run is a pivot.*

Proof. The proof is very similar to the proof of Lemma 5.3. Towards a contradiction, suppose no 0-run is a pivot. Recall that the set of processors is $\Pi = \{p_0, \ldots, p_{n-1}\}$. For each $i \in [0, n]$, let r_i be the 0-run in which all processors p_j for $j < i$ receive input 1, while all other processors receive input 0. Since $\mathcal{P}$ satisfies Validity, it follows that $\text{val}(r_0) = 0$, while $\text{val}(r_n) = 1$. From this it follows that there exists $i \in [0, n)$ such that $\text{val}(r_i) = 0$, while $\text{val}(r_{i+1}) = 1$. The only difference between r_i and r_{i+1} is the input received by p_i. Since $E(r_i - p_i)$ and $E(r_{i+1} - p_i)$ are indistinguishable for processors other than p_i, $\text{val}(r_i - p_i) = \text{val}(r_{i+1} - p_i)$. So, either $\text{val}(r_i - p_i) = 1 \neq \text{val}(r_i)$ or $\text{val}(r_{i+1} - p_i) = 0 \neq \text{val}(r_{i+1})$, meaning that either r_i is a p_i-pivot, or r_{i+1} is a p_i-pivot. This gives the required contradiction. □

Next, we show that each pivot can be properly extended to give another pivot. Claim (ii) of the following lemma will be used to ensure that repeated applications of the lemma produce a valid execution without infinite message delays.

Lemma 8.3. *If r is a p-pivot, then it has a proper extension r' satisfying:*

(i) The run r' is a p'-pivot for $p' \neq p$.

(ii) All messages sent by processors other than p in r' are received in r'.

Proof. The proof of Lemma 8.3 is divided into two parts. Suppose r is a p-pivot. First, we use an argument similar to the first part of the proof of Lemma 8.2 to produce r_a and r_b, which are two proper extensions of r with $\text{val}(r_a) \neq \text{val}(r_b)$. Then, for the second part, we also use an argument similar to the proof of Lemma 8.2. We describe a sequence of runs $r_a = r_0, \ldots, r_{n-1} = r_b$, such that each r_i differs from r_{i+1} by when a single processor receives certain messages from p. Since there must exist i with $\text{val}(r_i) \neq \text{val}(r_{i+1})$, this allows us to argue that either r_i or r_{i+1} is a p'-pivot for some $p' \neq p$. Now let us fill in the details.

Part 1: specifying r_a and r_b. Let k be such that $r \in R_k$. Since r is a p-pivot, we have that $\text{val}(r) \neq \text{val}(r - p)$. The rough idea in specifying r_a and r_b is to use the fact that all correct processors must output by some finite time in $E(r - p)$. If we define r^*_ℓ (for each $\ell \geq 1$) to proceed in lock-step at time-slots after k, except that messages sent by p at time-slots $\geq k$ are not received until time-slot $k + \ell$, then for all sufficiently large ℓ, we must have $\text{val}(r^*_\ell) = \text{val}(r - p)$. Since $\text{val}(r) = \text{val}(r^*_1) \neq \text{val}(r^*_\ell) = \text{val}(r - p)$ for sufficiently large ℓ, there must exist some least ℓ with $\text{val}(r^*_\ell) \neq \text{val}(r^*_{\ell+1})$. Then we can use r^*_ℓ and $r^*_{\ell+1}$ (or appropriate modifications of these runs) as r_a and r_b. Figure 8.1 illustrates the idea.

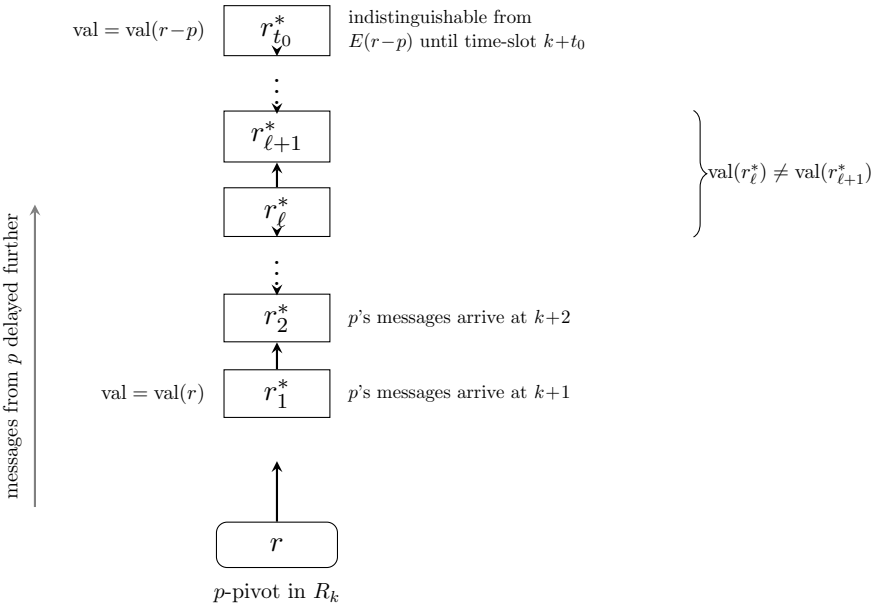


FIGURE 8.1: Part 1 of the proof of Lemma 8.3. Starting from the p-pivot $r \in R_k$, we construct a sequence of runs $r^*_1, r^*_2, \ldots$ in which all processors act in lock-step after time-slot k, except that messages from p are delayed until time-slot $k + \ell$. At one end, $\text{val}(r^*_1) = \text{val}(r)$. For sufficiently large t_0, $\text{val}(r^*_{t_0}) = \text{val}(r - p) \neq \text{val}(r)$. There must therefore exist some least ℓ at which the valency changes.

More precisely, for each $\ell \geq 1$ we define r^*_ℓ to be the run in $R_{k+\ell}$ extending r, such that messages sent in r but not received in r are received at time-slot $k + 1$, messages sent

by processors other than p at time-slots $\geq k$ are received at the next time-slot, and all messages sent by p at time-slots $\geq k$ are received at time-slot $k+\ell$. So, r^*_ℓ proceeds in 'lock-step' at time-slots after k, except that messages from p are delayed until time-slot $k+\ell$. Note that $E(r) = E(r^*_1)$, so that $\text{val}(r) = \text{val}(r^*_1)$.

Since $\mathcal{P}$ satisfies Termination, we can take t_0 sufficiently large that all correct processors output before time-slot $k+t_0$ in $E(r-p)$. Note that $E(r-p)$ is indistinguishable from $E(r^*_{t_0})$ until time-slot $k+t_0$ for processors other than p, i.e., all other processors receive the same inputs and the same messages at time-slots $< k+t_0$. It follows that $\text{val}(r) = \text{val}(r^*_1) \neq \text{val}(r-p) = \text{val}(r^*_{t_0})$. So, there must exist some least $\ell \geq 1$ with $\text{val}(r^*_\ell) \neq \text{val}(r^*_{\ell+1})$. We set $r_a = r^*_\ell \in R_{k+\ell}$, and we set r_b to be the element of $R_{k+\ell}$ extended by $r^*_{\ell+1}$.

Part 2: finding r'. Without loss of generality suppose the processors are numbered so that $p = p_{n-1}$. Let r_a and r_b be defined as above. For each $i \in [0, n-1]$, let $r_i \in R_{k+\ell}$ be the same as r_a, except that, for each $0 \leq j < i$, processor p_j does not receive messages from p at time-slot $k+\ell$. Note that $r_a = r_0$ and $r_{n-1} = r_b$. It follows that there exists some least i with $\text{val}(r_i) \neq \text{val}(r_{i+1})$. However, the only difference between r_i and r_{i+1} is the messages received by p_i, which means that $E(r_i - p_i)$ and $E(r_{i+1} - p_i)$ are indistinguishable for processors other than p_i. So, $\text{val}(r_i - p_i) = \text{val}(r_{i+1} - p_i)$. It follows that either $\text{val}(r_i - p_i) \neq \text{val}(r_i)$ or $\text{val}(r_{i+1} - p_i) \neq \text{val}(r_{i+1})$, meaning that either r_i is a p_i-pivot, or r_{i+1} is a p_i-pivot. Note also that $p_i \neq p$ and that, for $r' \in \{r_i, r_{i+1}\}$, all messages sent by processors other than p in r' are received in r'. So r' exists, as claimed. □

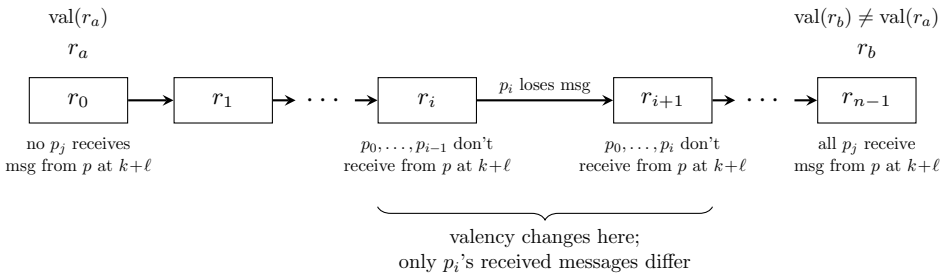


FIGURE 8.2: Part 2 of the proof of Lemma 8.3. Starting from $r_a = r_0$ and $r_b = r_{n-1}$ (with $\text{val}(r_a) \neq \text{val}(r_b)$), we construct a sequence of runs in $R_{k+\ell}$. Each r_{i+1} is the same as r_i, except that p_i does not receive a message from p at time-slot $k+\ell$. Since the valency differs at the two ends, there must exist adjacent runs r_i and r_{i+1} of different valency. Since these runs differ only in the messages received by p_i, it follows that either r_i or r_{i+1} is a p_i-pivot.

Completing the proof of Theorem 8.1. Combining Lemmas 8.2 and 8.3 produces an infinite sequence of pivots $r_0, r_1, \ldots$, such that each r_{i+1} properly extends r_i. If r_i is a p-pivot, then r_{i+1} is a p'-pivot for $p' \neq p$, and all messages sent by processors other than p in r_{i+1} are received in r_{i+1}. It follows that any message sent in r_i is received in r_{i+1}, so that the sequence $r_0, r_1, \ldots$ specifies a valid execution in $\mathcal{E}$ in which all messages are eventually received and correct processors do not output. □

How about SMR and BB? While Theorem 8.1 is stated for binary BA, it is straightforward to modify the proof to give the same result for SMR (Exercise 8.1). In Section

8.4, we will give a direct proof that (unlike BA and SMR) no protocol solves BB for $f \geq 1$ in the partially synchronous model: this implies that BB is also impossible in asynchrony.

The FLP Theorem shows that deterministic consensus faces insurmountable barriers in asynchronous settings. However, as we'll see next, adding even minimal timing assumptions can restore the possibility of deterministic consensus.

8.3 Defining the partially synchronous model

The partially synchronous model was introduced by Dwork, Lynch and Stockmeyer [16]. The model captures a realistic network setting where SMR protocols must maintain Consistency even during periods of unreliable communication, whilst finalising new transactions given sufficiently long periods of synchrony. The formal definition employs a technical approach that simplifies theorem proving but can initially seem disconnected from this intuitive description. We'll present the formal definition first, then explain how it captures the intended behaviour.

As for the asynchronous model, we'll define two variants of the partially synchronous model: one with synchronised clocks, and then the standard version, which does not assume synchronised clocks.

The partially synchronous model with synchronised clocks. This model is the same as the synchronous model, except that message delivery guarantees are now as follows: there exists a known bound Δ (given as input to each processor) and an unknown time-slot called the *global stabilisation time* (GST), such that any message sent at any time-slot t is received by time-slot $\max\{t, \text{GST}\} + \Delta$.

So, in this model there is no known upper bound on message delivery times, but, after some unknown point that may vary between executions, messages will always be delivered within Δ time-slots. It is common to think of GST as chosen by the adversary.

The actual (unknown) upper bound on message delays δ. The parameter Δ is a *known* upper bound on message delays after GST. Often, it will also be useful to consider δ which is the *actual* (unknown) least upper bound on message delays after GST.

Matching the intuition. How does this formal definition capture the intuitive picture described above? Suppose first that we have an SMR protocol that satisfies Consistency and Liveness in the formal model defined above. Then Consistency must be maintained in asynchrony, because any Consistency violation must occur within a finite amount of time. Since GST is arbitrarily large and unknown, an asynchronous execution in which Consistency is violated gives an execution consistent with the formal model above in which there is also such a violation: run the asynchronous execution until Consistency is violated, and then make message delivery lock-step thereafter. Equally, the protocol must finalise new transactions given any sufficiently long synchronous interval, because the adversary may choose GST to be the start of the given interval, and then Liveness requires that new transactions be finalised in a finite amount of time.

Conversely, suppose we have a protocol that maintains Consistency in asynchrony and finalises transactions during sufficiently long synchronous periods. This protocol automatically works in the formal model we have defined: Consistency is preserved because the protocol handles arbitrary message delays (which includes the pre-GST period), whilst Liveness is ensured because any execution eventually reaches GST, after which the protocol operates under the synchronous conditions it requires for progress.

Next, we define the standard version of the partially synchronous model, which does not assume synchronised clocks.

The partially synchronous model. Recall the notion of inactive time-slots, introduced in Section 8.1. The partially synchronous model is the same as the partially synchronous model with synchronised clocks, except that correct processors may be inactive at any arbitrary set of time-slots prior to GST (while being active at all other time-slots, including all time-slots $\geq$ GST). It is common to think of the adversary as choosing the time-slots before GST at which processors are inactive.

All of the impossibility results we prove for the partially synchronous model in this book will also hold for the variant with synchronised clocks. Sometimes it will be convenient to prove positive results first for the variant with synchronised clocks, and then show how to modify the proof to deal with the version without synchronised clocks.

Other versions of the partially synchronous model. Another variant of the model allows correct processors to have 'clock speeds' after GST that may be different from each other by some bounded amount. Formally, we might specify some known values $d_1 < d_2$ and require that, in each interval of d_2 time-slots after GST, each correct processor is active in at least d_1 time-slots. Clearly, any impossibility result for our version of partial synchrony also holds for this variant. The proofs of all of our positive results for the standard partial synchrony model can also be straightforwardly transformed to give a proof for this variant.

Yet another model that is sometimes referred to as the partially synchronous model drops the concept of GST altogether. In this model, some bound Δ on message delivery times always holds. However, Δ is now unknown, i.e., Δ is not given to the processors as input. This means a protocol must function for any finite Δ without knowing this value. As always, one could consider various versions of this model, with or without synchronised clocks. For the sake of simplicity, let us consider a version with synchronised clocks, and let's call this the 'unknown Δ' model.

So long as we are not concerned with matters of efficiency, the partially synchronous model with synchronised clocks and the 'unknown Δ' model are equivalent: we can convert a protocol for one model into one for the other. A protocol for the 'unknown Δ' model is already a protocol for the partially synchronous model with synchronised clocks because, in any execution consistent with the latter model, all messages are delivered within $\Delta' :=$GST$+\Delta$ time-slots. In the other direction, suppose we have a protocol for the partially synchronous model with synchronised clocks. To convert this into a protocol for the 'unknown Δ' model, we can give processors the input $\Delta = 1$, and then insert increasing intervals between the time-slots at which correct processors execute their instructions, e.g., processors might implement the instructions for time t at real time 2^t (efficiency considerations aside). Although the *actual* bound Δ on message delivery times is unknown, the interval between time-slots at which instructions are implemented is eventually larger than Δ.

In the next section, we consider for which f and n consensus is possible in the partially synchronous model.

8.4 When is consensus possible in partial synchrony?

We start with a negative result for BB.

Theorem 8.4. *Consider the partially synchronous model with synchronised clocks, crash-faults, and with signatures. No protocol solves Crash-fault Broadcast if $f \geq 1$ and $n \geq 2$.*

Proof. The proof is a simple indistinguishability argument. We suppose such a protocol exists, and consider two executions of the protocol with $\Delta = 1$, and with the same processor acting as broadcaster.

Execution 1. The broadcaster is faulty and crashes at time-slot 0. GST $= 0$. To satisfy Termination and Agreement, all other processors must eventually give the same output. Without loss of generality, suppose they all output 0 by some time-slot, t^* say.

Execution 2. All processors are correct. The broadcaster has input 1. GST $> t^*$. Prior to GST, messages between all processors other than the broadcaster are received at the time-slot after sending, but messages from the broadcaster are not received until GST.

Prior to GST, Executions 1 and 2 are indistinguishable for processors other than the broadcaster. So, processors other than the broadcaster output 0 in Execution 2, i.e., they give the same output as in Execution 1. Validity is therefore violated. □

Having shown BB is impossible with even one crash-fault, we now examine BA and SMR. Dwork, Lynch, and Stockmeyer showed that, for Byzantine faults, SMR and BA are solvable if and only if $f < n/3$ [16]. Next, we prove the negative part of that result. The theorem is stated and proved explicitly for BA, but the proof is straightforwardly modified to give the same result for SMR (Exercise 8.2). Of course, the impossibility result for the case with synchronised clocks immediately gives the result for the case without synchronised clocks.

Theorem 8.5. *Consider the partially synchronous model with synchronised clocks, Byzantine faults, and with signatures. No protocol solves BA if $f \geq n/3$.*

Proof. Towards a contradiction, suppose such a protocol exists. Fix n and f, with $f \geq n/3$. Partition Π into three sets, P_0, P_1 and P_2, such that $|P_2| = f$, while $|P_0| \leq f$ and $|P_1| \leq f$. We consider three executions of the protocol with $\Delta = 1$.

Execution 0. All processors have input 0. Processors in P_1 are faulty and perform no action, while other processors are correct. GST $= 0$. All correct processors must output 0. Suppose they all do so by t_0.

Execution 1. All processors have input 1. Processors in P_0 are faulty and perform no action, while other processors are correct. GST $= 0$. All correct processors must output 1. Suppose they all do so by t_1.

Execution 2. Processors in P_0 receive input 0 (as in Execution 0), while processors in P_1 receive input 1 (as in Execution 1). The inputs to processors in P_2 are unimportant (0, say). Processors in $P_0 \cup P_1$ are correct, while processors in P_2 are faulty. GST $> \max\{t_0, t_1\}$. Prior to GST, message delivery is as follows:

- Messages between processors in $P_0 \cup P_2$ are received the time-slot after sending.
- Messages between processors in $P_1 \cup P_2$ are received the time-slot after sending.
- Messages between processors in P_0 and processors in P_1 (in either direction) are not received until GST.

Prior to GST, processors in P_2 'simulate' Execution 0 for processors in P_0, i.e., they send them precisely the same messages at the same time-slots. They can do this, even though we consider a model with signatures, because processors in P_2 receive the same messages from processors in P_0 as in Execution 0. Similarly, processors in P_2 'simulate' Execution 1 for processors in P_1, i.e., they send them precisely the same messages at the same time-slots.

Analysis. Note that, in Execution 2, the task of simulating Execution 0 just requires processors in P_2 to send precisely the same messages to processors in P_0 that they would if correct with input 0, and if receiving no messages from processors in P_1. Similarly, simulating Execution 1 just requires emulating correct behaviour if given input 1 and not receiving messages from processors in P_0.

The crucial point is that, until GST, Execution 2 is indistinguishable from Execution 0 for processors in P_0, i.e., those processors receive the same inputs and receive precisely the same messages at each time-slot. Since they output 0 prior to t_0 in Execution 0, they must do the same in Execution 2. Symmetrically, processors in P_1 must output 1, violating Agreement. □

In the next chapter, we'll describe a simple protocol (a variant of Tendermint [17]) that solves SMR in the partially synchronous model with synchronised clocks and signatures when $f < n/3$. Solving SMR in this setting also suffices to solve BA:

- Given an SMR protocol satisfying Consistency and Liveness when $f < n/3$, let the transaction set $\mathcal{T}$ be values in the input set V signed by processors in Π.
- Require correct processors to send their signed input to all other processors at time-slot 0.
- Run the SMR protocol from time-slot 1, until some shortest sequence of transactions σ is finalised that includes values in V signed by $n - f$ different processors.
- Consider the set of 'voting' processors to be those with signed values in σ. Each voting processor 'votes for' their first signed value in σ. Determine the output by majority vote amongst voting processors, breaking ties by some predetermined rule.

By Consistency and Liveness for the SMR protocol, the process above terminates, with all correct processors giving the same output. So, Termination and Agreement are

satisfied. If all correct processors have the same input, v say, then at least $n-2f$ of the $n-f$ voting processors must vote for v. Since $n > 3f$, this is a majority of the voting processors, so all correct processors output v. Validity is therefore satisfied.

Next, we turn to protocol design, beginning with a streamlined version of Tendermint that illustrates some of the key principles underlying modern partially synchronous SMR protocols.

8.5 Exercises

Exercise 8.1. *Prove a version of Theorem 8.1 for SMR.*

Exercise 8.2. *Modify the proof of Theorem 8.5 to show that no protocol solves SMR in the partially synchronous model when $f \geq n/3$.*

Exercise 8.3. *So far, we have considered SMR protocols that satisfy both Consistency and Liveness for the same value of f. This question considers how one can generalise this requirement.*

Let us say a protocol has ***consistency resilience*** *ρ_C if it satisfies Consistency so long as at most ρ_C processors are faulty. Similarly, let us say that a protocol has* ***liveness resilience*** *ρ_L if it satisfies Liveness so long as at most ρ_L processors are faulty. Modify the proof of Theorem 8.5 to show that, in the partially synchronous model, if any protocol has consistency resilience ρ_C and liveness resilience ρ_L, then $\rho_C + 2\rho_L < n$.*

Chapter 9

Tendermint

In this chapter, we introduce Tendermint [18], which is a very simple protocol solving SMR when $f < n/3$ in the partially synchronous model. Tendermint was first described by Ethan Buchman in his PhD thesis in 2016 [17].

In Section 9.2, we first describe a version of Tendermint that assumes synchronised clocks. In Section 9.3, we then describe a 'pipelined' version of the protocol, which overlaps consensus instances to increase throughput whilst maintaining safety guarantees. This pipelined version of the protocol is essentially Casper [19], the finality mechanism presently employed by the Ethereum blockchain. Finally, in Section 9.4, we show how to modify the protocol to work in the partially synchronous model without synchronised clocks.

We begin with synchronised clocks to focus on the core consensus mechanism. However, before we get to Tendermint, we first need to cover some standard techniques: the use of *collision-free hash functions* and *quorum intersection* arguments. We introduce these in the next section.

9.1 Preliminary techniques

In this section, we discuss some standard techniques that will be useful throughout the remainder of the book. First, we introduce collision-free hash functions.

9.1.1 Building 'blockchains' with collision-free hash functions

In Section 7.2, we described a simple protocol for solving TOB in the lock-step model, which can also be straightforwardly converted into a protocol for SMR when $f < n/2$ (see Section 7.4). At a high level, the approach taken was as follows:

- We divide the execution into *views*. In each view, we carry out a single instance of Dolev-Strong.
- We let the *leader* for view v be p_i, where $i = v \bmod n$. So, leaders are 'rotating'.

- If p_i is the leader for view v, then they act as the broadcaster for this instance of the Dolev-Strong protocol. Upon starting the view, they collect all transactions received by the start of the view and not yet appended to their log. The 'value' they broadcast is a signed list of these transactions, indexed by the view.

With this approach, processors reach consensus on the result of each view: either all correct processors agree on a new block of transactions proposed by the leader of the view, or else all correct processors agree that the view does not finalise new transactions. This meant that it was not necessary for each new leader to specify which previous block their new proposal should extend: at the start of the view, correct processors have already reached consensus on this value.

The need for unique 'pointers'. A downside of this 'Dolev-Strong for each view' approach is that it is inefficient, requiring $f + 1$ rounds of communication in each view. Going forward, we want to define protocols for which each view requires only a small constant number of rounds. Moreover, we now need a protocol that functions in the partially synchronous model, where we have already seen that BB is not possible (see Section 8.4). In this context, it will be convenient for the leader of each view to be able to 'point to' the previous block of transactions that their new block is supposed to extend. How can they point to such a block? One method would be to simply include the previous block inside the new proposed block. However, this would mean blocks quickly grow in size: sending each new block would now involve sending all previous blocks.

What we need is a short 'digest' of the previous block that can be used as a pointer. To act correctly as a pointer, this short digest must be unique (or 'collision-free'). This is the functionality provided by collision-free hash functions.

Collision-free hash functions. For our purposes, a *hash* function is just a function that maps binary strings of arbitrary length to binary strings of some fixed length λ, where λ is a tuneable parameter. Since there are only a finite number of binary strings of a given length λ, the function cannot actually be injective, i.e., there must exist distinct strings which are mapped to the same value. However, this does not necessarily mean that finding such collisions is a computationally easy task. Roughly, when we say that a hash function is *collision-resistant*, this means that finding collisions is not a computationally feasible task. For example, SHA256 is a well-known hash function that maps any string to a 256 bit output, and for which no collision has been found to date.

In this book, we'll black box the cryptography behind hash functions and, for practical simplicity, we'll just suppose given a hash function H which is *ideal* in the sense that it is *collision-free*, i.e., we suppose H is injective. Formally, this means we restrict attention to executions in which the hash values of all 'relevant' messages are unique.[1]

Building blockchains. As hinted at above, one basic use of the hash function H will be in allowing each new 'block' of transactions to 'point to' a previous block. To achieve this, we begin with a *genesis* block, which is known to all processors at the start of the protocol execution, and which does not point to any previous block. Any block b other than the genesis block is required to point to some previous block to be *valid*: b points to previous block b' (such as the genesis block) simply by including $H(b')$ in some specified

[1] The 'relevant' messages will be protocol specific. For Tendermint, the relevant messages will be all blocks received by at least one processor.

place amongst its data. In this case, we say b' is the (unique) *parent* of b, and that b is a *child* of b'. We also talk in terms of *ancestors* and *descendants*. The genesis block has only itself as an ancestor. The *ancestors* of any other block b with parent b' are b and all ancestors of b': the genesis block is required to be amongst the ancestors of b for it to be a valid block. In this way, the ancestors of any block b form a 'blockchain' - a finite chain of blocks each pointing to its predecessor. Two blocks are *incompatible* if neither is an ancestor of the other. The *descendants* of any block b are b and all descendants of the children of b.

9.1.2 Quorum intersection arguments

The use of quorum intersection arguments, which we already saw when discussing the Phase-King protocol in Section 6.2, will be our 'bread and butter' when building protocols from this point on. Essentially, quorum intersection arguments are just simple counting arguments showing that, when the number of faulty processors is sufficiently bounded, voting procedures cannot result in multiple voting options receiving large majorities.

To make this concrete, let us consider the use of quorum intersection arguments in the case of Tendermint. Just like the previous approach that used repeated instances of Dolev-Strong, the Tendermint protocol divides the instructions into views, with each view having a leader. In each view, the leader will propose some new block of transactions b, and (if b is appropriately formed) other processors will then send out *votes* for b. If the leader is Byzantine and sends different blocks to different processors, then correct processors may vote for different blocks, but each correct processor will be instructed to vote for at most one block in each view. In this case, a simple counting argument shows that (assuming $f < n/3$) at most one block can receive votes from at least $n - f$ processors in each view. The argument is the same as that we used to establish 'Time-slot 1 Agreement' for the Phase-King protocol. Towards a contradiction, we suppose two distinct blocks b and b' both receive votes from $n - f$ processors:

- Let P be the set of processors that vote for b, so $|P| \geq n - f$;
- Let P' be the set of processors that vote for b', so $|P'| \geq n - f$;
- The intersection $|P \cap P'| \geq |P| + |P'| - n \geq (n - f) + (n - f) - n = n - 2f$;
- Since $f < n/3$, we have $n - 2f > f$;
- Therefore $|P \cap P'| > f$, meaning $P \cap P'$ contains at least $f + 1$ processors;
- Since at most f processors are faulty, $P \cap P'$ contains at least one correct processor;
- This correct processor votes for both b and b', contradicting the fact that correct processors vote for at most one block in each view.

We'll call a set of votes for b by $n - f$ processors a *quorum certificate* (QC) for b. So, the argument above shows that at most one block for each view can receive a QC.

The reason that SMR (and BA) are possible in the partially synchronous model if and only if $f < n/3$ is because it is precisely in this regime that the quorum intersection

argument above is valid. In fact, we'll see in Chapter 16 that protocols using randomness in the asynchronous model can also solve SMR (and BA) if and only if $f < n/3$, and this is true for the same reason.

Having established the essential tools of hash-based blockchain construction and quorum intersection arguments, we can now describe how these techniques combine to create an efficient SMR protocol.

9.2 Tendermint with synchronised clocks

To explain the protocol, we first informally describe an overly simple approach that does not work.

9.2.1 A simple (but failed) attempt

As described previously, the instructions will be divided into views, with each view having a designated leader. The leader for view v will be processor p_i, where $i = v \bmod n$, and blocks will be ordered by corresponding view number. One simple approach would be to consider blocks finalised upon receiving at least $n - f$ votes for them, i.e., upon receiving a QC for the block, and to proceed as follows within each view:

1. Each view is of length 2Δ. When the view begins (at time-slot $t = 2v\Delta$), the leader produces and sends a new block of transactions to all processors. This block includes the view number and should be a child of (point to) the greatest finalised block amongst those they have seen.
2. At time-slot $t + \Delta$, processors each consider the first block (if any) they have received from the leader. If it is a descendant of the greatest finalised block they have seen, then they produce a signed vote for that block and send that vote to all processors.

This approach is certainly simple. In the partially synchronous model, however, it is especially easy to see that such a protocol will not satisfy Consistency, i.e., incompatible blocks might become finalised. For example, a block b might receive votes from at least $n - f$ processors while in view v, without any correct processors actually *seeing* the QC by the end of the view. This might happen simply because the view takes place before GST (recall that GST is unknown), which allows for the possibility that the votes are sent but not received by some (or all) correct processors until after GST. Then a block which is incompatible with b might be proposed in view $v + 1$ and might also become finalised.

9.2.2 Using two stages of voting (informal analysis)

What happens if we consider two stages of voting in each view instead? Let us suppose that, if they see a QC for b produced in a first round of voting, a QC of 'stage 1' votes say, processors then produce a 'stage 2' vote. A QC of stage 2 votes is now required

for finalisation. The problem with the protocol from Section 9.2.1 was that a block might be finalised without correct processors actually seeing the QC. With our new approach, it is at least the case that, if the view v block b is finalised, then all correct processors producing stage 2 votes for b have seen the stage 1 QC. Suppose we now instruct processors who have seen the stage 1 QC for b to *lock* on b: p being locked on b does not yet mean that p has finalised b, but means that (so long as the lock is in place) p will not vote for subsequent proposals that are incompatible with b. It is not hard to see that this new rule produces Consistency:

- If b receives a stage 1 QC in view v then, by our previous quorum intersection analysis in Section 9.1.2, no other block can receive a stage 1 QC in view v. This means no other block can receive a stage 2 QC in view v, since no correct processor produces a stage 2 vote for any block without first seeing a stage 1 QC for that block.
- If b is finalised in view v, at least $n - f$ processors produce stage 2 votes for b, and so become locked on b if correct.
- Since at least $n - f$ processors are required to produce any stage 1 QC, this makes it impossible for any block b' incompatible with b to receive a stage 1 QC (or a stage 2 QC) in subsequent views:
  - By the same argument as in our previous quorum intersection analysis in Section 9.1.2, any set of $n - f$ processors producing stage 2 votes for b and any set of $n - f$ processors producing stage 1 votes for b' must have a correct processor p in the intersection.
  - This gives a contradiction, since p being locked on b means that it would not produce the stage 1 vote for b'.

In a bit more detail, we now proceed as follows. First, a small technical convenience: the genesis block is treated as a special case, in the sense that the empty set of votes is considered a stage 1 QC for the genesis block. This means that all processors begin the execution having received a stage 1 QC for the genesis block. The genesis block is considered a block for view 0. If Q is a stage 1 or 2 QC for b, then Q.view is the view corresponding to b. Like blocks, stage 1 QCs are ordered by view. Each processor now maintains a local variable `lock`, which is always a stage 1 QC for some block, and is initially set to be the empty set, i.e., a stage 1 QC for the genesis block. Blocks are finalised upon receiving a stage 2 QC. Views are now of length 3Δ. In each view $v > 0$, we proceed as follows:

1. When the view $v > 0$ begins (at time-slot $t = 3v\Delta$), the leader finds Q which is the greatest stage 1 QC it has received. Suppose Q is a stage 1 QC for b'. The leader then produces a new proposal with parent b' and sends this to all processors. The proposal includes Q.
2. At time-slot $t + \Delta$, processors only vote for the first proposal b received from the leader if it includes Q which is a stage 1 QC for the parent b', and if Q.view $\geq$ `lock`.view.
3. At time-slot $t + 2\Delta$, processors look to see if they have received Q' which is a stage 1 QC for some block b for view v. If so, they:

- Set `lock` := Q' (reset the lock).
- Send Q' and a stage 2 vote for b to all processors.

How about Liveness? We have seen above that it will be easy to argue that Tendermint satisfies Consistency. In fact, proving Liveness is also straightforward. The main task in establishing Liveness is to show that every view v that has a correct leader and begins after GST+Δ finalises a new block. This follows just because, when any correct processor has set its local value `lock` := Q' during a previous view v', it has sent Q' to all processors. This must have happened at least Δ time-slots before the start of view v. The leader will therefore have received this stage 1 QC, and will select a parent corresponding to a view greater than or equal to any correct processor's local value `lock`.view. All correct processors will therefore produce stage 1 votes, and then stage 2 votes for the leader's proposal during view v, and the proposal will therefore be finalised.

That's really all there is to the Tendermint protocol. However, the discussion above was somewhat informal. In the next section, we give a formal specification.

9.2.3 The formal specification

Recall the conventions described in Section 3.2 concerning the meaning of the instruction to 'disseminate'. For the sake of simplicity, we assume (without explicit mention in the pseudocode) that correct processors automatically send new transactions to all others upon first receiving them - we will revisit this assumption in Section 9.5. We write $\emptyset$ to denote the empty set or empty sequence, and we let $\perp$ be some default distinguished value. The pseudocode uses a number of message types, local variables, functions and procedures, detailed below.

The function `lead`(v). The value `lead`(v) specifies the *leader* for view v. To be concrete, we set `lead`$(v) := p_i$, where $i = v \bmod n$.

Blocks. A *block* b is a message specified by four values:

- b.view: this is a value in $\mathbb{N}_{\geq 0}$ that specifies the view corresponding to b;
- b.Tr: a sequence of transactions in $\mathcal{T}$;[2]
- b.par: either $\perp$, or a hash value specifying the parent of b;
- b.QC: a stage 1 QC for b.par if b.par $\neq \perp$.

A correct processor only regards a message as a block if it is of the form above. If b.view $= v$, we also refer to b as a 'view v block'. The *genesis block* is denoted b_g and satisfies b_g.view $= 0$, b_g.Tr $= \emptyset$, b_g.par $= \perp$, b_g.QC $= \perp$. We regard b_g as received by all correct processors at time-slot 0. Blocks are ordered by b.view and then by least hash.

The sequence of transactions specified by the ancestors of a block. Each block b specifies an extended sequence of transactions, denoted b.Tr*, as follows: we concatenate the values b'.Tr for all ancestors b' of b, removing any duplicate transactions.

[2]Recall that $\mathcal{T}$ is the set of possible transactions.

Votes. For $d \in \{1, 2\}$, a stage d *vote* is a message of the form $V = (\text{vote}, v, d, \tau)$ signed by some processor in Π, where $v \in \mathbb{N}_{\geq 0}$ specifies the view corresponding to V and τ is a finite binary string. If b is a block, $v = b.\text{view}$, and $\tau = H(b)$, we say V is a *stage d vote for b*.

Stage 1 and 2 QCs. For $d \in \{1, 2\}$, a stage d QC is a set Q of $n - f$ votes, each of the form $V = (\text{vote}, v, d, \tau)$ for the same values of v and τ, and each signed by a different processor in Π. We set $Q.\text{view} = v$, $Q.\text{block} = \tau$. Stage 1 QCs are ordered by view and then by least hash. We also say Q is a stage d QC for τ, and if b is a block, $v = b.\text{view}$, and $\tau = H(b)$, we say Q is a stage d QC for b. We stipulate that $\emptyset$ is a stage 1 QC and a stage 2 QC for b_g, and set $\emptyset.\text{view} = 0$.

The lock. Each processor maintains a local variable `lock`, initially set to $\texttt{lock} = \emptyset$.

Valid proposals. At time-slot t, p_i regards a block b as a *valid proposal for view v* if all of the following conditions are satisfied: (i) $b.\text{view} = v$; (ii) $b.\text{par} \neq \perp$; (iii) $Q := b.\text{QC}$ is a stage 1 QC for $b.\text{par}$, and; (iv) $Q.\text{view} \geq \texttt{lock}.\text{view}$.

The procedure MakeProposal. This procedure is executed by the leader p of view v to determine a new block. To execute the procedure, p:

1. Lets Q be the greatest stage 1 QC it has received.
2. Forms a sequence of distinct transactions T, containing all transactions received but not finalised by p;
3. Sets $b.\text{view} := v$, $b.\text{Tr} := T$, $b.\text{par} := Q.\text{block}$ and $b.\text{QC} = Q$.
4. Disseminates b.

Finalising blocks and transactions. Processor p regards b as finalised upon receiving a stage 2 QC for b. However, the transactions in $b.\text{Tr}$ cannot be finalised until p has received all ancestors of b. Formally, we define $\mathcal{F}$ (as required for SMR in Section 7.3) as follows. For any set of messages M, let b be the greatest block such that $M \cup \{b_g\}$ contains: (i) a stage 2 QC for b, and (ii) all ancestors of b and stage 1 QCs for all ancestors of b. Set $\mathcal{F}(M) := b.\text{Tr}^*$.

The instructions are shown in Figure 9.1.

While the protocol described above is quite efficient, there are a number of ways it can be optimised. We will discuss some of these in Section 9.5.

Having specified the protocol, we next verify that it satisfies Consistency and Liveness. The proofs are essentially just those outlined in Section 9.2.2.

9.2.4 Verifying Consistency and Liveness

Throughout this section, we consider the partially synchronous model with synchronised clocks, signatures and Byzantine faults, and we suppose that $f < n/3$. First, we verify that the protocol satisfies Consistency.

Lemma 9.1 (Consistency)**.** *The Tendermint protocol for synchronised clocks satisfies Consistency.*

Tendermint for synchronised clocks: the instructions for p_i.
At time-slot $3v\Delta$ for $v \in \mathbb{N}_{>0}$:
  If $p_i = \texttt{lead}(v)$:
    MakeProposal; ▷ Propose a new block if leader

At time-slot $3v\Delta + \Delta$ for $v \in \mathbb{N}_{>0}$:
  If p_i has received exactly one valid proposal b for view v from $\texttt{lead}(v)$:
    Disseminate b and a signed stage 1 vote for b; ▷ stage 1 vote

At time-slot $3v\Delta + 2\Delta$ for $v \in \mathbb{N}_{>0}$:
  If p_i has received exactly one valid proposal b for view v and Q which is a stage 1 QC for b:
    Set $\texttt{lock} := Q$;
    Disseminate Q and a signed stage 2 vote for b; ▷ stage 2 vote

FIGURE 9.1: The pseudocode for Tendermint with synchronised clocks.

Proof. For $d \in \{1, 2\}$, let us say that block b 'receives a stage d QC' if $b = b_g$ or there exist at least $n - f$ processors that each send a stage d vote for b to at least one processor. Any block b that receives a stage 2 QC must also receive a stage 1 QC, since no correct processor sends a stage 2 vote for b before receiving a stage 1 QC for b.

To argue that the protocol satisfies Consistency, it suffices to show that no two incompatible blocks can receive stage 2 QCs. Towards a contradiction, suppose that there exists a least v such that:

- Some b with $b.\text{view} = v$ receives stage 1 and 2 QCs, Q_1 and Q_2 say.
- For some least $v' \geq v$, there exists b' such that b' is incompatible with b, with $b'.\text{view} = v'$ and $b'.\text{QC} = Q_0$ (say), and the block b' receives a stage 1 QC, Q_3 say.

If $v = v'$ then, by the quorum intersection argument of Section 9.1.2, some correct processor must have sent votes in both Q_1 and Q_3. This gives a contradiction since correct processors send at most one stage 1 vote in each view.

So, suppose $v' > v$. Then, by the same quorum intersection argument, some correct processor p must have sent votes in both Q_2 and Q_3. This gives the required contradiction, since p must set its local value `lock` to be a stage 1 QC for b while in view v. However, our choice of (v, v') and the fact that b' is incompatible with b means that $Q_0.\text{view} < v$, so that p would not regard the proposal b' as valid while in view v' and would not produce a vote in Q_3. □

To prove Liveness, we first prove the following lemma.

Lemma 9.2 (Correct leaders finalise new blocks)**.** *If view $v > 0$ begins at least Δ time-slots after GST (i.e., $3v\Delta \geq GST + \Delta$) and $p = \texttt{lead}(v)$ is correct, then p proposes a view v block b and all correct processors receive a stage 2 QC for b by the end of view v.*

Proof. The proof proceeds just as sketched in Section 9.2.2. For each $p_i \in \Pi$, let $\texttt{lock}_i$ denote the local variable `lock` as defined for p_i. Suppose the conditions in the statement

of the lemma hold and let $t = 3v\Delta$. Suppose p_i is correct and let $Q' = \texttt{lock}_i$ as defined at t. Note that $\texttt{lock}_i$ is only redefined when p_i sends a stage 2 vote in some view, meaning that p_i most recently set this value at $t' \leq t - \Delta$ (possibly $t' = 0$). Upon doing so, p_i sends Q' to all processors (or, if $t' = 0$, then all processors have already received Q'). Since $t - \Delta \geq \text{GST}$, $p = \texttt{lead}(v)$ receives this stage 1 QC by t. From the definition of the procedure MakeProposal, it follows that p proposes a view v block b such that, for $Q = b.\text{QC}$, $Q.\text{view} \geq Q'.\text{view}$. Since our choice of p_i was arbitrary amongst correct processors, and because no correct processor resets its lock in the interval $(t - \Delta, t + \Delta]$, it follows that all correct processors will regard b as a valid proposal for view v at $t + \Delta$. All correct processors will therefore send a stage 1 vote for b to all processors at this time-slot. All correct processors will then receive a stage 1 QC for b by $t + 2\Delta$, and will send a stage 2 vote for b to all processors at this time-slot. All correct processors then receive a stage 2 QC for b by $t + 3\Delta$. □

Next, we prove that correct processors eventually receive all ancestors of any block finalised by a correct processor.

Lemma 9.3 (Processors receive necessary blocks)**.** *Suppose p is correct and finalises b. Then all correct processors eventually receive all ancestors of b.*

Proof. The *height* h of a block is its number of ancestors. First, we prove by induction on height that if b receives a stage 1 QC, then all ancestors of b receive stage 1 QCs. The statement is vacuously true for the genesis block (height 1). So, suppose the claim holds for $h \geq 1$. No correct processor sends a stage 1 vote for any block b of height $h+1$ before seeing a stage 1 QC for its parent, i.e., this condition is required for b to be considered a valid proposal for the view. It therefore follows from the induction hypothesis that, if b receives a stage 1 QC, all ancestors of b receive stage 1 QCs.

Now suppose that p is correct and finalises b (of any height). If $b = b_{\text{g}}$ then the claim of the lemma holds, since all correct processors receive b_{g} at time-slot 0. If $b \neq b_{\text{g}}$, then we have established that all ancestors of b receive stage 1 QCs. All correct processors that send stage 1 votes for any ancestor also send that ancestor to all processors. So, all correct processors receive all ancestors of b, as required. □

Finally, we establish Liveness.

Lemma 9.4 (Liveness)**.** *The Tendermint protocol for synchronised clocks satisfies Liveness.*

Proof. Suppose correct p_i receives the transaction tr at t_0. We specified in Section 9.2.3 that p_i sends tr to all processors upon receiving it. So, all correct processors receive tr by $t_1 = \max\{t_0, \text{GST}\} + \Delta$. Let v be a view that begins at $t_2 \geq t_1$, and with correct leader p. By Lemma 9.2, p proposes a block b that is finalised by all correct processors by the end of view v. By Consistency, and by the definition of the procedure MakeProposal, $\text{tr} \in b'.\text{Tr}$ for some ancestor b' of b, i.e., $\text{tr} \in b.\text{Tr}^*$. By Lemma 9.3, all correct processors receive all ancestors of b. So, tr is finalised by all correct processors. □

Hopefully, the Tendermint protocol we presented in this section seems quite simple. Next, we present a 'pipelined' version of the protocol, which has an even simpler protocol specification.

9.3 Pipelined Tendermint

The basic idea behind Pipelined Tendermint is that we *can* actually use a single round of voting in each view, so long as finalisation for b requires receiving a stage 1 QC and then a child b' of b also receiving a stage 1 QC in the next view. In this case, the round of voting on b' can be seen as playing the same role as the second round of voting on b in the previous section. Proceeding this way, the proof of Consistency will be almost unchanged. The only major change to the proof of Liveness is that we now require correct leaders in two consecutive views after GST to guarantee finalisation of a new block. To accommodate this, we could have each leader be responsible for two consecutive views. However, the assumption that $f < n/3$ already guarantees consecutive views with correct leaders.

All definitions, besides $\mathcal{F}$ and when processor finalise blocks, remain the same as in Section 9.2.3. When we refer to a 'QC' in what follows, we mean a stage 1 QC. When we refer to a 'vote', we mean a 'stage 1 vote'.

Processors now finalise b upon receiving a QC for b and a QC for a child b' of b with $b'.\text{view} = b.\text{view} + 1$. For any set of messages M, let b be the greatest block b such that either $b = b_g$ or $M \cup \{b_g\}$ contains: (i) a QC for b; (ii) a QC for a child b' of b with $b'.\text{view} = b.\text{view} + 1$, and; (iii) all ancestors of b and QCs for all ancestors of b. Set $\mathcal{F}(M) := b.\text{Tr}^*$.

The protocol instructions are shown in Figure 9.2.

Pipelined Tendermint: the instructions for p_i.
At time-slot $2v\Delta$ for $v \in \mathbb{N}_{>0}$:
  If $p_i = \texttt{lead}(v)$:
    MakeProposal; ▷ propose a new block if leader

At time-slot $2v\Delta + \Delta$ for $v \in \mathbb{N}_{>0}$:
  If p_i has received exactly one valid proposal b for view v from $\texttt{lead}(v)$:
    Set $\texttt{lock} := b.\text{QC}$ and send this value to all processors; ▷ set lock
    Disseminate b and a signed vote for b; ▷ vote

FIGURE 9.2: The pseudocode for Pipelined Tendermint.

Next, we prove consistency and liveness. The proofs are simple adaptations of those from Section 9.2.4.

9.3.1 Verifying Pipelined Tendermint

As in Section 9.2.4, we consider the partially synchronous model with synchronised clocks, signatures and Byzantine faults, and we suppose $f < n/3$. First, we verify that the protocol satisfies Consistency.

Lemma 9.5 (Consistency)**.** *Pipelined Tendermint satisfies Consistency.*

Proof. Let us say that block b 'receives a QC' if $b = b_g$ or there exist at least $n - f$ processors that each send a vote for b to at least one processor. Towards a contradiction, suppose that there exists a least v such that:

- Some b_1 with $b_1.\text{view} = v$ receives a QC, Q_1 say, and some child b_2 of b_1 with $b_2.\text{view} = b_1.\text{view} + 1$ also receives a QC, Q_2 say.
- For some least $v' \geq v$, there exists b_3 such that b_3 is incompatible with b_1, $b_3.\text{view} = v'$ and b_3 receives a QC, Q_3 say. Let $Q_0 = b_3.\text{QC}$.

If $v = v'$ then, by the quorum intersection argument of Section 9.1.2, some correct processor must have sent votes in both Q_1 and Q_3. This gives a contradiction since correct processors send at most one vote in each view. We reach the same contradiction if $v + 1 = v'$, by considering the processors that send votes in both Q_2 and Q_3.

So, suppose $v' > v + 1$. Then, by the same quorum intersection argument, some correct processor p must have sent votes in both Q_2 and Q_3. This gives the required contradiction, since p must set its local value `lock` to be a QC for b_1 while in view $v + 1$. However, our choice of (v, v') and the fact that b_3 is incompatible with b_1 means that $Q_0.\text{view} < v$, so that p would not regard the proposal b_3 as valid while in view v' and would not produce a vote in Q_3. □

To prove Liveness, we need simple adaptions of Lemmas 9.2-9.4. The proofs of the first two of these lemmas only require trivial modifications and are left as an exercise (Exercise 9.1).

Lemma 9.6 (Liveness)**.** *Pipelined Tendermint satisfies Liveness.*

Proof. The proof is almost the same as the proof of Lemma 9.4. Suppose correct p_i receives the transaction tr at t_0. We specified in Section 9.2.3 that p_i sends tr to all processors upon receiving it. So, all correct processors receive tr by $t_1 = \max\{t_0, \text{GST}\} + \Delta$. Let v be a view that begins at $t_2 \geq t_1$, and such that the leaders of views v and $v + 1$ are both correct: such a view exists because $f < n/3$. Let these leaders be p and p', respectively. Then, by almost exactly the same proof as for Lemma 9.2, p proposes a block b, and all correct processors receive a QC for b by the end of view v. Similarly, the next leader p' then proposes a child b' of b, and all correct processors receive a QC for b' by the end of view $v + 1$. So, b is finalised by all correct processors by the end of view $v + 1$. By Consistency, and by the definition of the procedure MakeProposal, $\text{tr} \in b''.\text{Tr}$ for some ancestor b'' of b, i.e., $\text{tr} \in b.\text{Tr}^*$. By almost exactly the same proof as for Lemma 9.3, all correct processors receive all ancestors of any block that receives a QC. So, tr is finalised by all correct processors. □

So far, in this section and in Section 9.2, we have considered protocols for the partially synchronous model with synchronised clocks. In the next section, we consider how to adapt these protocols for the version without synchronised clocks.

9.4 Tendermint without synchronised clocks

In this section, we'll consider how to adapt the protocol of Section 9.2 to the partially synchronous model without synchronised clocks. Similar modifications can also be applied to Pipelined Tendermint.

9.4.1 The intuition

When we move to the partially synchronous model without synchronised clocks, some mechanism is required to *synchronise* processors on the same view. Roughly, the standard approach works as follows:

- Processors no longer move into views at pre-specified times. A processor now moves into view v from a lower view upon receiving an appropriate *certificate* that gives them permission to do so. These certificates are formed and shared as specified in the bullet points below.
- Upon entering view v, each processor sets a local *timer* to expire in some fixed amount of time. Once the timer expires they send a 'time-out' message for view v to all other processors.
- Processors enter view $v+1$ if they are presently in a lower view and if they receive Q which is either a stage 2 QC corresponding to view v, or else a set of $n-f$ time-out messages for view v, each signed by a different processor. We call a collection of messages of the latter sort a *time-out certificate* (TC) for view v. Upon receiving Q and entering view $v+1$, the processor sends Q to all other processors.

Why does this work? First, note that none of these changes impact Consistency. Correct processors still only send at most one stage 1 vote and at most one stage 2 vote in each view, and still implement the same locking mechanism. This means the proof of Consistency goes through unchanged.

To prove Liveness, there are two new considerations:

1. For each $v \geq 1$, we now need to prove that there is some first time-slot, t_v say, at which a correct processor enters view v. Also, $t_{v'} > t_v$ if $v' > v$.
2. We need to check that, after GST, correct leaders still produce new finalised blocks.

Proving (1) is straightforward. It follows immediately that no correct processor can enter view $v+1$ before any correct processor has entered view v, because entering view $v+1$ requires either a stage 2 QC or a TC for view v, to which some correct processors must contribute. To show that each t_v is defined, the crucial point is that, upon entering any view, a correct processor sends the relevant certificate to all others. So, if any correct processor enters infinitely many views, they all do. If there were some greatest view v entered by any correct processor p, then all correct processors eventually enter view v. Then all correct processors send time-out messages for view v, and so receive a TC for view v.

The proof for (2) is close to that for Lemma 9.2, with the following caveat. Suppose p is the first correct processor to enter view v and does so at $t_v \geq \text{GST}$. Since p sends the certificate allowing it to enter the view to all others, all correct processors enter view v by $t_v + \Delta$. However, the fact that they now only enter the view within Δ time-slots of each other (rather than exactly together, as in Section 9.2) means that leaders now have to wait 2Δ time-slots after entering the view before producing their block. This is required to ensure they have sufficient time to receive the locks of correct processors: if

the leader is correct and is the first to enter the view, then other correct processors will enter by $t_v + \Delta$ and the leader will receive all necessary locks by $t_v + 2\Delta$.

In the next section, we describe the formal specification.

9.4.2 The formal specification

The formal definitions used to specify the protocol remain unchanged from Section 9.2.3, but we also make the definitions below.

The local variable `v`: this is a local variable, which p uses to record the present view. Initially, $\texttt{v} = 1$.

The local variables `1voted` and `2voted`: these record whether p_i has already sent stage 1 and stage 2 votes in the present view, and are initially set to 0.

Time-out messages and TCs: a time-out message for view v is a message of the form $(\text{time-out}, v)$, signed by some processor in Π. A set of $n - f$ messages of this form, each signed by a different processor, is called a time-out certificate (TC) for view v.

The timer: each processor p has a local timer, denoted `Timer`, which it can set to 0 at any time, and which then automatically increments at each time-slot at which p is active. This means that the timer increments in 'real time' after GST. Initially, $\texttt{Timer} = 0$.

The protocol instructions are shown in Figure 9.3. We'll refer to the version of Tendermint for partial synchrony (without synchronised clocks) simply as 'Tendermint'.

Why 5Δ for time-out? Here, we give an informal analysis: a more formal analysis appears in the proof of Lemma 9.8 in the next section. As already explained, leaders must wait 2Δ before forming their proposal. This means that if $\texttt{lead}(v)$ is correct, and if the first correct processor p to enter view v does so at $t \geq \text{GST}$, then $\texttt{lead}(v)$ will enter the view by $t + \Delta$. Then $\texttt{lead}(v)$ will send out a block b by $t + 3\Delta$. All correct processors will send stage 1 votes for b by $t + 4\Delta$, and will send stage 2 votes by $t + 5\Delta$. It is therefore safe to send time-out messages at $t + 5\Delta$, i.e., doing so cannot cause any correct processor to receive a TC for view v before sending their stage 2 vote.

In the next section, we show that Tendermint satisfies (Consistency and) Liveness.

9.4.3 Verifying Consistency and Liveness

Throughout this section, we consider the partially synchronous model (without synchronised clocks), signatures and Byzantine faults, and we suppose that $f < n/3$. The proof of Consistency is exactly the same as for Lemma 9.1, so we focus on Liveness.

We begin by establishing that correct processors progress through the views.

Lemma 9.7. *For each $v \geq 1$, there is some first time-slot, t_v say, at which a correct processor enters view v. Also, $t_{v'} > t_v$ if $v' > v$.*

Proof. The proof given in Section 9.4.1 works exactly as stated. □

Tendermint: the instructions for p_i.
At time-slot 0:
Set $\mathtt{v} := 1$, $1\mathtt{voted} := 0$, $2\mathtt{voted} := 0$, $\mathtt{Timer} := 0$, $\mathtt{lock} = \emptyset$; ▷ initialise

At every time-slot t:

If $p_i = \mathtt{lead}(\mathtt{v})$ and $\mathtt{Timer} = 2\Delta$:
MakeProposal; ▷ Propose a new block if leader

If p_i has received exactly one valid proposal b for view $\mathtt{v}$ from $\mathtt{lead}(\mathtt{v})$ and $1\mathtt{voted} = 0$:
Disseminate b and a signed stage 1 vote for b; ▷ stage 1 vote
Set $1\mathtt{voted} := 1$;

If p_i has received exactly one valid proposal b for view $\mathtt{v}$ and Q which is a stage 1 QC for b and if $1\mathtt{voted} = 1$ and $2\mathtt{voted} = 0$:
Set $\mathtt{lock} := Q$;
Disseminate Q and a signed stage 2 vote for b; ▷ stage 2 vote
Set $2\mathtt{voted} := 1$;

If p_i has received Q which is a stage 2 QC with $Q.\text{view} = v' \geq \mathtt{v}$ or which is a TC for view $v' \geq \mathtt{v}$:
Disseminate Q;
Set $\mathtt{v} := v' + 1$, $1\mathtt{voted} := 0$, $2\mathtt{voted} := 0$, $\mathtt{Timer} := 0$; ▷ enter higher view

If $\mathtt{Timer} = 5\Delta$:
Disseminate a time-out message for view $\mathtt{v}$; ▷ send time-out

FIGURE 9.3: The pseudocode for Tendermint.

Next, we establish the equivalent of Lemma 9.2.

Lemma 9.8 (Correct leaders finalise new blocks)**.** *For each $v \geq 1$, let t_v be as defined in the statement of Lemma 9.7. If $t_v \geq GST$ and $p = \mathtt{lead}(v)$ is correct, then p proposes a view v block b and all correct processors receive a stage 2 QC for b.*

Proof. The proof is very similar to the proof of Lemma 9.2, but we must be careful about the precise timing. For each $p_i \in \Pi$, let $\mathtt{lock}_i$ denote the local variable $\mathtt{lock}$ as defined for p_i.

Suppose the conditions in the statement of the lemma hold. Towards a contradiction, suppose it is not the case that p proposes a view v block b and all correct processors receive a stage 2 QC for b by $t_v + 6\Delta$. If any correct processor received such a 2 QC by $t_v + 5\Delta$ it would send it to all other processors, which gives an immediate contradiction. No correct processor sends a time-out message for view v before $t_v + 5\Delta$, so no processor can receive a TC for view v before this time. Since the first correct processor to enter view v sends the corresponding certificate (a stage 2 QC or TC for view $v - 1$) to all processors at t_v, it follows that all correct processors are in view v for the entirety of the interval $[t_v + \Delta, t_v + 5\Delta]$. This holds because no correct processor can leave view v without receiving either a stage 2 QC or TC for a view $\geq v$, neither of which can happen before $t_v + 5\Delta$.

Suppose p_i is correct. Note that, before entering view v, p_i most recently set $\mathtt{lock}_i$ to some value Q' at some $t' \leq t_v + \Delta$ (possibly $t' = 0$). Upon doing so, p_i sends Q' to all

processors (or, if $t' = 0$, then all processors have already received Q'). It follows that p receives Q' by $t_v + 2\Delta$, before it produces and sends a block. Note that p_i does not redefine $\texttt{lock}_i$ while in view v before sending a stage 1 vote: this is because $\texttt{lock}_i$ is only set when sending a stage 2 vote, which requires having already sent a stage 1 vote. From the definition of the procedure MakeProposal, it follows that p proposes a view v block b by $t_v + 3\Delta$ such that, for $Q = b.\text{QC}$, $Q.\text{view} \geq Q'.\text{view}$. Since our choice of p_i was arbitrary amongst correct processors, all correct processors send stage 1 votes for b by $t_v + 4\Delta$. All correct processors will then receive a stage 1 QC for b by $t_v + 5\Delta$, and will send a stage 2 vote for b to all processors by this time-slot. All correct processors then receive a stage 2 QC for b by $t_v + 6\Delta$, giving the required contradiction. □

Given Lemmas 9.7 and 9.8, the statements of Lemmas 9.3 and 9.4 can then be shown to hold for Tendermint, with essentially identical proofs to those from Section 9.2.4. This establishes Liveness for Tendermint.

In the next section, we consider various optimisations of Tendermint.

9.5 Tendermint: further analysis

9.5.1 A design principle

Of course, a big advantage of Tendermint over the 'proof-of-principle' protocol described in Sections 7.2 and 7.4 is that Tendermint solves SMR for the partially synchronous model. However, even in synchrony, Tendermint is much more efficient when leaders act correctly: while f faulty leaders in a row can cause a delay $O(f\Delta)$ between views that finalise new blocks, each correct leader finalises a new block in a small constant number of rounds of communication. The protocol of Sections 7.2 and 7.4 was impractical for large f because each view required $f + 1$ rounds of communication.

The considerations above speak to an important design principle for SMR protocols. While we are certainly interested in performance when a large number of processors are faulty, and while we certainly want to maintain Consistency when message delivery is unreliable, experience with real-world systems shows that *most of the time* message delivery is reliable and most processors act correctly. While we want strong Consistency guarantees when conditions are not favourable, this means that, when it comes to efficiency considerations, we are most interested in the 'common case' that message delivery is reliable and at most a few processors are faulty. Overall, the point is that Tendermint has two strong advantages over the protocol of Sections 7.2 and 7.4:

1. It solves SMR in partial synchrony;
2. A small constant number of rounds of communication are required to finalise each new block in the 'common case' that message delivery is reliable and leaders act correctly.

In Section 9.3, we already considered how 'pipelining' can be applied to Tendermint. In the remainder of this chapter, we give a number of forward pointers to other ways in which Tendermint and other similar protocols can be optimised. We'll formalise these optimisations in later chapters of the book.

9.5.2 Quick block proposals in the good case

The version of Tendermint described in Section 9.4.2 requires the leader of view $v+1$ to wait 2Δ before proposing. However, it is straightforward to see that this wait of 2Δ is not necessary in the case that the leader receives a stage 1 QC for view v: in this case the leader has already received a stage 1 QC for the greatest possible view $< v+1$, and so further waiting serves no purpose. This optimisation means that the wait of 2Δ will not be necessary in the common case that messages are delivered within Δ time-slots and the leaders of views v and $v+1$ are correct. The terminology that is often used to express this is that the modified protocol satisfies a form of *optimistic responsiveness*: we'll define two versions of optimistic responsiveness in Chapter 11.

9.5.3 Block echoing

In all of the protocols described in this chapter, we required processors to send the block b to all others whenever they send a stage 1 vote for b. This 'echoing' of received blocks was used to ensure that all correct processors receive any block finalised by a correct processor (required for SMR). In real-world scenarios, however, this process of echoing blocks may be expensive: blocks are often much larger objects than other protocol messages, such as votes. If *all* processors send b to *all other* processors, this will consume valuable network bandwidth.

A common approach is therefore *not* to require processors to echo received blocks, and to exploit the fact that receiving a stage 1 QC suffices to ensure *data-availability*: if b receives a stage 1 QC then at least $n-f$ processors (including many correct ones) must have received and voted for it. Correct processors can therefore use a (potentially rate-limited) "pull" mechanism to retrieve any missing finalised blocks from peers who possess them. This approach involves a trade-off: while it saves on communication costs when processors act correctly, extra time may be required to retrieve necessary blocks in the case that Byzantine leaders initially withhold those blocks from some processors. These ideas will be explored in further detail in Chapter 13, where we define the notion of *Extractable SMR*.

An alternative approach is to have leaders broadcast blocks using *erasure coding* techniques. This method will be explained in Chapter 17.

9.5.4 The Mempool

Another simplification we made in this chapter was to assume that processors automatically forward all newly received transactions to all other processors. This simplification allowed us to focus on the consensus mechanism without worrying about transaction availability. Similar to the considerations in Section 9.5.3, however, in real-world scenarios such all-to-all transaction echoing may be expensive. This means that different methods are often applied in practice. A standard approach is to use a separate 'mempool' protocol for transaction dissemination. For example, each processor might forward each newly received transaction to some small number of other processors, who then forward on the transaction themselves, and so on. Transactions are thus disseminated via a 'gossip network'.

In fact, the standard approach when describing and analysing SMR protocols is to black box the underlying mempool protocol. The SMR protocol is then required to function efficiently given that transaction dissemination is handled by the mempool. We will consider these issues in more detail in the next chapter, where we will also formally define the task of a 'mempool'.

9.5.5 Threshold signatures

In the versions of Tendermint described in this chapter, we required processors to send their lock (a stage 1 QC) to all other processors whenever they reset this value. In the protocol of Section 9.4.2, we also required processors to forward a stage 2 QC or a TC to all others upon leaving each view. If n is large, these certificates will be large objects. An approach that can sometimes increase efficiency in this case is to make use of a cryptographic technique called a *threshold signature scheme*. Roughly, the functionality that such a scheme provides is to give a mechanism for condensing any stage 1 QC or stage 2 QC (or TC) into a message of constant length. Rather than sending the entire QC (a set of $n - f$ messages), processors can now send the shorter message, saving on communication costs. In Chapter 12, we will see how to formally model the use of threshold signatures within our framework.

9.5.6 Random leaders

So far, we have considered protocols in which leaders rotate deterministically. If processors have access to a common source of randomness, then an alternative is to use that randomness to specify leaders. For a static adversary, and for any t after GST, the *expected* time after t until (the first and then all) correct processors enter a view with correct leader then becomes constant-bounded, which means that, after GST, the expected time to finalise new transactions becomes constant-bounded. Chapter 12 describes how we can model common sources of randomness within our formal framework.

Although we have talked informally about 'efficiency' for SMR protocols in this section, we do not yet have any formal metrics by which to compare SMR protocols. In the next chapter, we define notions of *message complexity*, *communication complexity* and *latency* for SMR protocols.

9.6 Exercises

Exercise 9.1. *Prove that Pipelined Tendermint, as described in Section 9.3, satisfies Liveness.*

Exercise 9.2. *Suppose block sizes are constant-bounded. As described in Section 9.5.3, drop the requirement for block echoing. As described in Section 9.5.5, suppose also that threshold signatures are available. Modify Tendermint with synchronised clocks, so that the total number of bits sent by correct processors in each view is $O(n)$. Hint: replace all-to-all message sending with a process whereby messages are relayed via the leader.*

Exercise 9.3. *Recall the notions of consistency resilience (ρ_C) and liveness resilience (ρ_L) from Exercise 8.3. Building on your answer to Exercise 8.3, and by considering versions of Tendermint with different quorum sizes, determine which ρ_C and ρ_L are possible for SMR protocols in the partially synchronous model.*

Chapter 10

SMR metrics

For BA and BB, we have already considered some metrics that allow one to formally calibrate protocol efficiency. One natural metric, considered in Chapter 5, is the *time* before correct processors output. Another, considered in the same chapter, is the number of *messages* sent by correct processors. A refinement of the latter metric, already briefly discussed in Section 5.3, considers the number of *bits* sent by correct processors. However, the fact that SMR is a 'multi-shot' protocol, i.e., there is no time-slot at which correct processors terminate and give some final output, means that there are a number of ways in which we can form similar notions for SMR.

In Section 10.1 of this chapter, we begin by defining some natural complexity notions for SMR protocols, which are analogues of the metrics above. Since SMR protocols are often defined and analysed assuming that some (separate and black boxed) 'mempool' protocol is responsible for transaction propagation, in Section 10.2 we then formally define the 'Mempool' task. In Section 10.3, we describe how to modify the notions of Section 10.1 so that they correctly apply to SMR protocols using a black boxed mempool. In Section 10.4, we analyse how Tendermint fares according to the metrics we have defined.

10.1 Complexity metrics for SMR

We begin by defining *latency*. Then, in Section 10.1.2, we consider message and communication complexity.

10.1.1 Latency for SMR protocols

For any transaction tr received by a correct processor in an execution, the latency for tr is the time between when tr is first received by any correct processor and when tr is first finalised by all correct processors.

The latency of an SMR protocol is the supremum, over all executions[1] and all transactions first received by a correct processor after GST, of the latency for that transaction. (In the synchronous model, we take GST = 0.)

[1] Of course, when analysing the latency of an SMR protocol, we consider only executions consistent with the stated assumptions (e.g., at most f Byzantine processors).

We note that the metric above considers the *worst case*, in the sense that we consider the largest possible latency for any transaction. To define the *good-case* latency, the definition is the same except that we restrict attention to executions in which all processors are correct. This terminology is fairly standard, but can lead to confusion since the latency considered is still the largest possible latency for a transaction first received by a correct processor after GST in the executions considered: it's the 'good case' latency only in the sense that processors are now assumed to be correct.

It is also worth noting that, while the definition of good-case latency above is formed so as not to be protocol specific, for many 'leader-based' protocols it accurately captures latency in the more general case that leaders act correctly.

Next, we consider message and communication complexity, which give rise to some more nuanced considerations.

10.1.2 Message and communication complexity for SMR protocols

Message complexity. The *message complexity* of an SMR protocol is the maximum number of messages sent by correct processors (combined) in any interval of the form $[t_1, t_2]$, such that $t_1 \geq \text{GST}+\Delta$, some transaction tr is first received by a correct processor at t_1, and t_2 is the first time-slot at which tr is finalised by all correct processors. Here, the maximum is taken over all executions, and the message complexity is considered to be infinite if no such maximum exists.

Why GST+Δ? It may initially seem more natural to consider GST, rather than GST + Δ, in the definition above. The latter is used because between GST and GST+Δ, correct processors may receive (and respond to) an unbounded backlog of messages from before GST. We are generally most interested in the number of messages sent once things have stabilised after GST.

Communication complexity. The communication complexity uses the same definition but counts bits rather than messages. We assume transactions have fixed bit-length (the precise value doesn't affect asymptotic complexity).

Complications in applying the definition. Applying the definition above often requires additional assumptions. If the environment sends an unbounded number of transactions whilst tr awaits finalisation, communication complexity (even for many reasonable protocols) may become unbounded. Meaningful analysis therefore typically requires bounding block sizes and the transaction arrival rate.

Single transaction complexity. To avoid these complications, we define *single transaction communication complexity*, for which we consider executions where the environment sends only one transaction tr (to at least one correct processor). The complexity is the maximum number of bits sent by correct processors between tr's first receipt by any correct processor and its finalisation by all correct processors. *Single transaction message complexity* counts messages rather than bits.

This definition counts the number of bits/messages sent by the consensus protocol to achieve finalisation, without the need to count the cost of dealing with large sets of transactions. Another approach is to distinguish between 'consensus' messages and others. The difficulty with the latter method is that protocols can convey information

by repeatedly sending transactions in specific orders, which makes proving lower bounds complicated.

Since single-transaction message/communication complexity lower bounds the general message/communication complexity, lower bounds proven for the former immediately apply to the latter.

Amortised complexity. While the definitions above capture important aspects of a protocol's efficiency, alternative metrics are sometimes required to properly reflect performance. For example, in Chapter 19 we will consider 'DAG-based' protocols, in which many processors propose blocks simultaneously. Accordingly, these protocols have large single transaction message complexity. However, each new consensus decision also finalises a large number of new transactions. In this case, the cost *per transaction*, also called the amortised complexity, may better reflect performance. We define the *amortised communication complexity* to be the supremum over all executions of:

$$\limsup_{t\to\infty} \frac{\text{total bits sent by correct processors by time } \ t}{\text{number of transactions finalised by all correct processors by time } t}.$$

Since GST occurs at some finite time in every infinite execution, the definition above does not restrict attention to costs after GST. When considering certain finite executions, it may be useful to make such a restriction.

When is amortised complexity lower? Since the environment may send only a single transaction in any execution, single transaction communication complexity lower bounds amortised communication complexity. However, the latter complexity may be lower than the former when we restrict attention to executions in which the environment sends many transactions, e.g., when many transactions can be included in each block, reducing the overall per transaction cost. These ideas will be discussed further in Section 10.1.4.

10.1.3 Lower bounds

In Chapter 5, we showed that deterministic protocols for BA and BB require $f+1$ rounds of communication and also require correct processors to send a number of messages that is quadratic in f. Exercises 10.1 and 10.2 walk you through how to modify these proofs to give analogous results for SMR: any deterministic SMR protocol has latency at least $(f+1)\Delta$ and single transaction message complexity at least quadratic in f. The proofs are simple modifications of those in Chapter 5. However, we will see later in this chapter that a modified proof is required to lower bound message complexity when protocols work modulo a Mempool protocol.

10.1.4 How to weigh these metrics?

Limitations of existing models. The lock-step, synchronous, partially synchronous and asynchronous models are the standard models used in Distributed Computing. Common to all these models is the fact that message sizes are unbounded: for example, in the synchronous setting, a message of *any size* sent at time t is guaranteed to be delivered by $t+\Delta$. Of course, part of the motivation for this assumption is to simplify analysis,

and consideration of message sizes may not be critical when establishing fundamental properties of protocols such as Consistency and Liveness. However, a consequence of their indifference to message sizes is that these models give no meaningful way to analyse some other important properties of a blockchain protocol, such as the maximum *throughput* it can handle (we'll consider throughput formally in Chapter 24).

These models also give limited ability to analyse 'real-world' latency. Formally, we can count the number of rounds of communication required for finalisation. In reality, however, sending larger amounts of data will generally take longer than sending a smaller amount. This means we are committed to an awkward dance in which we consider both the number of rounds of communication required and the message/communication complexity, with no formal way to weigh these metrics. If a protocol uses fewer rounds of communication but has higher communication complexity than another, what does this mean in terms of 'real-world' latency? Generally (but not always), it is the real-world latency that one actually cares about, rather than the round complexity or communication complexity: while potentially of interest in their own right, the latter metrics are generally used as proxies for the former.

To analyse throughput and to accurately weigh the metrics we have introduced in this chapter requires a model in which processors have limited *bandwidth*, i.e., can upload and download messages at a limited rate. We discuss such a model in Chapter 24.

Care in applying the metrics. The considerations above are important when applying the metrics defined in this chapter. For example, an approach that is sometimes taken when analysing amortised communication complexity is to reduce complexity through the use of *batching*: if blocks include enough transactions (say $\Theta(n \log n)$ or $\Theta(n^2)$) then the cost of the remaining 'consensus messages' *per transaction* can be made small. While this may be a useful approach in some contexts, one must also consider the impact on real-world latency. In what settings is it reasonable to suppose that the rate at which transactions are arriving is super-linear in n? What will be the impact on real-world latency if we have to wait for transactions to form blocks?

10.2 Defining the Mempool task and MSMR

Since many protocols work modulo a black boxed mempool protocol, in this section we first define the Mempool task and then consider corresponding complexity metrics.

10.2.1 Defining the Mempool task

As for SMR, we are given as input a transaction set $\mathcal{T}$. The requirement is as follows: there must exist some known bound ℓ such that, whenever a transaction $\text{tr} \in \mathcal{T}$ is received (either from the environment or from another processor) by a correct processor at some time-slot t, all correct processors receive tr by $\max\{t, \text{GST}\} + \ell$.

An alternative version of this definition might drop the requirement for the known bound ℓ, and require simply that tr is eventually received by all correct processors. Exercise 10.3 investigates whether insisting on the known bound ℓ is a stronger requirement for deterministic protocols.

10.2.2 Metrics for mempool protocols

The latency of a mempool protocol is naturally defined as the infimum of all ℓ for which the protocol satisfies the Mempool task. Then we can also consider notions of message and communication that correspond in a natural way to those of Section 10.1.2. For example, single transaction message complexity for a mempool protocol is defined by considering executions where the environment sends only one transaction tr (to at least one correct processor). The complexity is then the maximum number of messages sent by correct processors between tr's first receipt by any correct processor and the first time-slot at which tr is received by all correct processors.

10.2.3 Lower bounds for Mempool metrics

If transactions are immediately forwarded to all other processors upon first receipt, it is clear that latency $= \Delta$. On the other hand, we will generally want to consider (potentially probabilistic) mempool protocols with lower message/communication complexity and larger latency than this.

With regard to message/communication complexity, it is interesting to observe that the quadratic lower bound of Dolev and Reischuk (proved in Section 5.3) already holds for deterministic mempool protocols.

Theorem 10.1. *Consider the lock-step model with signatures and omission faults. Suppose $f < n - 1$. Any deterministic protocol satisfying the Mempool task has single transaction message complexity at least $(f/2)^2$. [Andy: Add in the n bound.]*

Proof. The proof is a simplified version of that in Section 5.3. Towards a contradiction, suppose we are given a protocol violating the claimed bound. Since the claim is trivially true if $f = 0$, we can suppose $f > 0$. First, we take an arbitrary subset $P_1 \subseteq \Pi$ of size $\lceil f/2 \rceil$, setting $P_2 = \Pi \setminus P_1$. Then:

- We consider the execution E_1, in which all processors except those in P_1 receive the transaction tr at time-slot 0 and no processors receive transactions from the environment at other time-slots. The processors in P_1 are faulty. Processors in P_1 act correctly except that they do not send or receive messages to or from each other, and ignore messages from processors in P_2.
- If each processor in P_1 receives at least $\lceil f/2 \rceil$ messages from processors in P_2 in E_1, then we are done, since the correct processors in P_2 must then send at least $|P_1| \cdot f/2 \geq (f/2)^2$ messages (combined). Otherwise, there exists $p \in P_1$ that receives at most $\lfloor f/2 \rfloor$ messages from processors in P_2.
- Now we use the fact that, because P_1 is of size $\lceil f/2 \rceil$, we have $\lfloor f/2 \rfloor$ processors left to be faulty. We consider an execution E_2 in which:
  - Processor p is correct.
  - All processors except those in P_1 receive tr from the environment at time-slot 0. No processor receives any transaction from the environment at time-slots > 0.

- The processors in P_1 other than p act as in E_1, and all processors in P_2 act correctly except that they do not send messages to p (this requires at most $\lfloor f/2 \rfloor$ of them to be faulty).

Analysis. Since p ignored messages from P_2 in E_1, does not receive messages from P_2 in E_2, and does not receive messages from P_1 in either execution, it sends the same messages to processors in P_2 in both executions. In fact, E_1 and E_2 are indistinguishable for all processors in P_2 that are correct in E_2, and since $f < n - 1$ this set is non-empty. So, E_2 is an execution in which a correct processor receives tr at time-slot 0, at most f processors are faulty, and in which p is a correct processor that does not receive tr. This gives the required contradiction. □

Theorem 10.1 concerns deterministic mempool protocols. Many practical mempool protocols use probabilistic gossip mechanisms, utilising techniques like random peer selection, to achieve better than quadratic expected complexity.

Now that we have formally defined the Mempool task, in the next section we formally define the problem to be solved by SMR protocols that work 'modulo' a mempool.

10.3 Mempool-SMR (MSMR)

To formally specify the task that must be satisfied by SMR protocols working 'modulo' a mempool, we simply make extra assumptions on the transactions that processors receive from the environment. So, *Mempool-SMR* (MSMR) is the same as SMR, except we also suppose that, for some known bound ℓ: whenever a correct processor receives a transaction tr at some time-slot t, all correct processors receive tr from the environment by time-slot $\max\{t, \text{GST}\} + \ell$.

When considering MSMR protocols, we modify the metrics of Sections 10.1.1 and 10.1.2 in the obvious way, so as to measure from the first point at which a transactions is received by all correct processors, rather than the first point at which it is received by any correct processor. So, latency is now defined as the supremum, over all executions and all transactions tr first received by a correct processor after GST, of the time between the first time-slot at which tr has been received by all correct processors and the first time-slot at which tr is finalised by all correct processors.

Similarly, the message complexity of an MSMR protocol is the maximum number of messages sent by correct processors (combined) in any interval of the form $[t_1, t_2]$, such that $t_1 \geq \text{GST} + \Delta$, some transaction tr has been received by all correct processors by t_1, and t_2 is the first time-slot at which tr is finalised by all correct processors.

The definitions of communication complexity and single transaction message/communication complexity are modified similarly. The definition of amortised complexity remains unchanged.

10.3.1 Lower bounds for MSMR

We consider latency first, and then message/communication complexity.

Latency. Essentially the same proof as for Theorem 5.2.5 (see Exercise 10.4) suffices to establish an $f\Delta$ lower bound on latency for deterministic MSMR protocols in the synchronous model with signatures and crash failures.

Message and communication complexity. We have shown that the Mempool task already requires quadratic (single transaction) message complexity for deterministic protocols. This raises the question as to whether such a lower bound still holds for deterministic MSMR protocols. In fact, it does, but the simplest proof we are aware of is more complex than the original proof by Dolev and Reischuk.

[Andy: State and prove result.]

10.4 Analysing efficiency for Tendermint

[Andy: Discuss all metrics for all three variants. Although we have not yet formally discussed how to model threshold signatures and random leaders, can still include discussion of these, with a pointer to the fact that formal modelling will be treated later. Add a comment and exercise establishing that communication via the leader can be used to give $O(n^2)$ single transaction message complexity (and $O(n^2)$ communication complexity with blah blah blah).]

10.5 Exercises

Exercise 10.1. *[Andy: Walk through proof that latency is at least $(f+1)\Delta$.]*

Exercise 10.2. *[Andy: Walk through proof that deterministic protocols require $\Omega(n+f^2)$ single transaction message complexity]*

Exercise 10.3. *[Andy: Investigation as to whether requiring ℓ for the mempool is a stronger requirement for deterministic protocols.]*

Exercise 10.4. *[Andy: Walk through proof that MSMR requires latency at least $f\Delta$.]*

Chapter 11

PBFT's view-change mechanism

Probably the best-known SMR protocol for partial synchrony (without synchronised clocks) is PBFT [20], which stands for Practical Byzantine Fault Tolerance. PBFT was introduced by Castro and Liskov in 1999, seventeen years before Tendermint, at a time when demands on SMR protocols were rather different from those in the context of 'blockchain' (we'll expand on some of these differences in Section 11.5). Accordingly, there are a number of somewhat superficial differences between the original PBFT protocol and Tendermint. For our purposes, the crucial difference is the view-change mechanism, and that is what we will focus on in this chapter. As we will see, an advantage of PBFT's approach is that it gives a strong form of *optimistic responsiveness*: roughly, this means that latency is a function of the *actual* (and unknown) network delay, rather than the known upper bound Δ. This is important because Δ may be conservatively set to ensure liveness. However, this comes at the cost of higher communication complexity during view changes.

In Section 11.1, we first outline the intuition behind PBFT's approach to view changes. In Section 11.2, we then formally describe and analyse a protocol that implements view changes using PBFT's approach. We define two forms of optimistic responsiveness in Section 11.3, and then compare performance for PBFT's and Tendermint's view-change mechanisms in Section 11.4. In Section 11.5, we summarise other differences between the original PBFT protocol and Tendermint.

11.1 PBFT's view changes: the intuition

In this section and the next, we'll describe an implementation of PBFT's view-change mechanism that we'll call *Streamlined PBFT*. The protocol is optimised to work efficiently in a 'blockchain' context, in which we construct a chain of blocks of transactions, with a new leader responsible for proposing each block. We'll also specify the protocol so that it is easily adapted to make use of threshold signatures (as discussed in Section 9.5.5).

Recall that the version of Tendermint described in Section 9.4 used a *locking* mechanism. Upon seeing Q which is a stage 1 QC for some view v block b while in view v, processors set their lock equal to Q before sending Q and a stage 2 vote for b to all processors. Upon entering view $v + 1$, the leader for that view waits for 2Δ to ensure that (if the

view starts after GST) they have received the locks of all correct processors. After this wait of 2Δ, they take Q' which is the greatest stage 1 QC they have received (with QCs ordered by view). If Q' is a QC for b', then the leader proposes a block with b' as parent. If the view starts after GST then Q' will be greater than or equal to the locks of all correct processors, which means that all correct processors will be able to vote for the proposed block.

The approach we will take now drops the locking mechanism. Recall that a TC (time-out certificate) for view v is a set of $n-f$ time-out messages, each from a different processor. As for the version of Tendermint in Section 9.4, upon receiving Q that is either a stage 2 QC for some view v block or else a TC for view v, p_i will forward Q to all processors and enter view $v+1$. Now, however, processors also send a *new-view* message to the leader of view $v+1$ upon entering view $v+1$ because they receive a TC for view v. A new-view message by p_i includes:

(a) a *view attestation*: this is a signed message indicating the greatest view for which p_i has seen a stage 1 QC, and;

(b) a stage 1 QC for that view.

Upon entering view $v+1$, the leader for that view no longer has to wait for a fixed duration of 2Δ. If they receive a stage 2 QC for a view v block b', then this is immediate proof that they should propose a block b with b' as parent (as noted in Section 9.5.2, Tendermint can also be modified to use this trick). Otherwise, they simply wait to receive $n-f$ new-view messages. Upon receipt of such a set of messages, they take the greatest stage 1 QC included in any of those $n-f$ new-view messages: suppose this is Q, which is a stage 1 QC for b' (say). They then construct a *justification* for proposing a block b with b' as parent. This justification is just Q, together with the set V of $n-f$ signed view attestations included in the $n-f$ new-view messages. Processors will then vote for b so long as it includes a (correctly formed) justification, in which the specified stage 1 QC is for the same view as the greatest view attestation in V. Crucially, the leader no longer waits for a fixed duration of 2Δ. Instead, they can propose as soon as they receive sufficient information, which may arrive much faster than 2Δ when actual network delays are small.

Why does this approach work? Suppose some block b_1 for view v_1 is finalised. Towards a contradiction, suppose there is some least $v_2 \geq v_1$ such that some block b_2 for view v_2 receives a stage 1 QC, but does not have b_1 as an ancestor. By the standard quorum intersection argument (see Section 9.1.2), we have $v_2 > v_1$. Let P be the set of processors who contribute to the stage 2 QC for b_1, noting that correct processors in P see a stage 1 QC for b_1 before leaving view v_1. Let P' be the set of $n-f$ processors whose signed view attestations are included in the justification for b_2. By our choice of v_2, the justification included in b_2 must specify a stage 1 QC, Q say, with $Q.\text{view} < v_1$. By the standard quorum intersection argument, $|P \cap P'| \geq f+1$ and so must include a correct processor. It follows that the justification for b_2 must include a view attestation for some view $\geq v_1$. This gives the required contradiction.

11.2 PBFT's view changes: formal treatment

11.2.1 Streamlined PBFT: the formal specification

The pseudocode uses a number of message types, local variables, functions and procedures. Many of these are unchanged from Chapter 9, but for ease of reference we repeat them below.

In what follows, we suppose that, when a correct processor sends a message to 'all processors', it regards that message as immediately received by itself. We will describe the protocol as an MSMR protocol, i.e., we assume that, whenever a correct processor receives a transaction tr at some time-slot t, all correct processors receive tr from the environment by time-slot $\max\{t, \mathrm{GST}\}+\ell$. We write $\emptyset$ to denote the empty set or empty sequence, and we let $\perp$ be some default distinguished value.

The function $\texttt{lead}(v)$. The value $\texttt{lead}(v)$ specifies the *leader* for view v. To be concrete, we set $\texttt{lead}(v) := p_i$, where $i = v \bmod n$.

Votes. For $d \in \{1, 2\}$, a stage d *vote* is a message of the form $(\text{vote}, v, d, \tau)$ signed by some processor in Π, where $v \in \mathbb{N}_{\geq 0}$ specifies the view corresponding to the vote and τ is a finite binary string. If b is a block with hash τ, we also say the vote is a *vote for* b.

Stage 1 and 2 QCs. For $d \in \{1, 2\}$, a stage d QC is a set Q of $n - f$ votes, each of the form $(\text{vote}, v, d, \tau)$ for the same values of v and τ, and each signed by a different processor in Π. We set $Q.\text{view} = v$, $Q.\text{block} = \tau$. Stage 1 QCs are ordered by view and then by least hash. We also say Q is a stage d QC for τ and, if b is a block with hash τ, we say Q is a stage d QC for b.

Time-out messages and TCs: a time-out message for view v is a message of the form $(\text{time-out}, v)$, signed by some processor in Π. A set of $n - f$ messages of this form, each signed by a different processor, is called a time-out certificate (TC) for view v.

View attestations. Recall that $\langle m \rangle_i$ is the message m signed by p_i. A *view* v *attestation* by p_i is a signed tuple $\langle v, i, v' \rangle_i$ for some $v' \in \mathbb{N}_{\geq 0}$ with $v' < v$. When we wish to make the value v' explicit, we also refer to such a tuple as a *view* v *attestation for* v'. (If p_i sends a new-view message that includes the view v attestation $\langle v, i, v' \rangle_i$, this will indicate that, upon entering view v, v' was the greatest view for which p_i had seen a stage 1 QC.)

New-view messages. A *new-view message* for view v by p_i is a tuple of the form $(\langle v, i, v' \rangle_i, Q)$, where Q is a stage 1 QC with $Q.\text{view} = v'$.

Justifications. A view v *justification* is either:

(a) A stage 2 QC for a view $v - 1$ block b', or;

(b) A pair (Q, V), where:

- V is a set of $n - f$ view v attestations, each by a different processor in Π.
- Let v' be the greatest view such that V contains a view v attestation for v'. Then Q is a stage 1 QC for a view v' block, b' say.

When (a) or (b) above holds, we also say that the view v justification is a *view v justification for the parent b'*.

Blocks. A *block* b is a message specified by four values:

- b.view: this is a value in $\mathbb{N}_{\geq 0}$ that specifies the view corresponding to b;
- b.Tr: a sequence of transactions in $\mathcal{T}$;[1]
- b.par: either $\perp$, or a hash value specifying the parent of b;
- b.just: a view b.view justification for the parent b.par if b.par $\neq \perp$.

A correct processor only regards a message as a block if it is of the form above. If b.view $= v$, we also refer to b as a 'view v block'. The *genesis block* is denoted b_g and satisfies b_g.view $= 0$, b_g.Tr $= \emptyset$, b_g.par $= \perp$, b_g.just $= \perp$. We regard b_g as received by all correct processors at time-slot 0. Blocks are ordered by b.view and then by least hash. We stipulate that $\emptyset$ is a stage 1 QC and a stage 2 QC for b_g, and set $\emptyset$.view $= 0$: all processors therefore begin the protocol execution having received stage 1 and stage 2 QCs for b_g.

The sequence of transactions specified by the ancestors of a block. Each block b specifies an extended sequence of transactions, denoted b.Tr*, as follows: we concatenate the values b'.Tr for all ancestors b' of b, removing any duplicate transactions.

The procedure MakeProposal. This procedure is executed by the leader p of view v upon receiving Q that is either a stage 2 QC for a view $v-1$ block or a set of $n-f$ new-view messages for view v, each by a different processor in Π. To execute the procedure, p proceeds as follows:

1. Form a sequence of distinct transactions T, containing all transactions received but not finalised by p;
2. Set b.view $:= v$, b.Tr $:= T$.
3. If Q is a stage 2 QC for a view $v-1$ block b', set b.par $:= Q$.block and b.just $= Q$.
4. Otherwise, if Q is a set of $n-f$ new-view messages, let V be the set of $n-f$ view attestations in messages in Q. Let v' be the greatest view such that V contains a view v attestation for v', and let Q' be a stage 1 QC included in one of the new-view messages in Q that include a view v attestation for v' (so that Q'.view $= v'$). Set b.par $= Q'$.block and b.just $:= (Q', V)$.
5. Send b to all processors.

Finalising blocks and transactions. Processor p regards b as finalised upon receiving a stage 2 QC for b. However, the transactions in b.Tr cannot be finalised until p has received all ancestors of b. Formally, we define $\mathcal{F}$ (as required for SMR in Section 7.3) as follows. For any set of messages M, let b be the greatest block such that $M \cup \{b_g\}$ contains: (i) a stage 2 QC for b, and (ii) all ancestors of b. Set $\mathcal{F}(M) := b$.Tr*.

[1] Recall that $\mathcal{T}$ is the set of possible transactions.

The local variable v: this is a local variable, which p uses to record the present view. Initially, $\mathtt{v} = 1$.

The local variables 1voted, 2voted and proposed: these record whether p_i has already sent stage 1 and stage 2 votes in the present view, and whether p_i has already proposed a block for the view. They are initially set to 0.

The timer. Each processor p has a local timer, denoted Timer, which it can set to 0 at any time, and which then automatically increments at each time-slot at which p is active. This means that the timer increments in 'real time' after GST. Initially, $\mathtt{Timer} = 0$.

The protocol instructions are specified in Figure 11.1.

Streamlined PBFT: the instructions for p_i.
At time-slot 0:
Set $\mathtt{v} := 1$, $1\mathtt{voted} := 0$, $2\mathtt{voted} := 0$, $\mathtt{proposed} := 0$, $\mathtt{Timer} := 0$; ▷ initialise

At every time-slot t:

If $p_i = \mathtt{lead}(\mathtt{v})$, $\mathtt{proposed} = 0$, and p_i has received Q that is either a stage 2 QC with $Q.\text{view} = \mathtt{v} - 1$, or else is a set of $n - f$ new-view messages for view v, each by a different processor in Π:
MakeProposal; Set $\mathtt{proposed} := 1$; ▷ propose a new block if leader

If p_i has received a view v block b from $\mathtt{lead}(\mathtt{v})$ and $1\mathtt{voted} = 0$:
Send b and a stage 1 vote for b to all processors; ▷ stage 1 vote
Set $1\mathtt{voted} := 1$;

If p_i has received a view v block b from $\mathtt{lead}(\mathtt{v})$ and Q which is a stage 1 QC for b and if $2\mathtt{voted} = 0$:
Send a stage 2 vote for b to all processors; ▷ stage 2 vote
Set $2\mathtt{voted} := 1$;

If p_i has received Q which is a stage 2 QC with $Q.\text{view} = v' \geq \mathtt{v}$ or which is a TC for view $v' \geq \mathtt{v}$:
Send Q to all processors;
If Q is a TC for view v':
Let v'' be the greatest view $\leq v'$ such that p_i has received some stage 1 QC Q' with $Q'.\text{view} = v''$;
Send the new-view message $(\langle v' + 1, i, v'' \rangle, Q')$ to $\mathtt{lead}(v' + 1)$;
Set $\mathtt{v} := v' + 1$, $1\mathtt{voted} := 0$, $2\mathtt{voted} := 0$, $\mathtt{Timer} := 0$, $\mathtt{proposed} := 0$;
▷ enter higher view

If $\mathtt{Timer} = 5\Delta$:
Send a time-out message for view v to all processors ▷ send time-out

FIGURE 11.1: The pseudocode for Streamlined PBFT

11.2.2 Streamlined PBFT: verifying Consistency and Liveness

Throughout this section, we consider the partially synchronous model (without synchronised clocks), signatures and Byzantine faults, and we suppose that $f < n/3$. First, we consider Consistency.

Lemma 11.1 (Consistency)**.** *Streamlined PBFT satisfies Consistency.*

Proof. The proof works exactly as already specified in Section 11.1. □

Next, we show that correct processors progress through the views.

Lemma 11.2. *For each $v \geq 1$, there is some first time-slot, t_v say, at which a correct processor enters view v. Also, $t_{v'} > t_v$ if $v' > v$.*

Proof. The proof is the same as the proof for Tendermint in Section 9.4.1. It follows immediately that no correct processor can enter view $v+1$ before any correct processor has entered view v, because entering view $v+1$ requires either a stage 2 QC or a TC for view v, to which some correct processors must contribute. To show that each t_v is defined, note that, upon entering any view, a correct processor sends the relevant certificate (the stage 2 QC or TC) to all others. So, if any correct processor enters infinitely many views, they all do. If there were some greatest view v entered by any correct processor p, then all correct processors eventually enter view v. Then all correct processors send time-out messages for view v, and so receive a TC for view v. □

Next, we establish the equivalent of Lemma 9.2.

Lemma 11.3 (Correct leaders finalise new blocks)**.** *For each $v \geq 1$, let t_v be as defined in the statement of Lemma 11.2. If $t_v \geq GST$ and $p = \texttt{lead}(v)$ is correct, then p proposes a view v block b and all correct processors receive a stage 2 QC for b.*

Proof. Suppose the conditions in the statement of the lemma hold. Towards a contradiction, suppose it is not the case that p proposes a view v block b and all correct processors receive a stage 2 QC for b by $t_v + 6\Delta$. If any correct processor received such a stage 2 QC by $t_v + 5\Delta$ it would send it to all other processors, which gives an immediate contradiction. No correct processor sends a time-out message for view v before $t_v + 5\Delta$, so no processor can receive a TC for view v before this time. Since the first correct processor to enter view v sends the corresponding certificate (a stage 2 QC or TC for view $v-1$) to all processors at t_v, it follows that all correct processors are in view v for the entirety of the interval $[t_v + \Delta, t_v + 5\Delta]$. This holds because no correct processor can leave view v without receiving either a stage 2 QC or TC for a view $\geq v$, neither of which can happen before $t_v + 5\Delta$.

We have observed that all correct processors (including $p = \texttt{lead}(v)$) enter view v by $t_v + \Delta$. If any correct processor enters view v upon receiving a stage 2 QC for view $v-1$, it sends that QC to all processors and they receive it by $t_v + 2\Delta$. If not, then all correct processors send new-view messages for view v to p by $t_v + \Delta$, meaning that p receives (correctly formed) new-view messages from at least $n - f$ processors by $t_v + 2\Delta$. In either case, p sends a (correctly formed) view v block b to all processors by $t_v + 2\Delta$, which is received by all correct processors by $t_v + 3\Delta$. All correct processors then send a vote for b by $t_v + 3\Delta$, and receive a stage 1 QC for b by $t_v + 4\Delta$. All correct processors then send a stage 2 vote for b, and receive a stage 2 QC for b by $t_v + 5\Delta$. This gives the required contradiction. □

To establish Liveness, it remains to prove the equivalent versions of Lemmas 9.3 (if p is correct and finalises b then all correct processors receive all ancestors of b) and 9.4 (Liveness, i.e., every transaction received by a correct processor is eventually finalised). The proofs of these lemmas are the same as for Tendermint.

11.3 Optimistic responsiveness

Roughly, optimistic responsiveness is the ability to finalise transactions at network speed, i.e., with latency proportional to the actual (unknown) network delay, so long as processors act correctly and while the network is synchronous. There are a number of ways to formalise the notion. In this section we consider two simple variants.

Weakly optimistically responsive protocols. For a given execution, let δ be as defined in Section 8.3, i.e., let δ be the least value such that any message sent to any processor at any time-slot t is delivered by $\max\{t, \text{GST}\} + \delta$. Note that δ may be less than the known bound Δ. Let $f_a \leq f$ be the actual (unknown) number of processors that do not behave correctly. A protocol is *weakly optimistically responsive* if there exists some known bound Δ^* which is $O(\Delta)$ and such that latency for any transaction first received by a correct processor[2] after GST+Δ^* is $O(f_a\Delta + \delta)$.

So, roughly, the definition allows for a 'grace period' after GST. Once that grace period has passed, each faulty processor (in particular leaders) can cause a delay $O(\Delta)$, but the remaining contribution to latency is $O(\delta)$. Specifically, if all processors are correct, latency will be $O(\delta)$ after GST+Δ^*.

For most well-known protocols it will actually be the case that only faulty *leaders* can cause delays, so that, for weakly optimistically responsive protocols, latency will be $O(\delta)$ after GST+Δ^* so long as leaders are correct. We give a general definition that does not explicitly refer to leaders, so that it applies uniformly to all protocols for the partially synchronous model, including those that are not leader-based.

Strongly optimistically responsive protocols. The definition is the same, except that we do not allow for a 'grace period'. Let δ and f_a be defined as above. A protocol is *strongly optimistically responsive* if latency for any transaction first received by a correct processor after GST[2] is $O(f_a\Delta + \delta)$.

The key difference is that strongly optimistically responsive protocols achieve good latency immediately after GST, whereas weakly optimistically responsive protocols may require a grace period for the system to stabilise.

[2]This definition is given for SMR protocols. For MSMR, one instead measures from the first time-slot by which the transaction has been received by all correct processors.

11.4 Comparing performance for PBFT/Tendermint view changes

11.4.1 Communication complexity

Communication complexities for Tendermint and Streamlined PBFT are similar. In Streamlined PBFT, each processor may send a new-view message to the leader upon entering view v. This message is of size $\Theta(n)$, so that the combined cost over all processors is $O(n^2)$. If threshold signatures are used (see Section 9.5.5), then new-view messages will be of constant size, giving $O(n)$ combined cost over all processors.

In both protocols, the leader sends a block to all. Suppose the transaction information included in the block is of constant size. For both protocols, the block includes a stage 1 QC, which is of size $\Theta(n)$, or of constant size if threshold signatures are used. For Streamlined PBFT, the block may also include a set of view attestations, which is of size $\Theta(n)$. Since the leader must send the block to all processors, the total cost for Tendermint is $\Theta(n^2)$, or $\Theta(n)$ if threshold signatures are used. For Streamlined PBFT, the need for the block to include n view attestations (in some cases) means that the total cost is $\Theta(n^2)$ in the worst case, even if threshold signatures are used.

For both protocols, the sending of stage 1 and stage 2 votes induces a total communication cost that is $O(n^2)$ per view. Tendermint requires processors to also send a stage 1 QC to all processors upon sending any stage 2 vote. This induces a communication cost that is $\Theta(n^3)$, or $\Theta(n^2)$ if threshold signatures are used. Both protocols also require processors to send stage 2 QCs to all processors upon first receiving them, inducing a communication cost of the same order.

Finally, the sending of time-out messages and TCs induces $O(n^3)$ cost per view, or $O(n^2)$ cost if threshold signatures are used. Overall, the cost for both protocols is $\Theta(n^3)$ per view, or $\Theta(n^2)$ with the use of threshold signatures.

So why did we say PBFT's approach has higher complexity? The version of PBFT we have described here—Streamlined PBFT—is optimised for a blockchain context, with rotating leaders proposing successive blocks. As we'll see in Section 11.5, the original PBFT used a stable leader who remains in charge until processors requested a change due to lack of progress. Since view changes were rarer in that design, the protocol was not optimised to handle them efficiently.

Even comparing *Streamlined* PBFT and Tendermint, however, there *is* a fundamental difference in the communication complexities induced by view changes. This difference will become significant when we consider Hotstuff in Chapter 13, and to understand it one has to differentiate between mechanisms for *view change* and mechanisms for *view synchronisation*.

Mechanisms for view change and view synchronisation. Roughly, a mechanism for *view change* ensures that the leader receives enough information to propose a block that other processors can vote for, while maintaining Consistency and Liveness. In Tendermint this is achieved via a locking mechanism, while PBFT uses justifications. On the other hand, a mechanism for *view synchronisation* ensures that, once any correct processor enters a view after GST, all correct processors do so within some short time (such as Δ). Generally, this is required to ensure that each view with a correct leader

that starts after GST finalises a new block. Tendermint and Streamlined PBFT both achieve view synchronisation in the same way: processors enter views upon receiving stage 2 QCs or TCs, and immediately forward these to all other processors. This induces $\Theta(n^3)$ communication cost per view, or $\Theta(n^2)$ cost per view with the use of threshold signatures.

Looking forward to Hotstuff. One of the basic aims of Hotstuff is to achieve communication cost that is $O(n)$ in each view *modulo* the mechanism for view synchronisation. More sophisticated mechanisms for view synchronisation can then be combined with Hotstuff (see Chapter 14) to achieve single transaction communication complexity $O(n^2)$. The quadratic cost arises because $f + 1$ views may be required after GST to finalise a new block, each inducing linear cost. Hotstuff achieves linear complexity within each view using a simple idea: as well as using threshold signatures, all communication within each view is relayed through the leader (see Exercise 9.2). If n processors each send votes of constant size to the leader, who then forms a threshold signature of constant size and sends this to all processors, the communication cost induced is linear, as opposed to the quadratic cost induced if all processors send votes to all other processors. However, the view-change mechanism employed by PBFT already induces quadratic cost in the case that the leader must send a set of attestations of size $\Theta(n)$ to all processors. For this reason, Hotstuff uses a view-change mechanism based on locking (similar to Tendermint's), but modified to also achieve strong optimistic responsiveness.

11.4.2 Comparing optimistic responsiveness

Weak optimistic responsiveness. The version of Tendermint described in Section 9.4 is not weakly optimistically responsive, because leaders always wait 2Δ after entering the view before proposing a new block. However, we also observed in Section 9.5.2 that the protocol is easily modified so that the leader of view v proposes a block immediately upon receiving a stage 2 QC for a view $v - 1$ block. To see that this modified protocol is weakly optimistically responsive, we analyse latency under the MSMR model. Let v be the greatest view that any correct processor is in at GST and let GST $+ \Delta^*$ be the first time at which any correct processor is in view $v+2$, noting that Δ^* is $O(\Delta)$. Recall that δ is the (unknown) actual maximum message delay after GST, and that f_a is the (unknown) actual number of faulty processors.

Suppose first that $f_a = 0$. Then, since all leaders are correct, view $v+1$ produces a new finalised block in time $O(\Delta)$ (the delay $O(\Delta)$ results because view v may not produce a new finalised block), and each view $\geq v + 2$ produces a new finalised block in time $O(\delta)$. It follows that latency for any transaction first received by a correct processor after GST $+ \Delta^*$ is $O(\delta)$.

If $f_a > 0$, then suppose the transaction tr is first received by all correct processors at $t \geq$ GST $+ \Delta^*$. Let v' be the greatest view that any correct processor is in at t, and let v'' be the least view $> v'$ with a correct leader. Then all correct processors enter view v'' by $t + O(f_a\Delta)$ and then finalise tr in time $O(\delta)$ after entering the view.

We will see below that Streamlined PBFT is weakly optimistically responsive because it is strongly optimistically responsive.

Strong optimistic responsiveness. To see that the modified version of Tendermint described in Section 9.5.2 is not strongly optimistically responsive, suppose $f_a = 0$.

Suppose all correct processors enter some view v at GST and also receive the transaction tr at GST. Suppose further that view $v-1$ does not produce any finalised block. Then the correct leader of view v waits 2Δ before proposing a new block, meaning that latency for tr is $\Omega(\Delta)$.

It remains to show that Streamlined PBFT is strongly optimistically responsive. To see this, suppose tr is received by all correct processors by $t \geq$ GST. Let v be the greatest view that any correct processor is in at t. Suppose first that the leader of view v is correct. Then all correct processors will enter view $v+1$ by $t + O(\delta)$. Let v' be the least view $\geq v+1$ with correct leader. All correct processors will enter view v' by $t + O(\delta + f_a\Delta)$ and will finalise tr in time $O(\delta)$ once in view v'. If the leader of view v is faulty, then all correct processors enter view v' by $t + O(f_a\Delta)$ and will again finalise tr in time $O(\delta)$ once in view v'.

11.5 The original PBFT

This section describes how the original PBFT protocol differs from Streamlined PBFT. We do not give a formal specification, but provide sufficient detail to make the basic protocol operation clear.

Stable leaders. In the original version of PBFT, each view has a designated leader as in Streamlined PBFT. However, processors remain in each view until they see evidence that the leader may be faulty. As detailed below, the leader of a view can make an unbounded number of successive proposals, and it is only if sufficient time passes after a processor receives a transaction without it being finalised that they send a message to all other processors indicating that they wish to enter a new view. While the use of stable leaders has efficiency advantages (removing the need for view changes while the leader is correct during synchrony) it is generally avoided in the 'blockchain' context for at least two reasons:

- So far, we have considered transactions only as abstract messages to be sequenced. However, transactions will generally be instructions to update some database: each transaction is really an operation that, when executed, updates the *state* of the database (or 'state machine'). In a context where these operations have financial consequence, the ability of a leader to choose transaction ordering gives it a greater level of control than is preferable for any single participant over an extended period.[3]
- In the stable leaders approach, a single Byzantine leader can slow the protocol for an unbounded period during synchrony by operating as if message delays are always Δ.

Proposals are for individual transactions. Since PBFT was intended for a context with lower transaction throughput, standard operation for PBFT involves the leader proposing individual transactions. Rather than considering blocks of transactions, with

[3] A leader's ability to choose transaction ordering can be financially valuable. For example, a leader receiving a large 'buy order' on an exchange may be able to insert buy and sell orders either side of the transaction to profit from the resulting price change. The profit that can be extracted by choosing transaction ordering is often referred to as 'MEV' (maximum extractable value) [].

hash pointers used to specify the parent of each block, the leader allocates each received transaction to a different *slot* number s. Upon hearing a proposal from the leader that transaction tr should be allocated to slot number s, processors send stage 1 votes for the proposal. Upon receiving a stage 1 QC, processors send stage 2 votes, and then finalise tr in slot s upon receiving the corresponding stage 2 QC. There is no set bound on the number of transactions that can be finalised in a given view.

Transaction execution and garbage collection. In PBFT, processors *execute* a transaction in slot s upon finalising the transaction in slot s, if they have already executed transactions for all previous slots. To remove the need to store old messages, the protocol also uses 'checkpoints' at which proofs of the current state of the database/state machine are produced. For some constant c (e.g., $c = 100$), whenever a processor executes a transaction for a slot number that is a multiple of c, they send a message to all others indicating the slot number and the resulting state. By the standard quorum intersection argument, $n - f$ such messages all indicating the same resulting state corresponding to the same slot number constitute a *checkpoint-proof* that the state at that slot number is correct. Upon receiving such a proof, processors can discard messages corresponding to previous slot numbers.

View changes. In Streamlined PBFT, we separated time-out messages and new-view messages. The former are sent to all processors for the purpose of view synchronisation, while the latter are sent to the leader so that they can form a justification for their proposal. In PBFT, the information in new-view messages is piggy-backed on the time-out messages, and is therefore sent to all processors. When a processor times out during view v they no longer vote for proposals in the view. To form an appropriate time-out message, they take the checkpoint-proof for the greatest slot number s amongst those they have received. In their time-out message, they include this checkpoint-proof and every stage 1 QC they have received corresponding to a slot number greater than s.

Upon receiving a set $\mathcal{T}$ of $n - f$ time-out messages for view v, the leader of view $v + 1$ forms a message for the start of that view that includes the messages in $\mathcal{T}$. PBFT then applies the view-change approach of Streamlined PBFT to individual slots. Let s' be the greatest slot such that some message in $\mathcal{T}$ includes a checkpoint-proof for slot s'. Let s'' be the greatest slot such that any message in $\mathcal{T}$ includes a stage 1 QC for slot s''. For each slot $s \in (s', s'']$, if any message in $\mathcal{T}$ contains a stage 1 QC for slot s, then the leader's message for the start of the view contains a re-proposal for the same corresponding transaction. Upon receiving a correctly formed leader's message for the start of the view, correct processors will send stage 1 votes for this re-proposal. By the same quorum intersection argument as for Streamlined PBFT, this ensures Consistency: any transaction finalised by a correct processor for some slot $s \in (s', s'']$ must be re-proposed in that slot, meaning that no other transaction can be finalised in that slot. For each slot s in $(s', s'']$ such that no message in $\mathcal{T}$ contains a stage 1 QC for slot s, the new leader proposes the 'null transaction', and correct processors send stage 1 votes for this proposal.

View synchronisation. Upon receiving a TC for view v, Streamlined PBFT had processors send this TC to all others before entering view $v+1$. This ensured that (after GST) all correct processors enter the next view within time Δ. The original PBFT protocol takes a slightly different approach, in which processors do not resend a received TC to all others. Upon receiving a TC for view v, a processor p immediately starts a timer, whose expiration will cause it to send a time-out message for view $v + 1$. At any

point, if p receives $f+1$ time-out messages, each of which corresponds to a view greater than or equal to its present view, p sends a time-out message for the smallest view in this set, even if its timer has not expired. This suffices to ensure that, after GST, if p receives a TC for view v at t, then all correct processors receive such a TC by $t+2\Delta$: since p has received $n-f$ time-out message by t, all processors must receive $n-2f \geq f+1$ time-out messages by $t+\Delta$, and so will send timeout messages by this time if they have not already done so. Then all correct processors receive at least $n-f$ time-out messages by $t+2\Delta$. A disadvantage of this approach is that it increases required time-outs, since processors are synchronised to within 2Δ, rather than Δ as for Streamlined PBFT.

Partial synchrony. A final difference is that the original PBFT paper does not assume our standard version of partial synchrony. Let delay(t) be the maximum message delay for any message sent at time-slot t in a given execution. Then the assumption is that, in any given execution, 'delay(t) does not grow faster than t indefinitely'. To accommodate this assumption, PBFT utilises exponentially increasing successive time-outs. If p times out without seeing any new transactions finalised in view v, then it sets the expiry time for view $v+1$ to be twice that for view v.

11.6 Exercises

Exercise 11.1. *[Andy: Exercise on piggybacking new-view messages on time-outs. Does this impact strong optimistic responsiveness?]*

Chapter 12

Using oracles to model cryptographic primitives

In Chapters 9 and 11, we introduced Tendermint and PBFT. These protocols can be adapted to use threshold signatures, though they do not do so by default. In analysing the communication complexity of both protocols, we have already informally considered the impact of threshold signatures. In the next chapter, we will describe Hotstuff, which uses threshold signatures by default. As preparation, this chapter gives a simple way to model threshold signatures using an *oracle*. The oracle abstraction allows us to analyse protocols without delving into cryptographic implementation details, while still capturing the essential properties that the cryptographic primitives provide.

We also show how oracles can be used to model two other cryptographic primitives: a *common reference string* (CRS), which provides shared setup information available to all processors, and a *common coin*, which provides shared randomness. Both primitives can be used for random leader selection. In Chapter 16, we will show how a common coin can be used to circumvent the FLP Impossibility Theorem for the asynchronous model.

In Section 12.1, we describe a simple modification of the state-transition-diagram model that allows processors to make repeated oracle queries within a single time-slot. In Section 12.2, we then show how to model threshold signatures, a CRS, and a common coin.

12.1 Modifying the state-transition-diagram model

In this book, we do not attempt to define a general notion of oracle sufficient to model any cryptographic primitive. In Section 12.2, we simply specify three different types of oracle, used to model threshold signatures, a CRS, and a common coin. If a protocol uses one of these oracles, then processors can send messages to the oracle (referred to as 'querying' the oracle). The oracle will respond by sending messages to some processors. Formally, we suppose there exists a two-way communication channel between each processor and each oracle.

One slightly subtle point is that some oracles may respond "instantaneously" to queries, and processors should be able to query such oracles repeatedly in the same time-slot. (By contrast, messages sent by one processor to another in a time-slot t cannot arrive at their destinations prior to time-slot $t + 1$.) For example, p may query an oracle that models a threshold signature scheme, and then make further queries at the same time-slot depending on the response.

For this reason, we modify state-transition-diagrams to specify a special subset St^* of the set St of all states, indicating the states from which the processor will proceed to the next time-slot without further oracle queries. At the beginning of the protocol's execution, a processor p begins in an *initial state* in St^*, which is determined by p's inputs. At each time-slot t for which p is active, p begins in some state $s^* \in St^*$ and receives a multi-set of messages on each of its channels. Processor p then enters a state s (which may or may not belong to St^*). In the deterministic model, s is determined by s^*, and the multi-set of messages received on each of its channels. In the probabilistic model, s is a random variable with distribution determined by these values. Processor p then repeatedly carries out the following steps, until instructed to proceed to the next time-slot:

(1) Processor p receives R, which is a multi-set of messages sent to p by the oracles.

(2) If $s \in St^*$, then p sends a set of messages M and stops instructions for time-slot t. The set of messages M is determined by s and R. Processor p will begin its next active time-slot in state s.

(3) If $s \notin St^*$, then p sends a set of oracle queries Q and enters a new state s', where Q and s' are determined by s and R. Set $s := s'$ and return to (1).

For certain state-transition-diagrams, the instructions above might not terminate. In this case, p sends no messages to other processors at time-slot t and is regarded as inactive at all subsequent time-slots. If p is inactive at time-slot t, then it does not send or receive any messages (to processors or oracles), and does not change state.

12.2 Oracle examples

First, we show how to use an oracle to model a threshold signature scheme. Then we consider how to model a CRS and a common coin.

12.2.1 Modelling a threshold signature scheme

For any given message m, a 'k-of-n threshold signature scheme' allows *signature shares* on m from any set of k out of the n processors to be combined to form a single signature on m of constant-bounded length. The resulting threshold signature acts as proof that at least k processors have signed m, but does not indicate which processors have signed the message. Recall that H is a collision-free hash function. To model such a scheme, the oracle O behaves as follows:

- To form its signature share on the message m, p_i sends the message (share, m) to the oracle. Upon sending this message to the oracle, p_i receives the response $\langle H(m), i \rangle_O$ from the oracle upon transitioning to any new state. This message is signed by the oracle O, and indicates for which processor it is the signature share on m.
- To form a threshold signature on m given a set M of signature shares on m from k different processors, p_i sends $(\text{combine}, M)$ to the oracle. Upon sending this message to the oracle, p_i receives the response $\langle H(m) \rangle_O$ from the oracle upon transitioning to any new state. This message is the threshold signature on m.

Since the signature share is signed by the oracle, only p_i can form its own signature share on the message m. Similarly, since the threshold signature on m is signed by the oracle, it can only be formed by a processor that has signature shares on m from k different processors.

12.2.2 Modelling a CRS

It will often be useful for all processors to have access to some randomly chosen 'common reference string'. For example, this can be used to specify random leaders in a context where processors should know the leader of each view ahead of time. Our CRS oracle behaves as follows:

- At the start of the protocol execution, but *after* the adversary (if static) has chosen which processors are faulty, the oracle samples an infinite binary string x uniformly at random.
- To determine $x(k)$ (the k^{th} bit of x), processor p_i sends the query k to the oracle. Upon sending this message to the oracle, p_i receives the response $x(k)$ from the oracle upon transitioning to any new state.

We note that when analysing *expected* latency for protocols that use a CRS to determine leaders, it is common to consider GST to be chosen uniformly at random (rather than being adversarially chosen, as standard). This is because the adversary could otherwise use knowledge of the CRS and choose GST to ensure a maximum possible number of views after GST have faulty leaders.

12.2.3 Modelling a common coin

The difference between a CRS and a common coin is that the latter gives a source of randomness that is revealed over the course of the execution, in a manner that can be used to prevent the adversary knowing values ahead of time. Our common coin oracle produces a random string of some fixed length $\ell \in \mathbb{N}$ for each view $v \in \mathbb{N}$. Generally, $\ell \geq \lceil \log_2 n \rceil$, so that each random string of length ℓ can be used to determine a leader from the set of n processors. The oracle works as follows:

- Messages to and from the oracle are subject to the standard message delays of the model considered. For example, in the asynchronous model, such messages are always delivered but message delays are arbitrary.
- To query the common coin for view v, processor p_i sends the message v to the oracle.
- Upon receipt of the message v sent by a set P of $n - f$ different processors, the oracle samples a string σ of length ℓ uniformly at random and sends $\langle (v, \sigma) \rangle_O$ to all processors in P. Subsequently, upon receiving the message v from any processor p, the oracle sends $\langle (v, \sigma) \rangle_O$ to p.

Further comments. In this book we consider *idealised* versions of cryptographic primitives that can be realised in practice if certain cryptographic assumptions hold. In reality, the safe use of such primitives requires restricting to polynomial-time-bounded adversaries and accepting a negligible chance of error in any given execution. We take the approach of using an idealised model to avoid these complications. While one could recast all our arguments in the standard formal frameworks of cryptography, to do so would be a distraction here.

Chapter 13

Hotstuff

In Chapter 11, we observed that the communication complexity of both Tendermint and Streamlined PBFT is $\Theta(n^2)$ per view when threshold signatures are used, and $\Theta(n^3)$ otherwise. When n is large—as in many blockchain applications with hundreds or thousands of validators—this quadratic (or cubic) cost per view becomes a significant bottleneck. The natural question arises: can we do better?

In this chapter, we introduce Hotstuff [21], a protocol designed to achieve $O(n)$ communication complexity per view while maintaining strong optimistic responsiveness. The key insight is simple: if all communication within a view is routed through the leader and if threshold signatures are used to keep messages at constant size, then (if blocks are of constant-bounded size) each view requires only $O(n)$ total communication. However, achieving this requires careful protocol design to avoid the pitfalls that forced earlier protocols into quadratic complexity.

The block echoing problem. In the versions of Tendermint and PBFT described in previous chapters, we instructed processors to echo a leader's proposal to all other processors upon first voting for it. This ensured that all correct processors receive all finalised blocks, which is required for SMR as defined in Section 7.3. However, such 'all-to-all' communication immediately induces $\Omega(n^2)$ communication complexity in each view, regardless of whether threshold signatures are used.

To achieve $O(n)$ communication complexity per view, Hotstuff must forgo such block echoing. Instead, the protocol guarantees only that finalised blocks are *available*—meaning they can be retrieved by any processor that needs them—rather than immediately delivered to all. As discussed in Section 9.5.3, processors can use a "pull" mechanism to retrieve missing blocks from peers who possess them. This separation of concerns allows the core consensus protocol to achieve linear complexity per view, while block retrieval can be handled by a separate protocol optimised for that purpose.

Since Hotstuff does not explicitly ensure that all processors receive all finalised blocks, it does not directly solve SMR as defined in Section 7.3. Instead, it solves a related problem that we call *Extractable SMR*: roughly, this means solving SMR except that retrieving certain finalised blocks may require additional communication. In Section 13.1, we formally define this task.

Separating view change from view synchronisation. In Section 11.4, we distinguished between mechanisms for *view change* (ensuring leaders have enough information

to propose blocks that maintain Consistency) and mechanisms for *view synchronisation* (ensuring correct processors enter the same view within a short time after GST). The versions of Tendermint and Streamlined PBFT described in previous chapters achieved view synchronisation by having processors forward stage 2 QCs and TCs to all others upon entering a new view—an approach that induces $\Theta(n^2)$ complexity (or $\Theta(n^3)$ without threshold signatures).

Hotstuff treats view synchronisation as a black box, focusing instead on minimising the communication complexity of the view-change mechanism and the consensus logic within each view. In Chapter 14, we describe efficient view synchronisation protocols that can be combined with Hotstuff to achieve $O(n^2)$ overall single-transaction communication complexity. The quadratic bound arises because $O(f)$ views may be required after GST to finalise a transaction (when faulty processors are leaders), with each view contributing $O(n)$ communication.

Achieving strong optimistic responsiveness. Recall from Section 11.3 that a protocol is *strongly optimistically responsive* if latency after GST is $O(f_a\Delta + \delta)$, where f_a is the actual number of faulty processors and δ is the actual (unknown) network delay. In Section 11.4.2, we observed that Tendermint (even with the optimisation of Section 9.5.2) is only *weakly* optimistically responsive: it requires a grace period after GST before achieving good latency. Streamlined PBFT achieves strong optimistic responsiveness, but its view-change mechanism induces quadratic complexity.

Hotstuff achieves both strong optimistic responsiveness and linear complexity per view. It does so using a locking mechanism similar to Tendermint's, but modified so that leaders do not need to wait a fixed duration before proposing. The key insight is to use an extra round of voting: while Tendermint uses two stages of voting, Hotstuff uses three. This additional round allows the protocol to maintain Consistency and Liveness without requiring leaders to wait for locks to arrive.

Chapter outline. The remainder of this chapter is organised as follows. In Section 13.1, we formally define Extractable SMR. In Section 13.2, we explain the intuition behind Hotstuff's design, focusing on why three stages of voting are sufficient for strong optimistic responsiveness. In Section 13.3, we give a formal specification of the protocol. Finally, in Section 13.4, we prove that Hotstuff satisfies Consistency and Liveness for Extractable SMR, and that it is strongly optimistically responsive.

13.1 Extractable SMR

Before specifying Hotstuff, we must define the task it solves. As noted above, Hotstuff does not ensure that all correct processors immediately receive all finalised blocks. Instead, it ensures that any processor can *extract* (retrieve) any finalised block, given sufficient communication.

The setup. The setup is the same as for SMR (Section 7.3). We consider a set of n processors, of which at most f may be Byzantine. Processors receive transactions from the environment and must collectively agree on a sequence of finalised transactions. A *transaction* is just a signed message, belonging to a set of signed messages $\mathcal{T}$ that is known to the protocol (given as input to all processors). We allow that messages in

$\mathcal{T}$ may be signed by processors outside Π. If σ is a sequence of transactions, we write $\text{tr} \in \sigma$ to denote that the transaction tr belongs to the sequence σ.

The requirements. As for SMR, a protocol for *Extractable SMR* must specify a function $\mathcal{F}$ that maps any set of messages to a sequence of transactions. Let M^* be the set of all messages that are received by at least one (potentially Byzantine) processor during the execution. For any time-slot t, let $M(t)$ be the set of all messages that are received by at least one correct processor at a time-slot $\leq t$. We require the following conditions to hold for any sets of messages M_1 and M_2 and any transaction tr:

Consistency. If $M_1 \subseteq M_2 \subseteq M^*$, then $\mathcal{F}(M_1) \preceq \mathcal{F}(M_2)$.

Liveness. If correct p receives the transaction tr, there must exist t such that $\text{tr} \in \mathcal{F}(M(t))$.

As for SMR, consistency suffices to ensure that, for arbitrary $M_1, M_2 \subseteq M^*$, $\mathcal{F}(M_1)$ and $\mathcal{F}(M_2)$ are compatible. To see this, note that, by consistency, $\mathcal{F}(M_1) \preceq \mathcal{F}(M_1 \cup M_2)$ and $\mathcal{F}(M_2) \preceq \mathcal{F}(M_1 \cup M_2)$.

Relationship to SMR. Any protocol solving SMR also solves Extractable SMR. However, a protocol for Extractable SMR does not require correct processors to produce their own finalised log. The requirement is simply that the finalised log can be extracted from the messages received by all correct processors combined. A protocol for Extractable SMR can therefore be converted into a protocol for SMR by having correct processors echo certain received messages.

Variants of Extractable SMR. Just as for SMR (see Section 10.3), we can consider a version of Extractable SMR that assumes transactions are disseminated by a separate mempool protocol. *Extractable MSMR* is the same as Extractable SMR, except we also suppose that the following holds for some known bound ℓ: whenever a correct processor receives a transaction tr at some time-slot t, all correct processors receive tr from the environment by time-slot $\max\{t, \text{GST}\} + \ell$. Appendix D also defines 'extractable' analogues of BA and BB. A natural question that arises is as to whether the quadratic lower bound on communication complexity established by Dolev and Reischuk (see Section 5.3) still holds for these tasks. In Appendix D we establish a negative answer: Extractable BA can be solved with communication complexity $O(f \log f)$.

13.2 The intuition behind Hotstuff

In this section, we explain the key ideas behind Hotstuff. We focus on two questions: (1) why does Hotstuff use three stages of voting instead of two? and (2) how does Hotstuff achieve linear communication complexity per view?

13.2.1 Why three stages of voting?

Recall the locking mechanism in Tendermint (Section 9.2). Upon seeing a stage 1 QC for a block b, processors *lock* on b and will not vote for incompatible blocks in subsequent views unless they see a stage 1 QC for a higher view. This ensures Consistency: if b is

finalised (receives a stage 2 QC), then at least $f + 1$ correct processors are locked on b, preventing any incompatible block from receiving enough votes.

The problem with Tendermint's approach is that leaders must wait to ensure they have received the locks of *all* correct processors before proposing, otherwise some correct processors may not be able to vote for their proposal, threatening Liveness. In Section 9.4, leaders wait 2Δ after entering a view before proposing. This waiting is necessary because a correct processor p might lock on some block b just before entering a view, and might enter the view Δ time-slots after the leader, meaning that the leader only receives the lock 2Δ time-slots after entering the view.

Streamlined PBFT avoids this waiting by using *justifications*: leaders include view attestations from $n - f$ processors, proving they have gathered enough information to propose safely. However, these attestations are of size $\Theta(n)$ (even with threshold signatures), inducing quadratic communication complexity when the leader sends them to all processors.

Hotstuff's solution: three stages of voting. Hotstuff uses a third stage of voting to avoid both the waiting of Tendermint and the large justifications of PBFT. The key insight is as follows:

- In Tendermint, processors lock upon seeing a stage 1 QC and finalise upon seeing a stage 2 QC. This means that when a correct processor sets its lock, it may be the only processor to have seen the corresponding QC.
- In Hotstuff, processors lock upon seeing a stage 2 QC and finalise upon seeing a stage 3 QC. Any proposal by a leader must include a stage 1 QC for the parent. At the start of the view, the leader waits to hear from $n - f$ processors as to the greatest stage 1 QC they have received. If the greatest stage 1 QC amongst these $n - f$ messages is Q, which is a stage 1 QC for b, then the leader proposes a block b' with b as parent, and includes Q with the proposal. Correct processors vote for the proposal so long as Q.view is at least the view corresponding to their lock.

Note first that these modifications leave the proof of Consistency essentially unchanged. If a block b is finalised, then at least $f + 1$ correct processors lock on b, preventing any incompatible blocks from receiving a stage 1 QC in subsequent views.

However, the key point is that Liveness is now satisfied without the leader having to wait 2Δ. If correct p has locked on some block, then it may be the only processor to have seen the corresponding *stage 2* QC, but at least $n - f$ processors must have seen a *stage 1* QC for the block. By the standard quorum intersection argument, at least one of these processors must be correct and also be amongst the $n - f$ processors whose messages are received by the leader before forming their proposal. A correct leader is therefore guaranteed to produce a block that other correct processors can vote for.

13.2.2 Achieving linear complexity per view

The second key idea in Hotstuff is to route all communication through the leader. Instead of having all processors send votes to all others (which would induce $\Omega(n^2)$ complexity),

processors send votes only to the leader. The leader collects $n-f$ votes, forms a threshold signature (a 'threshold QC' of constant size), and broadcasts this to all processors.

This approach requires $O(n)$ messages per stage of voting:

- Each processor sends one vote to the leader: n messages.
- The leader broadcasts the threshold QC to all processors: n messages.

With three stages of voting, this gives $O(n)$ communication per view (assuming messages are of constant size, which requires threshold signatures).

The cost of view changes. When a view times out without producing a finalised block, processors must synchronise to enter the next view. Hotstuff treats this view synchronisation as a black box. When we formally specify Hotstuff in the next section, we will use a simple mechanism for view synchronisation that entails quadratic cost per view. Then, in Chapter 14, we will consider more efficient methods. The view-change mechanism itself requires only that the new leader receives stage 1 QCs from other processors. Since QCs are of constant-bounded size with threshold signatures, this can be achieved with $O(n)$ communication.

13.3 Hotstuff: the formal specification

We now give a formal specification of Hotstuff. The protocol uses threshold signatures as modelled in Section 12.2.1. We describe the protocol for the partially synchronous model without synchronised clocks.

In what follows, we suppose that, when a correct processor sends a message to 'all processors', it regards that message as immediately received by itself. We will describe the protocol as a protocol for Extractable MSMR, i.e., we assume that, whenever a correct processor receives a transaction tr at some time-slot t, all correct processors receive tr from the environment by time-slot $\max\{t, \mathrm{GST}\} + \ell$. We write $\emptyset$ to denote the empty set or empty sequence, and we let $\perp$ be some default distinguished value.

The pseudocode uses a number of message types, local variables, functions and procedures. Many of these are unchanged from Chapters 9 and 11, but for ease of reference we repeat them below.

The function $\texttt{lead}(v)$. The value $\texttt{lead}(v)$ specifies the *leader* for view v. To be concrete, we set $\texttt{lead}(v) := p_i$, where $i = v \bmod n$.

Votes. For $d \in \{1, 2, 3\}$, a stage d *vote* is a message of the form $(\mathrm{vote}, v, d, \tau)$ signed by some processor in Π, where $v \in \mathbb{N}_{\geq 0}$ specifies the view corresponding to the vote and τ is a finite binary string. If b is a block with hash τ, we also say the vote is a *vote for* b.

Stage 1, 2, and 3 QCs. For $d \in \{1, 2, 3\}$, a stage d QC is a set Q of $n - f$ votes, each of the form $(\mathrm{vote}, v, d, \tau)$ for the same values of v and τ, and each signed by a different processor in Π. We set $Q.\mathrm{view} = v$, $Q.\mathrm{block} = \tau$. A stage d threshold QC Q' can also be formed from Q using the threshold signature scheme, and then we also define $Q'.\mathrm{view} = v$, $Q'.\mathrm{block} = \tau$. Stage 1 QCs and threshold QCs are ordered by view and then by least hash. We also say Q (or Q') is a stage d (threshold) QC for τ and, if b is a

block with hash τ, we say Q (or Q') is a stage d (threshold) QC for b. Without explicit mention in the pseudocode, we suppose that whenever a correct processor first receives a QC, it immediately forms the corresponding threshold QC and regards the latter as received.

The lock. Each processor maintains a local variable `lock`, initially set to `lock` $= \emptyset$. The lock is always a stage 2 threshold QC (or $\emptyset$).

Time-out messages and TCs: a time-out message for view v is a message of the form $(\text{time-out}, v)$, signed by some processor in Π. A set of $n - f$ messages of this form, each signed by a different processor, is called a time-out certificate (TC) for view v. The threshold signature scheme can be used to form a *threshold TC* from a TC. Without explicit mention in the pseudocode, we suppose that whenever a correct processor first receives a TC, it immediately forms the corresponding threshold TC and regards the latter as received.

New-view messages. A *new-view message* for view v by p_i is a tuple of the form (v, i, Q), where Q is a stage 1 threshold QC.

Blocks. A *block* b is a message specified by four values:

- b.view: this is a value in $\mathbb{N}_{\geq 0}$ that specifies the view corresponding to b;
- b.Tr: a sequence of transactions in $\mathcal{T}$;[1]
- b.par: either $\perp$, or a hash value specifying the parent of b;
- b.QC: a stage 1 threshold QC for b.par if b.par $\neq \perp$.

A correct processor only regards a message as a block if it is of the form above. If b.view $= v$, we also refer to b as a 'view v block'. The *genesis block* is denoted b_{g} and satisfies b_{g}.view $= 0$, b_{g}.Tr $= \emptyset$, b_{g}.par $= \perp$, b_{g}.QC $= \perp$. We regard b_{g} as received by all correct processors at time-slot 0. Blocks are ordered by b.view and then by least hash. We stipulate that $\emptyset$ is a stage 1, 2, and 3 QC for b_{g}, and set $\emptyset$.view $= 0$: all processors therefore begin the protocol execution having received stage 1, 2, and 3 QCs for b_{g}.

The sequence of transactions specified by the ancestors of a block. Each block b specifies an extended sequence of transactions, denoted $b.\text{Tr}^*$, as follows: we concatenate the values $b'.\text{Tr}$ for all ancestors b' of b, removing any duplicate transactions.

The procedure MakeProposal. This procedure is executed by the leader p of view v to determine a new block. To execute the procedure, p:

1. Lets Q be the greatest stage 1 threshold QC it has received (including those in new-view messages).
2. Forms a sequence of distinct transactions T, containing all transactions received but not finalised by p;
3. Sets b.view $:= v$, b.Tr $:= T$, b.par $:= Q$.block and b.QC $= Q$.
4. Sends b to all processors.

[1] Recall that $\mathcal{T}$ is the set of possible transactions.

Valid proposals. At time-slot t, p_i regards a block b as a *valid proposal for view v* if all of the following conditions are satisfied: (i) b.view $= v$; (ii) b.par $\neq \perp$; (iii) $Q := b$.QC is a stage 1 threshold QC for b.par, and; (iv) Q.view $\geq$ `lock`.view.

Finalising blocks and transactions. We define $\mathcal{F}$ as follows. For any set of messages M, let b be the greatest block such that $M \cup \{b_g\}$ contains: (i) a stage 3 threshold QC for b, and (ii) all ancestors of b. Set $\mathcal{F}(M) := b.\mathrm{Tr}^*$.

The local variable `v`: this is a local variable, which p uses to record the present view. Initially, `v` $= 1$.

The local variables `1voted`, `2voted`, `3voted`, `1sent`, `2sent`, `3sent`, `sentnv`, and `proposed`: these record whether p_i has already sent stage 1, stage 2, and stage 3 votes in the present view, whether p_i (as leader) has already sent stage 1, stage 2, and stage 3 threshold QCs in the present view, whether p_i has already sent a new-view message for view v, and whether it has proposed a block for the view. They are all initially set to 0.

The timer. Each processor p has a local timer, denoted `Timer`, which it can set to 0 at any time, and which then automatically increments at each time-slot at which p is active. This means that the timer increments in 'real time' after GST. Initially, `Timer` $= 0$.

The pseudocode for Hotstuff (which only details what p_i should carry out *within each view*) is shown in Figure 13.1.

Hotstuff: the instructions for p_i.
At every time-slot t:

If `sentnv` $= 0$:
Let Q be the greatest stage 1 threshold QC that p_i has received;
Send $(\mathtt{v}, i, Q)$ to `lead`(`v`) and set `sentnv` $:= 1$; ▷ send new-view message

If $p_i =$ `lead`(`v`), `proposed` $= 0$, and p_i has received new-view messages for view `v` from at least $n - f$ processors:
MakeProposal; Set `proposed` $:= 1$; ▷ propose a new block if leader

If p_i has received a valid proposal b for view `v` from `lead`(`v`) and `1voted` $= 0$:
Send a signed stage 1 vote for b to `lead`(`v`); ▷ stage 1 vote
Set `1voted` $:= 1$;

For $d \in \{1, 2, 3\}$, if $p_i =$ `lead`(`v`), d`sent` $= 0$, and p_i has received Q which is a stage d QC for some view `v` block:
Form a threshold QC Q' from Q;
Send Q' to all processors; Set d`sent` $:= 1$; ▷ leader sends stage d threshold QC

For $d \in \{1, 2\}$, if d`voted` $= 1$, $(d+1)$`voted` $= 0$, and p_i has received Q which is a stage d threshold QC for some view `v` block b:
Send a stage $(d+1)$ vote for b to `lead`(`v`); ▷ stage 2 or 3 vote
Set $(d+1)$`voted` $:= 1$;
If $d = 2$, set `lock` $:= Q$; ▷ set lock

FIGURE 13.1: The pseudocode for Hotstuff (modulo view synchronisation).

The instructions for view synchronisation are shown in Figure 13.2.

View synchronisation: the instructions for p_i.
At every time-slot t:
If `Timer` $= 9\Delta$:
Send a time-out message for view `v` to all processors ▷ send time-out

If p_i has received Q which is a stage 3 threshold QC with Q.view $= v' \geq$ `v` or which is a threshold TC for view $v' \geq$ `v`:
Send Q to all processors;
Set `v` $:= v' + 1$, `1voted` $:= 0$, `2voted` $:= 0$, `3voted` $:= 0$;
Set `1sent` $:= 0$, `2sent` $:= 0$, `3sent` $:= 0$, `proposed` $:= 0$;
Set `sentnv` $:= 0$, `Timer` $:= 0$; ▷ enter higher view

FIGURE 13.2: The pseudocode for view synchronisation.

13.4 Hotstuff: the analysis

Throughout this section, we consider the partially synchronous model (without synchronised clocks), signatures and Byzantine faults, and we suppose that $f < n/3$.

13.4.1 Consistency and Liveness

First, we establish Consistency.

Lemma 13.1 (Consistency)**.** *Hotstuff satisfies Consistency.*

Proof. The proof is almost identical to that for Tendermint. For $d \in \{1, 2, 3\}$, let us say that block b 'receives a stage d QC' if $b = b_g$ or at least one correct processor receives a stage d QC or threshold QC for b. If $d \in \{1, 2\}$, any block b that receives a stage $d + 1$ QC must also receive a stage d QC, since no correct processor sends a stage $d + 1$ vote for b before receiving a stage d threshold QC for b.

To argue that the protocol satisfies Consistency, it suffices to show that no two incompatible blocks can receive stage 3 QCs. Towards a contradiction, suppose that there exists a least v such that:

- Some b with b.view $= v$ receives stage 1, 2, and 3 QCs, Q_1, Q_2, and Q_3 say.
- For some least $v' \geq v$, there exists b' such that b' is incompatible with b, with b'.view $= v'$ and b'.QC $= Q_0$ (say), and the block b' receives a stage 1 QC, Q_4 say.

If $v = v'$ then, by the quorum intersection argument of Section 9.1.2, some correct processor must have sent votes in both Q_1 and Q_4. This gives a contradiction since correct processors send at most one stage 1 vote in each view.

So, suppose $v' > v$. Then, by the same quorum intersection argument, some correct processor p must have sent votes in both Q_3 and Q_4. This gives the required contradiction, since p must set its local value `lock` to be a stage 2 threshold QC for b while in view v. However, our choice of (v, v') and the fact that b' is incompatible with b means that

Q_0.view $< v$, so that p would not regard the proposal b' as valid while in view v' and would not produce a vote in Q_4. □

Next we establish that correct processors progress through the views.

Lemma 13.2. *For each $v \geq 1$, there is some first time-slot, t_v say, at which a correct processor enters view v. Also, $t_{v'} > t_v$ if $v' > v$.*

Proof. The proof is a small modification of that in Section 9.4.1 and is left as an exercise (Exercise 13.1). □

Next, we establish the equivalent of Lemma 9.2.

Lemma 13.3 (Correct leaders finalise new blocks)**.** *For each $v \geq 1$, let t_v be as defined in the statement of Lemma 13.2. If $t_v \geq GST$ and $p = \texttt{lead}(v)$ is correct, then p proposes a view v block b and all correct processors receive a stage 3 threshold QC for b.*

Proof. Suppose the conditions in the statement of the lemma hold. For each $p_i \in \Pi$, let $\texttt{lock}_i$ denote the local variable `lock` as defined for p_i.

Towards a contradiction, suppose it is not the case that p proposes a view v block b and all correct processors receive a stage 3 QC for b by $t_v + 9\Delta$. If any correct processor received such a stage 3 QC by $t_v + 8\Delta$, the first to do so would send it to all other processors, which gives an immediate contradiction. No correct processor sends a time-out message for view v before $t_v + 9\Delta$, so no processor can receive a TC or threshold TC for view v before this time. Since the first correct processor to enter view v sends the corresponding certificate (a stage 3 threshold QC or threshold TC for view $v - 1$) to all processors at t_v, it follows that all correct processors are in view v for the entirety of the interval $[t_v + \Delta, t_v + 8\Delta]$. This holds because no correct processor can leave view v without receiving either a stage 3 threshold QC or a threshold TC for a view $\geq v$, neither of which can happen before $t_v + 8\Delta$.

Suppose p_i is correct and that, before entering view v, p_i most recently set $\texttt{lock}_i$ to some stage 2 threshold QC Q'. Since $n - f$ votes are required to form a stage 2 QC and correct processors do not send stage 2 votes before seeing a stage 1 threshold QC for the same block, it follows that at least $f + 1$ correct processors receive a stage 1 threshold QC for Q'.block before entering view v. By the standard quorum intersection argument, it follows that when p runs MakeProposal in view v, it has received a new-view message for view v from at least one correct processor that received a stage 1 threshold QC for Q'.block before entering view v. Note that p_i does not redefine $\texttt{lock}_i$ while in view v before sending a stage 1 vote: this is because $\texttt{lock}_i$ is only set when sending a stage 3 vote, which requires having already sent stage 1 and stage 2 votes for the view. From the definition of the procedure MakeProposal, it follows that p proposes a view v block b by $t_v + 2\Delta$ such that, for $Q = b$.QC, Q.view $\geq Q'$.view. Since our choice of p_i was arbitrary amongst correct processors, all correct processors send stage 1 votes for b to p by $t_v + 3\Delta$. All correct processors will then receive a stage 1 threshold QC for b by $t_v + 5\Delta$, and will send a stage 2 vote for b to p by this time-slot. All correct processors then receive a stage 2 threshold QC for b by $t_v + 7\Delta$, and will send a stage 3 vote for b to p by this time-slot. All correct processors will then receive a stage 3 threshold QC for b by $t_v + 9\Delta$, giving the required contradiction. □

Next, we establish Liveness.

Lemma 13.4 (Liveness)**.** *Hotstuff satisfies Liveness as defined for Extractable MSMR protocols.*

Proof. Suppose all correct processors receive the transaction tr by t_0. Suppose the first correct processor to enter view v does so at $t_1 \geq \max\{\text{GST}, t_0\}$ and that $p = \texttt{lead}(v)$ is correct. By Lemma 13.3, p proposes a block b and all correct processors receive a stage 3 threshold QC for b. In fact, the proof suffices to show that all correct processors receive such a QC before entering any later view. Any block receiving a stage 1 QC must have been received by at least one correct processor (who voted for it), and that processor must have verified that the block includes a valid stage 1 QC for its parent. It follows by induction that all ancestors of b receive stage 1 QCs and are received by correct processors. If t_2 is any time-slot by which all correct processors are in a view $> v$, it follows that $\text{tr} \in \mathcal{F}(M(t_2))$, as required. □

13.4.2 Strong optimistic responsiveness

Defining strong optimistic responsiveness for Extractable MSMR. In Section 11.3 we defined strong optimistic responsiveness for SMR and MSMR protocols. Since Hotstuff only satisfies the weaker form of Liveness defined for Extractable MSMR, we first need to consider how strong optimistic responsiveness should be defined for protocols satisfying this task. We modify the definition in the obvious way. Let δ be the (unknown) maximum message delay after GST and let $f_a \leq f$ be the actual (unknown) number of processors that do not behave correctly. For a transaction tr that is received by some correct processor, let t_1 be the first time-slot at which tr has been received by all correct processors (assumed to exist for MSMR) and let t_2 be the least time-slot such that $\text{tr} \in \mathcal{F}(M(t_2))$. Latency for tr is defined to be $\max\{0, t_2 - t_1\}$. A protocol for Extractable MSMR is *strongly optimistically responsive* if latency for any transaction first received by all correct processors after GST is $O(f_a\Delta + \delta)$.

Lemma 13.5. *Hotstuff is strongly optimistically responsive as a protocol for Extractable MSMR.*

Proof. Suppose tr is received by all correct processors by $t \geq \text{GST}$. Let v be the greatest view that any correct processor is in at t. Suppose first that the leader of view v is correct. Either view v will produce a new finalised block by $t + O(\delta)$, or a TC for the view will be received by all correct processors by $t + O(\delta)$. Either way, all correct processors will enter view $v + 1$ by $t + O(\delta)$. Let v' be the least view $\geq v + 1$ with correct leader. All correct processors will enter view v' by $t + O(\delta + f_a\Delta)$ and will finalise a block b for which $\text{tr} \in b.\text{Tr}^*$ in time $O(\delta)$ once in view v'. If the leader of view v is faulty, then all correct processors enter view v' by $t + O(f_a\Delta)$ and will again finalise a block b for which $\text{tr} \in b.\text{Tr}^*$ in time $O(\delta)$ once in view v'. □

13.4.3 Further considerations

We defer the formal analysis of message and communication complexity for Hotstuff until we have considered more efficient methods for view synchronisation in Chapter 14.

For now, we observe that the instructions for Hotstuff clearly incur linear communication complexity within each view if blocks are of constant-bounded size. However, the method of view synchronisation we have used in this chapter incurs quadratic cost per view.

Hotstuff achieves linear communication cost within each view by using the leader as a relay for votes and QCs. Since this increases the number of rounds of communication, it is natural to ask: does this actually increase the efficiency of the protocol? Doesn't routing through the leader create a bottleneck? Is latency decreased, or is the ability to handle high *throughput* (i.e., a large number of transactions per second) increased? We will revisit these issues in Chapter 24.

13.5 Exercises

Exercise 13.1. *[Andy: View progression proof.]*

Chapter 14

View synchronisation protocols

Chapter 15

Simplex

In previous chapters, we encountered several different view-change mechanisms: the locking mechanism of Tendermint (Chapter 9), the justification-based approach of PBFT (Chapter 11), and Hotstuff's hybrid approach (Chapter 13), which combines locking with new-view messages. In this chapter, we present Simplex [22], a protocol by Chan and Pass that introduces a distinctive view-change mechanism. Simplex is an elegant protocol that allows for fast view-progression. It is also of particular importance because a number of state-of-the-art protocols, including Kudzu [23] and Minimmit [24] (presented in Chapter 23), build on its design.

In Section 15.1, we describe the intuition behind the protocol. Section 15.2 gives the formal specification, and Section 15.3 formally verifies Consistency and Liveness.

15.1 Simplex: the intuition

Like Tendermint, PBFT, and Hotstuff, Simplex is a protocol for partial synchrony assuming $n \geq 3f + 1$. Like those protocols, it proceeds in sequential *views* $v = 1, 2, \ldots$, each with leader $\texttt{lead}(v)$.

Two rounds of voting. The protocol uses two rounds of voting per view. We begin by describing how each view progresses under good conditions, i.e., during synchrony with correct leaders:

- Upon entering view v, processors set a timer to expire in time 3Δ.
- When the leader enters the view, it immediately proposes a block (the precise mechanism for constructing the block will be described below).
- Upon receiving a valid proposal b, each processor disseminates a signed stage 1 vote for b.
- Upon receiving a stage 1 QC for b, processors move immediately to view $v+1$. At this point, if their timer has not yet expired (the significance of this condition will soon become clear), they disseminate a signed stage-2 vote for b.
- A block b is finalised when any descendant (possibly b) receives a stage 2 QC.

So far, a basic difference between Simplex and previous protocols we have considered is that processors proceed to the next view immediately upon seeing a stage 1 QC. However, we also have to consider what happens when no stage 1 QC is produced. We do this next.

Time-outs. If a processor is still in view v when its timer expires (meaning that it has not received a stage 1 QC for a view v block), then it disseminates a signed time-out message for view v. A set of $n - f$ time-out messages for view v, each signed by a different processor, is called a time-out certificate (TC) for view v. We now stipulate that processors move from view v to view $v + 1$ under two possible conditions:

(i) They receive a stage 1 QC for a view v block, or;

(ii) They receive a TC for view v.

The critical design constraint is that each correct processor *either* sends a stage-2 vote *or* a time-out message for view v, but never both. It follows by the standard quorum intersection argument ($(n - f) + (n - f) - n \geq f + 1$ when $n \geq 3f + 1$) that:

(a) A stage 2 QC and a TC for the same view cannot coexist.

(b) Two distinct blocks cannot both receive a stage 1 (or stage 2) QC in the same view.

To verify that the protocol functions correctly, it remains to establish that correct processors progress through all views, that it satisfies Consistency, and that it satisfies Liveness. We treat these in order below.

Progression through views. We stipulate that correct processors should disseminate stage-1 QCs and TCs upon first receipt. From this, view progression for correct processors follows straightforwardly. If any correct processor leaves view v upon receiving a stage 1 QC, then all correct processors receive it and also leave the view. Otherwise, all correct processors eventually disseminate time-out messages for the view, meaning that all correct processors receive a TC. Recall that δ is the (unknown) least upper bound on message delays after GST. It also follows that if a correct processor enters view $v+1$ at $t \geq$ GST, then all correct processors enter the view by $t + \delta$.

Consistency. To argue Consistency, we must first stipulate how leaders form blocks.[1] Upon entering view v, the leader p will let $v' < v$ be the greatest view such that it has received Q which is a stage 1 QC for some view v' block, b' say. The fact that it has entered view v means that p must have received TCs for all views in the open interval (v', v), and will have disseminated all such TCs as well as Q. Then p will propose a block b with b' as parent. Other processors will consider b a *valid proposal*, and will vote for it, so long as they have received:

(i) A stage 1 QC for b', and;

[1] The original Simplex paper gives a theoretically minded presentation, in which the leader of each view sends the entire blockchain in each view, i.e., they form a new block and send that block *and all ancestors of the block* to other processors. Here, we modify the protocol to give a more practical version.

(ii) TCs for all views in (v', v).

To see that this ensures Consistency, suppose b receives a stage 2 QC and there exists some least view $v'' \geq v$ in which a block b'' incompatible with b is proposed and receives a stage 1 QC. Then we reach an immediate contradiction: by the standard quorum intersection argument $v'' \neq v$, by our choice of v'' the parent of b'' must be for a view $< v$, and it follows from the definition of a valid proposal above that no correct processor would vote for b'' without receiving a TC for view v. By (a) above, such a TC cannot exist. (We will give a formal proof in Section 15.3.)

Liveness. The main lemma required to establish Liveness is that correct leaders finalise new blocks after GST. In fact, this now follows straightforwardly from the observations already made above. Suppose the first correct processor to enter view v does so at $t \geq$ GST. We observed above that this means all correct processors enter the view by $t + \delta$. If the leader of the view is correct, they will therefore propose a block b for view v by $t + \delta$. Suppose b has parent b' for view v'. Since the leader has entered view v, it must have received TCs for all views in (v', v), as well as a stage 1 QC for b'; it disseminates all of these upon first receipt. All correct processors therefore receive these, together with the proposal b, by $t + 2\delta$. They therefore consider the proposal valid and disseminate stage 1 votes for b by $t + 2\delta$, receiving a stage 1 QC for b by $t + 3\delta$. Since $\delta \leq \Delta$, this occurs before any correct processor's timer expires (the earliest a timer can expire is $t + 3\Delta \geq t + 3\delta$). All correct processors therefore send stage 2 votes upon entering view $v + 1$, and receive a stage 2 QC for b by $t + 4\delta$.

15.2 Simplex: the formal specification

For the sake of simplicity, we suppose (without explicit mention in the pseudocode) that correct processors disseminate transactions to all upon first receipt. The pseudocode uses a number of message types, local variables, functions and procedures. Below, we describe only those that differ from the treatment of Tendermint in Section 9.2.3.

Blocks. A *block* b is a message specified by three values:

- b.view: this is a value in $\mathbb{N}_{\geq 0}$ that specifies the view corresponding to b;
- b.Tr: a sequence of transactions in $\mathcal{T}$;
- b.par: either $\perp$, or a hash value specifying the parent of b;

The *genesis block* is denoted b_g and satisfies b_g.view $= 0$, b_g.Tr $= \emptyset$, b_g.par $= \perp$. We regard b_g as received by all correct processors at time-slot 0.

QCs. As in Section 9.2.3, we let $\emptyset$ be a stage 1 QC for b_g, and set $\emptyset$.block $= H(b_g)$ and $\emptyset$.view $= 0$. Blocks and QCs are ordered by view and then by least hash.

The local variable `v`: initially set to 1, this records the present view.

The local variables `1voted`(v), `timedout`(v), and `proposed`(v): initially set to 0 for all v, these record whether p_i has already sent a stage 1 vote, whether it has timed-out, and whether it has made a proposal in view v.

The timer: each processor p_i has a local timer, denoted `Timer`, which it can set to 0 at any time, and which then automatically increments at each time-slot at which p is active. This means that the timer increments in 'real time' after GST. Initially, `Timer` $= 0$.

Time-out messages and TCs: a time-out message for view v is a message of the form $(\text{time-out}, v)$, signed by some processor in Π. A set of $n - f$ messages of this form, each signed by a different processor, is called a time-out certificate (TC) for view v.

Valid proposals. At time-slot t, p_i regards a block b as a *valid proposal for view* v if all of the following conditions are satisfied: (i) b.view $= v$; (ii) b.par $\neq \perp$; (iii) p_i has received Q, which is a stage 1 QC for b.par, with Q.view $= v' < v$ (say); (iv) p_i has received TCs for all views in (v', v).

The procedure MakeProposal. This procedure is executed by the leader p of view v to determine a new block. To execute the procedure, p:

1. Lets Q be the greatest stage 1 QC it has received.
2. Forms a sequence of distinct transactions T, containing all transactions received but not finalised by p;
3. Sets b.view $:= v$, b.Tr $:= T$, b.par $:= Q$.block.
4. Disseminates b.

The instructions are shown in Figure 15.1.

Simplex: the instructions for p_i.
At every time-slot t:
  Disseminate all newly received QCs and TCs;

  If $p_i =$ `lead(v)` and `proposed(v)` $= 0$:
    MakeProposal; Set `proposed(v)` $:= 1$; ▷ Propose a new block if leader

  If p_i has received a valid proposal b for view `v` from `lead(v)` and `1voted(v)` $= 0$:
    Send b and a signed stage 1 vote for b to all processors; ▷ stage 1 vote
    Set `1voted(v)` $:= 1$;

  If p_i has received Q which is a stage 1 QC for a view `v` block b:
    If `timedout(v)` $= 0$:
      Send a signed stage 2 vote for b to all processors; ▷ stage 2 vote
    Set `v` $:=$ `v` $+ 1$, `Timer` $:= 0$; ▷ go to next view

  If `Timer` $= 3\Delta$:
    Disseminate a time-out message for view `v`; ▷ time-out
    Set `timedout(v)` $:= 1$;

  If p_i has received a TC for view `v`:
    Set `v` $:=$ `v` $+ 1$, `Timer` $:= 0$; ▷ go to next view

FIGURE 15.1: The pseudocode for Simplex.

15.3 Simplex: the formal verification

In this section, we establish Consistency and Liveness. For $d \in \{1, 2\}$, we say a block b *receives a stage d QC* if any processor receives such a QC (and similarly for TCs).

Lemma 15.1. *Suppose b is a block with $b.view = v$. Then:*

(i) If b receives a stage 1 QC, no other view v block receives a stage 1 QC.

(ii) If b receives a stage 2 QC, it receives a stage 1 QC.

(iii) If b receives a stage 2 QC, view v does not receive a TC.

Proof. Claims (i) and (iii) follow from the standard quorum intersection argument, and the fact that no correct processor disseminates a stage 1 vote for more than one block, or sends both a time-out message for view v and also a stage 2 vote for some view v block. Claim (ii) follows since no correct processor disseminates a stage 2 vote for b before receiving a stage 1 QC for b. □

Lemma 15.2 (Consistency)**.** *Simplex satisfies Consistency.*

Proof. Towards a contradiction, suppose some block b_1 with $b_1.\text{view} = v_1$ receives a stage 2 QC, and that for some *least* $v_2 \geq v_1$ some block b_2 satisfies:

1. $b_2.\text{view} = v_2$;
2. b_1 is not an ancestor of b_2, and;
3. b_2 receives a stage 1 QC.

From Lemma 15.1, it follows that $v_2 > v_1$. According to clause (iii) from the definition of a valid proposal, correct processors will not disseminate a stage 1 vote for b_2 until receiving a stage 1 QC for the parent, b_0 say. We conclude that b_0 receives a stage 1 QC. By our choice of v_2, it follows that $b_0.\text{view} < v_1$. This gives a contradiction, because by clause (iv) from the definition of a valid proposal, correct processors would not disseminate a stage 1 vote for b_2 without receiving a TC for view v_1. By Lemma 15.1, such a TC cannot exist. This gives the required contradiction. □

To prove Liveness, we first establish that correct processors progress through all views.

Lemma 15.3 (View progression)**.** *Every correct processor enters every view $v \in \mathbb{N}_{\geq 1}$. Also, if p is correct and enters view v at $t \geq GST$, then all correct processors enter view v by $t + \Delta$.*

Proof. The proof works exactly as already described in Section 15.1. □

Next, we establish that correct leaders finalise new blocks after GST.

Lemma 15.4 (Correct leaders finalise new blocks)**.** *If the first correct processor to enter view v does so at $t \geq GST$, and $p = \texttt{lead}(v)$ is correct, then p proposes a view v block b and all correct processors receive a stage 2 QC for b by $t + 4\delta$.*

Proof. The proof proceeds just as described in Section 15.1. By Lemma 15.3, all correct processors enter view v by $t + \delta$. Since p is correct, it proposes a block b for view v by $t+\delta$. Since p has entered view v, it must have received TCs for all views in (v', v) (where v' is the view of b's parent) as well as a stage 1 QC for the parent; it disseminates all of these upon first receipt. All correct processors therefore receive these, together with the proposal b, by $t + 2\delta$. They consider the proposal valid and disseminate stage 1 votes for b by $t + 2\delta$, receiving a stage 1 QC for b by $t + 3\delta$. Since $\delta \leq \Delta$, this occurs before any correct processor's timer expires (the earliest a timer can expire is $t + 3\Delta \geq t + 3\delta$). All correct processors therefore send stage 2 votes upon entering view $v + 1$, and receive a stage 2 QC for b by $t + 4\delta$. □

Next, we prove that correct processors eventually receive all ancestors of any block finalised by a correct processor.

Lemma 15.5 (Processors receive necessary blocks)**.** *Suppose p is correct and finalises b. Then all correct processors eventually receive all ancestors of b.*

Proof. The proof is essentially the same as for Lemma 9.3. □

Finally, we establish Liveness.

Lemma 15.6 (Liveness)**.** *Simplex satisfies Liveness.*

Proof. The proof is essentially the same as for Lemma 9.4. Suppose correct p_i receives the transaction tr at t_0. All correct processors receive tr by $t_1 = \max\{t_0, \text{GST}\} + \Delta$. Let v be a view that begins at $t_2 \geq t_1$, and with correct leader p. By Lemma 15.4, p proposes a block b that is finalised by all correct processors. By Consistency, and by the definition of the procedure MakeProposal, $\text{tr} \in b'.\text{Tr}$ for some ancestor b' of b, i.e., $\text{tr} \in b.\text{Tr}^*$. By Lemma 15.5, all correct processors receive all ancestors of b. So, tr is finalised by all correct processors. □

Chapter 16

Protocols for asynchrony

We saw in Chapter 8 that deterministic protocols cannot solve BA (or BB) in the asynchronous model. In this chapter, we develop positive results that reveal what can be achieved despite this fundamental limitation.

We begin by showing that deterministic protocols can solve a task called *Reliable Broadcast* (RB). Reliable Broadcast is similar to BB, but relaxes the requirement for Termination: correct processors must terminate and agree on the broadcaster's value when the broadcaster is correct, but need not terminate when the broadcaster is faulty. This relaxation sidesteps the FLP impossibility while still providing a useful primitive.

We then turn to probabilistic protocols, which offer another avenue for circumventing the FLP impossibility. We describe a simple protocol that solves SMR in the asynchronous model using a common coin.

16.1 Reliable Broadcast

We proved in Chapter 8 that no protocol can solve BB in the partially synchronous model, when $f \geq 1$ and $n \geq 2$. The basic reason is that one cannot distinguish between a crashed broadcaster and unbounded periods of asynchrony prior to GST. If processors do not hear from the broadcaster, they must eventually output. This means that if messages from the broadcaster are delayed for too long, then correct processors will output without hearing the broadcaster's value (violating Validity). A natural way to make the task easier is therefore only to require processors to output in the case that the broadcaster is correct. This is precisely what is required by Reliable Broadcast (RB). The problem is defined as follows:

- We consider a set of n processors, of which at most f display Byzantine faults.
- One processor is designated the 'broadcaster'. All processors are told the name of the broadcaster.
- The broadcaster is given an input in some set V. The set V is told to all processors.
- The protocol must satisfy the following conditions:

- **Validity**. If the broadcaster is correct and has input v, then all correct processors must output v.
- **Agreement**. No two correct processors can give different outputs.
- **Totality**. If any correct processor outputs, then all correct processors output a value in V.

Comparing this definition with BB (Chapter 4), the difference is that the Termination condition has been replaced by Totality. For BB, all correct processors must terminate regardless of whether the broadcaster is correct. For RB, termination is guaranteed only when the broadcaster behaves in a way that causes at least one correct processor to output. Validity requires correct processors to terminate when the broadcaster is correct. However, when the broadcaster is Byzantine and sends no messages, Validity and Totality allow that correct processors may wait indefinitely with outputting. The Totality condition also ensures that a Byzantine broadcaster cannot cause only some correct processors to output while others wait forever: either all correct processors output, or none do.

16.2 Bracha's Broadcast

In this section, we describe a deterministic protocol called Bracha's Broadcast [], which solves RB without signatures when $f < n/3$. Exercise **??** asks you to modify the proof of Theorem 8.5 to show that RB is not solvable in the partially synchronous (or asynchronous) model when $f \geq n/3$.

16.2.1 The specification

The idea behind Bracha's Broadcast is quite simple:

- **Agreement**. First, we have the broadcaster send their value to all processors. Then correct processors 'echo' the value they receive, sending it to all others. By the standard quorum intersection argument (Section 9.1.2), it is not possible for two different values to be echoed by $n - f$ processors. So, if we have processors 'vote' for a value v upon receiving v as an echo from $n - f$ processors, correct processors cannot vote for different values. So far, we've established Agreement (for votes).
- **Validity**. We must also satisfy Totality and Validity. If we had processors just output their vote, the problem is that some correct processors might receive $n - f$ echoes of the same value, while others do not. When the broadcaster is correct with value v, however, it's easy to see that all correct processors will echo and then vote for v, and so will receive (at least) $n - f$ votes for v. We can therefore have correct processors output v upon receiving $n - f$ votes for v and Validity will be satisfied.
- **Totality**. It remains to satisfy Totality. To do so, we note that, if any correct processor receives $n - f$ votes for v, then all correct processors receive at least $n - 2f \geq f + 1$ votes for v. Since correct processors don't vote for different values,

there can only be one value that receives $f + 1$ votes. We can therefore add the instruction that processors should vote for v (if they haven't already) upon receiving $f + 1$ votes for v. This will not violate Agreement for votes, and means that if any correct processor outputs v, all correct processors vote for v and give that value as output.

The pseudocode appears in Figure 16.1. When a processor p votes for v upon receiving $n - f$ echoes of v, we say p votes for v "by reason 1". When a processor p votes for v upon receiving $f + 1$ votes for v, we say p votes for v "by reason 2".

Bracha's Broadcast: the instructions for p_i.
At time-slot 0:
  Set `echoed` := false, `voted` := false;
  If p_i is the broadcaster, send input to all processors.

At every time-slot t:
  If p_i has received a value v from the broadcaster and `echoed` = false:
    Send (echo, v) to all processors; Set `echoed` := true; ▷ echo

  If there exists v s.t. p_i has received (echo, v) from $n - f$ processors and `voted` = false:
    Send (vote, v) to all processors; Set `voted` := true; ▷ vote (reason 1)

  If there exists v s.t. p_i has received (vote, v) from $f + 1$ processors and `voted` = false:
    Send (vote, v) to all processors; Set `voted` := true; ▷ vote (reason 2)

  If there exists v s.t. p_i has received (vote, v) from $n - f$ processors:
    Output v ▷ output

FIGURE 16.1: The pseudocode for Bracha's Broadcast.

16.2.2 The verification

To verify the protocol, we elaborate on the arguments from Section 16.2.1.

Lemma 16.1. *No two correct processors vote for different values.*

Proof. We first show that at most one value can receive $n - f$ echoes. Towards a contradiction, suppose distinct values v and v' both receive $n-f$ echoes. By the standard quorum intersection argument (Section 9.1.2), the sets of processors sending these echoes must intersect in at least $n - 2f \geq f + 1$ processors, so at least one correct processor must have echoed both v and v'. This contradicts the fact that correct processors echo at most once.

Now suppose there is some first time-slot at which a correct processor votes for a value, v say. This vote must be triggered by reason 1 (receiving $n - f$ echoes of v). Towards a contradiction, suppose there is some first time-slot t at which a correct processor votes for a value v' different from v. Since at most one value receives $n - f$ echoes, this vote must be triggered by reason 2 (receiving $f + 1$ votes for v'). But then some correct processor must have voted for v' before t, contradicting our choice of t. □

Lemma 16.2. *Validity is satisfied.*

Proof. Suppose the broadcaster is correct with value v. Then all correct processors receive v from the broadcaster and echo v. Each correct processor therefore receives $n - f$ echoes of v and at most f echoes of any other value. This means no correct processor can vote for a different value v' by reason 1, and no correct processor can vote for any value v' by reason 2 until some correct processor does so by reason 1. It follows that all correct processors vote for v, receive $n - f$ votes for v, and so output v. □

Lemma 16.3. *Totality and Agreement are satisfied.*

Proof. Suppose some correct processor p outputs a value v. Then p receives $n - f$ votes for v, so all correct processors receive $n - 2f \geq f + 1$ votes for v. All correct processors therefore vote for v (either by reason 1 or 2). So, all correct processors receive $n - f$ votes for v and output that value. □

This completes the verification that Bracha's Broadcast solves Reliable Broadcast when $f < n/3$.

16.3 Solving SMR in asynchrony using a common coin

In this section, we describe a simple protocol (similar to []) for solving SMR in the asynchronous model using a common coin.[1] First, in Section 16.3.1, we describe the intuition behind the protocol. Section 16.3.2 gives the formal specification, and Section 16.3.3 verifies Consistency and Liveness. Finally, Section 16.3.4 discusses some more efficient approaches from the literature.

16.3.1 The intuition

First, we consider how the obstacle identified by the FLP impossibility theorem (Chapter 8) manifests for leader-based SMR protocols. Then we show how to circumvent this obstacle using a common coin.

The problem. If we try implementing a leader-based SMR protocol such as Streamlined PBFT (Chapter 11) or Hotstuff (Chapter 13) in the asynchronous model, the basic difficulty is that messages from the leader can always be arbitrarily delayed. If we wait as long as necessary until we hear from the leader, a faulty leader can cause us to wait forever. On the other hand, if we time out after some bounded period, it may turn out that the leader was actually correct. The same situation may then repeat in all subsequent views.

A simple solution. One simple, although rather inefficient, approach is to select the leader using the common coin *at the end of the view*. Rather than having a single leader propose in each view, we have all processors act as potential leaders. Each processor proposes a block, and other processors vote on each of these proposals. Eventually, proposals from at least $n - f$ processors will receive sufficient votes for finalisation

[1] See Section 12.2.3 for our formalisation of a common coin.

(e.g., a stage-2 QC for Streamlined PBFT, or a stage-3 QC for Hotstuff). Once correct processors see that this threshold has been reached, they query the common coin to randomly select a leader p for the view. If p's proposal has received sufficient votes for finalisation, that proposal is appended to the finalised log.

The key insight is that the adversary cannot know which processor will be selected as leader until after the proposals have been formed. By that point, at least $n - 2f$ correct processors will have proposals with sufficient votes, ensuring that the selected leader is correct (and has a valid proposal) with probability at least $(n - 2f)/n > 1/3$.

16.3.2 The formal specification

We describe a protocol that is similar to Hotstuff, in the sense that it uses a similar locking mechanism and three stages of voting, but with a number of differences. First, every processor now proposes a block for each view. Since we are not initially concerned with matters of efficiency, we also have processors send votes directly to all processors, rather than having messages relayed via leaders, and also have processors forward blocks they vote for (to produce a protocol for SMR rather than Extractable SMR). At the end of each view, the common coin oracle is used to select the leader, and all other proposals are discarded.

In what follows, we use the term 'disseminate' to mean 'send to all processors'. [Andy: Use this earlier in the book.] As always, we suppose that, when a correct processor is instructed to send a message to itself, it regards that message as immediately received. We write $\emptyset$ to denote the empty set or empty sequence, and we let $\perp$ be some default distinguished value. We write $x\downarrow$ to indicate that the variable x is defined.

The pseudocode uses a number of message types, local variables, functions and procedures. Many of these are unchanged from Chapter 13, and we assume the reader is familiar with that chapter in what follows.

The local variable $\texttt{lead}(v)$. This value is initially undefined for all $v \in \mathbb{N}_{\geq 1}$, while $\texttt{lead}(0)$ is initially set to $\perp$.

Votes and stage 1, 2, and 3 QCs, threshold QCs, and newview messages. These are defined as in Chapter 13.

The lock. As in Chapter 13, each processor maintains a local variable `lock`, initially set to be a stage 2 threshold QC for the genesis block. [Andy: correct Hotstuff.] The lock is always a stage 2 threshold QC.

Blocks. Blocks other than the genesis block are defined as in Chapter 13, except that each block b now contains a fifth value b.auth, which specifies the block creator. If Q is a stage d threshold QC for b, then we also set Q.auth := b.auth, Q.view := b.view, and Q.block := $H(b)$. Blocks and threshold QCs are ordered by view and then by least hash. The genesis block b_g is defined as in Chapter 13, except that it contains the extra value b_g.auth = $\perp$. The genesis block is regarded as received by all correct processors at time-slot 0. We stipulate that $\emptyset$ is a stage 1, 2, and 3 QC for b_g: all processors begin the protocol execution having received stage 1, 2, and 3 QCs and threshold QCs for b_g.

The local variable `v`: As in Chapter 13, this is a local variable, which p uses to record the present view. Initially, `v` = 1.

The procedure MakeProposal. To execute the procedure, processor p_i:

1. Lets Q be the greatest stage 1 threshold QC it has received (including those in new-view messages) such that $Q.\text{view} = v'$ for some $v' < \texttt{v}$ and $\texttt{lead}(v') \downarrow = Q.\text{auth}$.
2. Forms a sequence of distinct transactions T, containing all transactions received but not finalised by p_i; [Andy: Check Hotstuff.]
3. Sets $b.\text{view} := \texttt{v}$, $b.\text{Tr} := T$, $b.\text{par} := Q.\text{block}$, $b.\text{QC} := Q$, and $b.\text{auth} := i$.
4. Disseminates b.

Valid proposals. At time-slot t, p_i regards a block b received from p_j as a *valid proposal for view* v if all of the following conditions are satisfied: (i) $b.\text{view} = v$; (ii) $b.\text{par} \neq \perp$; (iii) for some $v' < v$, $Q := b.\text{QC}$ is a stage 1 threshold QC for a view v' block b' with $H(b') = b.\text{par}$, and $\texttt{lead}(v') \downarrow = b'.\text{auth}$; (iv) $Q.\text{view} \geq \texttt{lock}.\text{view}$, and (v) $b.\text{auth} = j$.

Finalising blocks and transactions. We define $\mathcal{F}$ as follows. For any set of messages M, let b be the greatest block such that $M \cup \{b_g\}$ contains: (i) a stage 3 threshold QC for b; (ii) all ancestors of b; (iii) stage 1 threshold QCs for all ancestors of b, and; (iv) a message $\langle (b.\text{view}, b.\text{auth}) \rangle_O$ (signed by the common coin oracle). Set $\mathcal{F}(M) := b.\text{Tr}^*$ (where $b.\text{Tr}^*$ is defined as in Chapter 13).

The local variables $\texttt{sentnv}$, $\texttt{proposed}$, and $(1,j)\texttt{voted}$, $(2,j)\texttt{voted}$, $(3,j)\texttt{voted}$, for each $j \in [0, n-1]$: these record whether p_i has already sent a new-view message for view v, whether it has proposed a block for the view, and whether p_i has already sent stage 1, stage 2, and stage 3 votes for p_j's proposal in the present view. They are all initially set to 0.

The pseudocode is shown in Figure 16.2.

16.3.3 The verification

16.3.4 Further discussion

[Andy: In Chapter 3, we said all our positive results for static adversaries carry over to adaptive adversaries. Better comment on that here.]

16.4 Exercises

Exercise 16.1. *[Andy: Modify proof of 8.5 for RB]*

AsyncSMR: the instructions for p_i.
At every time-slot t:

If $\texttt{sentnv} = 0$:
Let Q be the greatest stage 1 threshold QC that p_i has received s.t. $Q.\text{view} < \texttt{v}$ and $\texttt{lead}(Q.\text{view}) \downarrow = Q.\text{auth}$;
Disseminate $(\texttt{v}, i, Q)$ and set $\texttt{sentnv} := 1$; ▷ disseminate new-view message

If $\texttt{proposed} = 0$, and p_i has received new-view messages for view $\texttt{v}$ from at least $n - f$ processors:
MakeProposal; Set $\texttt{proposed} := 1$; ▷ propose a new block

For each $j \in [0, n-1]$, if p_i has received a valid proposal b for view $\texttt{v}$ from p_j and $(1, j)\texttt{voted} = 0$:
Disseminate b and a signed stage 1 vote for b; ▷ stage 1 vote
Set $(1, j)\texttt{voted} := 1$;

For each $j \in [0, n-1]$ and $d \in \{1, 2\}$, if $(d, j)\texttt{voted} = 1$, $(d+1, j)\texttt{voted} = 0$, and p_i has received Q which is a stage d threshold QC for some view $\texttt{v}$ block b with $b.\text{auth} = j$:
Disseminate a stage $(d+1)$ vote for b; ▷ stage 2 or 3 vote
Set $(d+1, j)\texttt{voted} := 1$;

Let I be the set of $j \in [0, n-1]$ such that p_i has received stage 1, 2 and 3 threshold QCs for some view $\texttt{v}$ block b with $b.\text{auth} = j$;
If $|I| \geq n - f$, send $\texttt{v}$ to the common coin oracle; ▷ query oracle

If there exists k, s.t. p_i has received a message $\langle(\texttt{v}, k)\rangle_O$ from the common coin oracle:
Set $\texttt{lead}(\texttt{v}) := k$;
If p_i has received a stage 2 threshold QC Q with $Q.\text{view} = \texttt{v}$ and $Q.\text{auth} = k$:
Set $\texttt{lock} := Q$;
Set $\texttt{sentnv} := 0$, $\texttt{proposed} := 0$;
For each $j \in [0, n-1]$ and $d \in \{1, 2, 3\}$, set $(d, j)\texttt{voted} := 0$;
Set $\texttt{v} := \texttt{v} + 1$; ▷ go to next view

FIGURE 16.2: Pseudocode for AsyncSMR.

Chapter 17

Combining erasure codes with Reliable Broadcast

Chapter 18

Payment systems

Chapter 19

DAG-based protocols

[Andy: Intro. Using DAG for payment systems (shows payment systems are easier than consensus). Then a ps protocol (easiest possible).]

Chapter 20

Accountability

In previous chapters, we have designed SMR protocols that maintain Consistency so long as fewer than $n/3$ processors are Byzantine. But what happens when this threshold is exceeded? If more than $n/3$ processors behave dishonestly, Consistency may be violated: correct processors might finalise incompatible sequences of transactions. While we cannot prevent this from happening (as shown by the impossibility results of Chapter 8), we can ask a different question: when Consistency is violated, is it possible to identify which processors misbehaved?

This question is of considerable practical importance. In many real-world systems, processors are operated by known entities with reputations or financial stakes. If a protocol can produce cryptographic proof that specific processors acted dishonestly whenever Consistency is violated, this creates a powerful deterrent against misbehaviour. In the context of blockchain systems, this idea is often formalised through *slashing*: processors deposit collateral that is forfeited if evidence of misbehaviour is produced. The effectiveness of such mechanisms depends on the protocol's ability to generate irrefutable evidence of fault.

Roughly, a protocol satisfies *accountability* if, whenever Consistency is violated, it is possible to produce cryptographic proofs that certain processors deviated from the protocol. These proofs must be based solely on signed messages sent during the execution, so that the evidence is independently verifiable by any outside observer. We will make this precise in Section 20.1.

Not all protocols satisfy accountability. Whether a protocol does so depends on its structure, and in particular on its view-change mechanism. In this chapter, we formally define accountability and then examine two protocols we have already studied: Tendermint (Chapter 9) and Streamlined PBFT (Chapter 11). We show that both protocols satisfy accountability, though the proofs differ in interesting ways that reflect the different approaches these protocols take to view changes. We conclude with a discussion of an important practical consideration: to actually enact slashing, processors must reach consensus on the set of guilty parties, which requires techniques beyond the accountability mechanism itself.

20.1 Defining accountability

We say an *execution* has a *consistency violation* if there exist two incompatible sequences of transactions which are both finalised by correct processors. However, in Chapter 21, it will also be useful to consider *sets of messages* as having a *number* of consistency violations. To prepare for that discussion, we consider the more general definition now.

The number of consistency violations. For any given execution, we let M_c be the set of messages that are received by at least one correct processor, and let $M_c(t)$ be the set of messages that are received by at least one correct processor by time-slot t. [Andy: Check consistency with earlier notation.] Recall from Chapter 7 that a protocol $\mathcal{P}$ for SMR must specify a function $\mathcal{F}$. When $\mathcal{F}$ is clear from context, we say the set of messages M *has* r *consistency violations* if there exist $M_0 \subset M_1 \subset \cdots \subset M_r \subseteq M$ such that, for each $s \in \{0, 1, \ldots, r-1\}$, $\mathcal{F}(M_s) \not\preceq \mathcal{F}(M_{s+1})$. We also say M *has a consistency violation* (w.r.t. $\mathcal{F}$) if it has at least one consistency violation. An execution has r consistency violations if M_c has r consistency violations. Intuitively, each step in the chain witnesses a new incompatibility: the messages in $M_{s+1} \setminus M_s$ cause the finalised sequence to change in a way that is incompatible with what was previously finalised. The usefulness of counting consistency violations in this way will become apparent in Chapter 21.

Accountable protocols (informal discussion). Informally, a protocol is *accountable* if it produces *proofs of guilt* for some faulty processors in the event of a consistency violation. In the partially synchronous model, we cannot generally require proofs of guilt for a fraction $\lambda > 1/3$ of processors, since consistency violations may occur when less than a fraction λ of processors are faulty. On the other hand, all standard protocols that provide accountability produce proofs of guilt for at least $1/3$ of the processors in the event of a consistency violation. Roughly, this is because a consistency violation requires conflicting quorum certificates, and by the standard quorum intersection argument (Section 9.1.2), two quorums of size $n-f$ intersect in at least $n-2f \geq n/3$ processors (when $f < n/3$). Each processor in the intersection must have signed conflicting messages, providing evidence of misbehaviour.

Accountable protocols (formal definition). Consider an SMR protocol $\mathcal{P}$:

- We say the set of messages M is a *proof of guilt* for $p \in \Pi$ if there does not exist any execution of $\mathcal{P}$ with processor set Π in which p is correct and for which $M \subseteq M_c$. Intuitively, M is a proof of guilt for p if the signed messages in M could not all have been produced in any execution in which p behaves correctly.
- We say $\mathcal{P}$ is λ-*accountable* if the following holds at every timeslot t of any execution of $\mathcal{P}$: if $M_c(t)$ has a consistency violation, then $M_c(t)$ is a proof of guilt for at least a λ fraction of processors in Π. Note that this requires accountability to hold as soon as the consistency violation is witnessed by the set of messages received by correct processors, without requiring any additional communication.

Given that all standard protocols that are λ-accountable for any $\lambda > 0$ are also $1/3$-accountable, we will say that a protocol is *accountable* to mean that it is $1/3$-accountable.

20.2 Tendermint is accountable

In this section, we show that Tendermint is accountable. The proof just requires a careful examination of the proof of Consistency.

Theorem 20.1. *Tendermint is* 1/3*-accountable.*

Proof. Consider an execution of Tendermint, and suppose $M_c(t)$ has a consistency violation. From the definition of $\mathcal{F}$, it follows that there exists a least v such that:

- For some b with $b.\text{view} = v$, $M_c(t)$ contains all ancestors of b, stage 1 QCs for all ancestors of b, and also stage 1 and 2 QCs for b, Q_1 and Q_2 say.
- For some least $v' \geq v$, there exists b' such that b' is incompatible with b, with $b'.\text{view} = v'$ and $b'.\text{QC} = Q_0$ (say), and $M_c(t)$ contains a stage 1 QC for b', Q_3 say, as well as all ancestors of b' and stage 1 QCs for those blocks.

If $v = v'$ then, by the quorum intersection argument of Section 9.1.2, at least $f + 1$ processors must have sent votes in both Q_1 and Q_3. Since correct processors send at most one stage 1 vote in each view, $M_c(t)$ is a proof of guilt for all such processors: the two conflicting signed stage 1 votes for view v from each such processor could not both have been produced by a correct processor.

So, suppose $v' > v$. Then, by the same quorum intersection argument, at least $f + 1$ processors must have sent votes in both Q_2 and Q_3. We claim that $M_c(t)$ is a proof of guilt for all such processors. To see this, note that a correct processor p that produces a vote in Q_2 must set its local value `lock` to be a stage 1 QC for b while in view v. However, our choice of (v, v') and the fact that b' is incompatible with b means that $Q_0.\text{view} < v$, so that p would not regard the proposal b' as valid while in view v' and would not produce a vote in Q_3. To verify that this constitutes a proof of guilt from the messages in $M_c(t)$ alone, observe that $M_c(t)$ contains: the stage 2 vote by p in Q_2 (witnessing that p sent a stage 2 vote for b in view v); the stage 1 vote by p in Q_3 (witnessing that p voted for b' in view v'); and the block b' itself, including $Q_0 = b'.\text{QC}$ with $Q_0.\text{view} < v$. An outside observer can therefore verify from these messages that p voted for a proposal in view v' that it should not have regarded as valid, given its stage 2 vote in view v.

In both cases, we have identified at least $f + 1$ processors for which $M_c(t)$ is a proof of guilt. Since $f < n/3$, we have $f + 1 > n/3$, meaning that $M_c(t)$ is a proof of guilt for more than a 1/3 fraction of processors in Π. □

Chapter 21

Recovery

Chapter 22

Player reconfiguration protocols

In all previous chapters, we have considered protocols in which the set of processors Π is fixed throughout the execution. In practice, however, it is often desirable to allow the set of participating processors to change over time. For example, in a blockchain network, new nodes may wish to join whilst existing nodes may wish to leave or may become permanently unavailable. The process of changing the set of participating processors is called *player reconfiguration* (also referred to as *dynamic validator sets* or *membership changes* in parts of the literature).

Player reconfiguration introduces a number of subtleties that do not arise in the fixed-processor setting. Perhaps most importantly, the reconfiguration process itself must be agreed upon by the processors: if different processors have different views as to who the current participants are, Consistency may be violated.

In Section 22.1, we give an intuitive account of how player reconfiguration can be incorporated into an SMR protocol, building on the techniques developed in previous chapters. To keep things simple, we present a protocol which is based on the Pipelined Tendermint protocol of Section 9.3, and which functions in the partially synchronous model with synchronised clocks. However, the basic mechanisms we employ for incorporating player reconfiguration can also be straightforwardly carried over to deal with protocols for the asynchronous model. In Section 22.2, we give a formal specification. In Section 22.3, we verify that the protocol satisfies Consistency and Liveness. Finally, in Section 22.4, we briefly discuss the application of these techniques to 'proof-of-stake' protocols.

22.1 The intuition

Changing participants and the use of epochs. Instead of a fixed set of n processors Π, we now wish to consider a sequence of sets of processors, $\Pi_0, \Pi_1, \ldots$. The first set, Π_0, is given as input to all processors, while each Π_e for $e \geq 1$ will be determined as the execution progresses. A standard way to implement this is to divide the instructions into *epochs*. Processors in Π_0 carry out the instructions for epoch 0. For a parameter x, epoch 0 is responsible for finalising blocks of all heights[1] in the interval $[1, x]$. Then

[1] Recall from Section 9.2.4 that the height of a block is its number of ancestors.

processors in Π_1 carry out the instructions for epoch 1, and epoch 1 is responsible for finalising blocks of all heights in the interval $[x+1, 2x]$, and so on. We note that, while one could set $x = 1$, there are often motivations for considering more general values of x. For example, the use of cryptographic schemes such as threshold signatures may require a fresh (and potentially time-intensive) 'setup' for each new processor set, meaning that changes should be limited in frequency. Of course, the fact that the processor set changes with each epoch means that quorum certificates (QCs) must now be defined relative to the corresponding processor set, so that QCs are now epoch specific.

How should Π_e be defined for $e \geq 1$? We will suppose that this information is contained in the blocks finalised during the previous epoch. For example, certain transactions may specify changes to the processor set. Formally, we suppose given a function P^*. If b is the block of height x finalised by epoch 0, and if B is the set of all ancestors of b, then we set $\Pi_1 := P^*(B)$, and we proceed similarly in later epochs.

The size of the processor sets. In general, one could allow processor sets to vary arbitrarily in size. We will suppose $|\Pi_e| = n$ for all $e \in \mathbb{N}$, but this choice is made only for the sake of simplicity of notation – no significant complexities result from generalising our approach to allow processor sets of different sizes.

How to end an epoch? One complexity that arises is the precise process for ending an epoch. Here, we adapt Pipelined Tendermint to incorporate player reconfiguration, but similar considerations arise for other protocols such as Tendermint or (Streamlined) PBFT. The complication at the end of epoch 0, for example, is that when a block b of height x receives a QC, it is not yet finalised.[2] Since multiple blocks of height x can receive QCs, it is not yet safe to use the block b and its ancestors to specify Π_1. Instead, we must continue producing blocks for epoch 0 of height potentially greater than x, until a block b' (possibly b) of height x is finalised. Once this occurs, the blocks for epoch 0 of height greater than x, together with any received QCs, are kept as part of the data that verifies the validity of the chain, but are not used to constitute part of the chain of finalised transactions, i.e., processors for the next epoch begin building above b'. A similar process also applies at the end of later epochs, as illustrated in Figure 22.1.

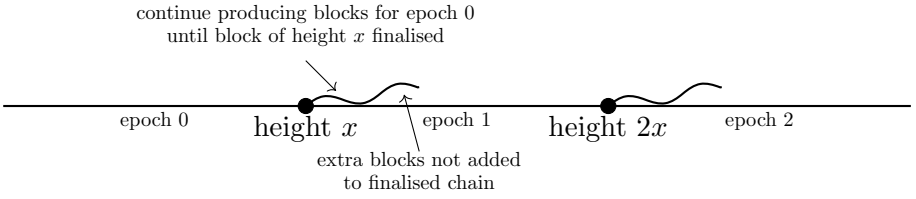


FIGURE 22.1: How to end an epoch without causing a consistency violation.

22.2 The formal specification

22.2.1 Modifying the formal framework

To incorporate the possibility of changing sets of participants requires some modification to the framework of Chapter 3. Accordingly, we now consider a potentially infinite set

[2] Similarly, in Tendermint or PBFT, a block of height x might receive a stage 1 QC and not yet be finalised.

of processors $\Pi^* = \{p_0, p_1, \dots\}$, and suppose that each Π_e is a subset of Π^*. There are a number of different formal assumptions we could make regarding the time-slots at which processors are active, and with respect to bounds on the number of Byzantine processors. For example, Lewis-Pye and Roughgarden [25] define a hierarchy of possible assumptions: the formalisation we consider here corresponds most closely (but is different) to what they call the *quasi-permissionless* setting. For the sake of simplicity, we just assume that:

- All processors in Π^* are active at all time-slots after GST, and;
- For any block b, if B is the set of all ancestors of b, then less than $1/3$ of the processors in $P^*(B)$ are Byzantine.

For an examination of ways in which these assumptions can be weakened, we refer the reader to [25].

22.2.2 Specifying the protocol

In what follows, we assume (without explicit mention in the pseudocode) that correct processors automatically send new transactions to all others upon first receiving them. We write $\emptyset$ to denote the empty set or empty sequence, and we let $\perp$ be some default distinguished value. The pseudocode uses a number of message types, local variables, functions and procedures, detailed below.

The local variables Π_e. The value Π_0 is given as input, while each Π_e for $e \geq 1$ is initially undefined.

The function $\texttt{lead}(e, v)$. This value specifies the leader for view v in epoch e. Processor p_i regards it as undefined until Π_e is defined. If $\Pi_e = \{p_0^*, \dots, p_{n-1}^*\}$, then $\texttt{lead}(e, v) := p_j^*$, where $j = v \bmod n$.

Blocks. A *block* b is a message specified by five values:

- b.epoch: this is a value in $\mathbb{N}_{\geq 0}$ that specifies the epoch corresponding to b;
- b.view: this is a value in $\mathbb{N}_{\geq 0}$ that specifies the view corresponding to b;
- b.Tr: a sequence of transactions in $\mathcal{T}$;[3]
- b.par: either $\perp$, or a hash value specifying the parent of b;
- b.QC: a stage 1 QC for b.par if b.par $\neq \perp$.

A correct processor only regards a message as a block if it is of the form above. The *genesis block* is denoted b_g and satisfies b_g.epoch $= 0$, b_g.view $= 0$, b_g.Tr $= \emptyset$, b_g.par $= \perp$, b_g.QC $= \perp$. We regard b_g as received by all correct processors at time-slot 0. Blocks are ordered by b.epoch, then by b.view, and then by least hash. The value b.Tr* is defined as in Section 9.2.3.

Notation for views. We refer to view v in epoch e as *view* (e, v).

[3] Recall that $\mathcal{T}$ is the set of possible transactions.

Votes. A *vote* for b with $e := b.\text{epoch}$, $v := b.\text{view}$, and $\tau := H(b)$, is a message of the form $V = (\text{vote}, e, v, \tau)$ signed by some processor in Π_e.

QCs. A QC for b with $e := b.\text{epoch}$ is a set Q of $n - f$ votes for b, each signed by a different processor in Π_e. We set $Q.\text{epoch} := e$, $Q.\text{view} := v$, and $Q.\text{block} := H(b)$. QCs are ordered by epoch, then view, and then by least hash. We stipulate that $\emptyset$ is a QC for b_g, and set $\emptyset.\text{epoch} := 0$, $\emptyset.\text{view} := 0$, and $\emptyset.\text{block} := H(b_\text{g})$. If Q is a QC for b, we also say Q is a QC for $H(b)$.

The lock. Each processor maintains a local variable `lock`, initially set to $\texttt{lock} := \emptyset$.

The local variable `e`. Initially set to 0, this variable specifies the present epoch.

The local variable `S`. Automatically updated, this local variable stores all messages received. Initially it contains only b_g.

When blocks are finalised. Recall that a protocol for SMR must specify a function $\mathcal{F}$ that determines when transactions are finalised. We will specify $\mathcal{F}$ below. First, we define a corresponding notion of finalisation for *blocks*. If M is a set of messages, we say a block b with $b.\text{epoch} = e$ is *finalised in* M if there exists a descendant of b_1 of b (possibly $b_1 = b$) and $b_2 \in M$ such that:

- $b_1.\text{epoch} = b_2.\text{epoch} = e$;
- $M \cup \{b_\text{g}\}$ contains all ancestors of b_1;
- $b_2.\text{view} = b_1.\text{view} + 1$, and;
- M contains QCs for b_1 and b_2.

The procedure SetEpoch. Each processor runs this procedure at time-slots that are integer multiples of Δ to set the local value `e`. Upon entering a new epoch e, the procedure also automatically defines Π_e and resets `lock` to be a QC for an epoch $e - 1$ block of height ex. The procedure appears in Figure 22.2.

The SetEpoch **procedure**.
1. Set $e := 0$;
2. If there exists b of height $(e + 1)x$ with $b.\text{epoch} = e$ that is finalised in `S`:
   Set $e := e + 1$;
   If $e >$ `e`:
     Let B be all ancestors of b and set $\Pi_e := P^*(B)$;
     Let Q be a QC for b in `S` and set $\texttt{lock} := Q$;
   Go to 2;
3. Set $\texttt{e} := e$;

FIGURE 22.2: The SetEpoch procedure

Valid proposals. At time-slot t, p_i regards a block b as a *valid proposal for view* (e, v) if all of the following conditions are satisfied:

(i) $b.\text{epoch} = e$ and $b.\text{view} = v$;

(ii) $b.\text{par} \neq \perp$ and $Q := b.\text{QC}$ is a QC for $b.\text{par}$;

(iii) Either $Q.\text{epoch} = e$, or else $Q.\text{block} = \texttt{lock}.\text{block}$, and;

(iv) $Q.\text{view} \geq \texttt{lock}.\text{view}$.

The procedure MakeProposal. This procedure is executed by the leader p of view (e, v) to determine a new block. To execute the procedure, p:

1. Lets Q be the greatest QC it has received with $Q.\text{epoch} = e$, or, if there exists no such, sets $Q := \texttt{lock}$.
2. Forms a sequence of distinct transactions T, containing all transactions received but not finalised by p;
3. Sets $b.\text{epoch} := e$, $b.\text{view} := v$, $b.\text{Tr} := T$, $b.\text{par} := Q.\text{block}$ and $b.\text{QC} = Q$.
4. Disseminates b.

Defining $\mathcal{F}$. Recall that blocks are ordered by epoch, then by view, then by least hash. For any set of messages M, let b be the greatest block such that b is finalised in M and such that, for $e := b.\text{epoch}$, b is of height at most $(e+1)x$. Set $\mathcal{F}(M) := b.\text{Tr}^*$.

The instructions are shown in Figure 22.3.

The instructions for p_i.
At time-slot $k\Delta$ for $k \in \mathbb{N}_{>0}$:
  SetEpoch;
  Disseminate all QCs received but not previously disseminated;

At time-slot $2v\Delta$ for $v \in \mathbb{N}_{>0}$:
  If $p_i = \texttt{lead}(\texttt{e}, v)$:
    MakeProposal; ▷ propose a new block if leader

At time-slot $2v\Delta + \Delta$ for $v \in \mathbb{N}_{>0}$:
  If p_i has received one valid proposal b for view $(\texttt{e}, v)$ from $\texttt{lead}(\texttt{e}, v)$:
    Set $\texttt{lock} := b.\text{QC}$ and send this value to all processors; ▷ set lock
    Send b and a signed vote for b to all processors; ▷ vote

FIGURE 22.3: The pseudocode for Pipelined Tendermint with player reconfiguration.

22.3 The verification

22.4 Further discussion

Chapter 23

2-round finality

Chapter 24

The Pipes model for latency and throughput analysis

Appendix A

Index of Notation

General

n	The number of processors.
f	The maximum number of faulty processors.
$\Pi = \{p_0, \ldots, p_{n-1}\}$	The set of processors.
Π^*	A potentially infinite set of processors, used in the reconfiguration setting (Chapter 22).
Π_e	The set of processors active in epoch e (Chapter 22).
P	A set of processors.
V	The set of possible input/output values for Byzantine Agreement and Byzantine Broadcast.
$\mathcal{T}$	The set of possible transactions.
$\perp$	A default distinguished value.
$\emptyset$	The empty set or empty sequence; also used as the default stage 1 and stage 2 QC for the genesis block.
E	An execution.
$\mathcal{P}$	A protocol.

Time and communication

t	A time-slot ($t = 0, 1, 2, \dots$).
Δ	A known upper bound on message delivery time (in the synchronous and partially synchronous models).
δ	The actual (unknown) message delivery time after GST, satisfying $\delta \leq \Delta$.
GST	The global stabilisation time: the unknown time-slot after which all messages are delivered within Δ time-slots (in the partially synchronous model).
$\{i, j\}$	The authenticated communication channel between processors p_i and p_j.

Signatures and hash functions

$\langle m \rangle_i$	The message m signed by processor p_i.
$\langle v \rangle_{i_1,\dots,i_t}$	The value v signed successively by processors $p_{i_1}, \dots, p_{i_t}$.
$\langle m \rangle_O$	The message m signed by an oracle O.
H	A collision-free hash function.

Blocks and chains

b	A block.
b_{g}	The genesis block.
b.view	The view corresponding to block b.
b.epoch	The epoch corresponding to block b (Chapter 22).
b.Tr	The sequence of transactions contained in block b.
b.Tr*	The extended sequence of transactions specified by the ancestors of b (with duplicates removed).
b.par	The hash of the parent of block b (or $\perp$ for the genesis block).
b.QC	A QC for the parent of b, included in b.
b.just	A justification included in b (used in Streamlined PBFT).

Voting and certificates

QC	A quorum certificate: a set of $n - f$ votes from distinct processors for the same block in the same view.
Q.view	The view corresponding to the QC Q.
Q.block	The hash of the block corresponding to the QC Q.
TC	A time-out certificate: a set of $n - f$ signed time-out messages for the same view from distinct processors.
`lock`	A local variable maintained by each processor, recording a stage 1 (or stage 2) QC. Used to prevent voting for blocks that would violate Consistency.

Protocol structure

v	A view number ($v = 0, 1, 2, \dots$).
`v`	A local variable recording the processor's current view.
`lead`(v)	The leader for view v, generally defined as p_i where $i = v \bmod n$.
`lead`(e, v)	The leader for view v in epoch e (Chapter 22).
`Timer`	A local timer variable, which increments in real time after GST.

Sequences and logs

$\log_i$	The append-only log maintained by processor p_i.
$\log_i(t)$	The value of $\log_i$ at the end of time-slot t.
$\sigma \preceq \tau$	The sequence σ is a prefix of the sequence τ.
$\text{tr} \in \sigma$	The transaction tr belongs to the sequence σ.

Complexity metrics

ℓ	A liveness parameter or mempool dissemination bound.
$\mathcal{F}$	The finalisation function used in the definitions of SMR and Extractable SMR.

Appendix B

Communication complexity for the crash-fault-model

In this appendix, we give a simple proof that binary Crash-fault Broadcast can be solved with communication complexity $n + f$. The proof is aimed at simplicity rather than practicality. Accordingly, we make no effort to minimise the time before correct processors output.

The intuition. The basic idea behind the protocol is almost trivial. We consider the processors $p_0, \ldots, p_{n-1}$ ordered so that p_0 is the broadcaster. We attempt to pass p_0's input value v from one processor to the next in order, so that each can set its own value to v and output v. However, we must also accommodate the possibility that up to f processors may crash. To do so, we let $t_0^* = 0$ and set t_i^* for $i > 0$ to be some time-slot by which p_i expects to be passed a value by p_{i-1} if the latter processor is not faulty: we will work out precisely how to define t_i^* for $i > 0$ subsequently. If $i > 0$ and p_i does not receive a value from p_{i-1} by t_i^*, then it sends a 'request value' message to p_{i-2} (or sets its own value to be some default value if $i - 2 < 0$). If p_{i-2} is not faulty, then it will send its value to p_i, so that p_i can then pass this value to p_{i+1}. Otherwise, if p_i does not receive any value from p_{i-2} (by $t_i^* + 2$), it requests a value from p_{i-3} (if $i - 3 \geq 0$), and so on.

Let $f_a \leq f$ be the actual number of faulty processors. Then each of the $n - f_a$ correct processors (except, perhaps, the last) sends one message to the next processor in the sequence. So far, that means correct processors sending $n - f_a$ messages. If $p_{i+1}, \ldots, p_{i+k}$ is a consecutive sequence of faulty processors, they may cause p_i to send an extra k messages (in response to requests for its value), and may also cause p_{i+k+1} to send an extra k (request) messages. So, each faulty processor can cause the correct processors to send at most two extra messages, and the correct processors send a total of $(n - f_a) + 2f_a = n + f_a$ messages.

To define each t_i^* for $i > 0$, we must choose some sufficiently large value. Given t_{i-1}^*, it suffices to set $t_i^* = t_{i-1}^* + 2f + 1$, since p_i must send at most f request messages. Sending each request message and waiting for a response requires two time-slots.

Keeping 'request value' messages small. Values are either 0 or 1, meaning that messages that pass values need only be a single bit. However, to achieve communication complexity $n + f$, we must also keep 'request value' messages small. Since processors

only send values to processors that are greater in the ordering, and only send 'request value' messages to processors that are earlier in the ordering, this can be achieved by sending 0 to request a value: processor p_i can interpret receiving a 0 from p_j for $j > i$ as requesting their value.

The protocol specification. Let the processors be ordered so that p_0 is the broadcaster. For $i \in \mathbb{N}_{\geq 0}$, let t_i^* be defined as above. Each processor p_i maintains a local value v, initially undefined. The instructions are shown below.

Crash-fault Broadcast: the instructions for p_i at time-slot t.
If v is undefined and $i = 0$:
  Set $v :=$ input; output v; ▷ leader sets v

If v is undefined and p_i has received $v' \in \{0, 1\}$ from some p_j for $j < i$:
  Set $v := v'$; output v; ▷ set value and output

If v is defined, $i < n - 1$, and p_i has not yet sent a message to p_{i+1}:
  Send v to p_{i+1} ▷ pass on value

If v is defined and p_i receives 0 from some p_j for $j > i$ at t:
  Send v to p_j ▷ pass on value due to request

If v is undefined and $t = t_i^* + 2k$ for $k \in \mathbb{N}_{\geq 0}$:
  If $i - k - 2 \geq 0$, send 0 to p_{i-k-2} ▷ request value
  If $i - k - 2 < 0$, set $v := 0$; output v; ▷ default to 0

Analysis. First, we establish satisfaction of Termination, Agreement, and Validity.

To show Termination is satisfied, we establish a stronger result by induction: if p_i is correct, then it defines its local value v and outputs before t_{i+1}^*. For $i = 0$, the result is immediate. So, suppose the claim holds for all $i - 1 < n - 1$. Then p_i either defines its local value v by t_i^*, or else must request values from at most f previous processors before either setting its value to the default 0, or requesting a value from some correct processor p_j. In the latter case, it follows from the induction hypothesis that p_j has already defined its value, and so passes this value to p_i. Processor p_i therefore defines its local value by $t_i^* + 2f < t_{i+1}^*$, as claimed.

To establish Agreement, let i be the least value in $[0, n - 1]$ such that p_i is correct, and suppose p_i outputs x (after setting its value to x). If $j > i$ and if p_j defines its local value and outputs, then this value is passed to it by some processor p_k where $i \leq k < j$: as established above, t_i^* and t_j^* are set so that p_i sets its value to x prior to t_j^* and will send this value to p_j if requested, so that p_j does not request values from processors before p_i. It follows by induction that if $j > i$ and p_j outputs, then it outputs the same value x as p_i. This establishes Agreement, as required.

To establish Validity, let p_i and x be as defined in the discussion of Agreement above. If the broadcaster is correct, then $p_i = p_0$ and x is p_0's input. So, correct processors give the broadcaster's input as their output.

That the protocol has communication complexity $n + f$ then follows as explained previously.

Appendix C

Recursive Phase-King

Phase-King requires $f + 1$ views because we need to ensure there is at least one view with a correct leader. The first such view suffices to ensure all correct processors have the same value, and once this is the case, agreement is maintained in subsequent views. Since each view requires correct processors to send $O(n^2)$ bits, the overall communication complexity for Phase-King is $O(n^2 f)$. In this appendix, we briefly outline the idea behind Recursive Phase King [11], which achieves communication complexity $O(n^2)$. As for Phase-King, we consider the lock-step model without signatures, and suppose $f < n/3$.

Replacing $f + 1$ views with two. The key observation behind Recursive Phase King is that we can replace the $f + 1$ views with two if we replace leaders with *committees*. We partition the n processors into two committees, C_1 and C_2, which differ in size by at most 1. At least one of these committees must be *sufficiently honest*, meaning it has a fraction of faulty processors at most f/n. Rather than having a single leader send their value in view 1, we have committee C_1 carry out a recursive call to BA, and then send their outputs to all processors. If C_1 is sufficiently honest, the committee members will agree on a value, and all processors will then receive the same majority value from the committee. In view 2, we do the same using committee C_2. Since at least one of C_1 or C_2 is sufficiently honest, at least one of them effectively acts as a correct leader.

Why is communication complexity $O(n^2)$? For some constant $c \in \mathbb{N}$, the all-to-all communication in views 1 and 2 requires correct processors to send at most cn^2 bits (at 'depth 1', say) *in addition to* those bits required by the two recursive calls. Ignoring rounding errors, those two recursive calls require the sending of $c(n/2)^2 + c(n/2)^2 = cn^2/2$ bits (at 'depth 2', say), and so on. Summing over all depths, the protocol requires correct processors to send at most $2cn^2$ bits. (For sufficiently small committees, one can invoke some fixed BA protocol, requiring $O(n)$ bits over all instances.)

Appendix D

Bounding complexity for Extractable BA

[Andy: To add]

Bibliography

[1] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982. ISSN 0164-0925. doi: 10.1145/357172.357176. URL https://doi.org/10.1145/357172.357176.

[2] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[3] Russell Turpin and Brian A Coan. Extending binary byzantine agreement to multivalued byzantine agreement. *Information Processing Letters*, 18(2):73–76, 1984.

[4] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[5] Marcos Kawazoe Aguilera and Sam Toueg. A simple bivalency proof that t-resilient consensus requires t+ 1 rounds. *Information Processing Letters*, 71(3-4):155–158, 1999.

[6] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *Journal of the ACM (JACM)*, 32(1):191–204, 1985.

[7] Piotr Berman, Juan A Garay, and Kenneth J Perry. Bit optimal distributed consensus. In *Computer science: research and applications*, pages 313–321. Springer, 1992.

[8] Atsuki Momose and Ling Ren. Optimal communication complexity of authenticated byzantine agreement. *arXiv preprint arXiv:2007.13175*, 2020.

[9] Zvi Galil, Alain Mayer, and Moti Yung. Resolving message complexity of byzantine agreement and beyond. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 724–733. IEEE, 1995.

[10] Michael J Fischer, Nancy A Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1:26–39, 1986.

[11] P Berman, JA Garay, and KJ Perry. Recursive phase king protocols for distributed consensus. *Penn State Report CS-89-24*, 1989.

[12] Srivatsan Sridhar, Ertem Nusret Tas, Joachim Neu, Dionysis Zindros, and David Tse. Consensus under adversary majority done right. *arXiv preprint arXiv:2411.01689*, 2024.

[13] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2): 374–382, 1985.

[14] Eli Gafni and Giuliano Losa. Time is not a healer, but it sure makes hindsight 20: 20. In *International Symposium on Stabilizing, Safety, and Security of Distributed Systems*, pages 62–74. Springer, 2023.

[15] Hagen Völzer. A constructive proof for flp. *Information processing letters*, 92(2): 83–87, 2004.

[16] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[17] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains.* PhD thesis, University of Guelph, 2016.

[18] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus. *arXiv preprint arXiv:1807.04938*, 2018.

[19] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[20] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, number 1999 in 99, pages 173–186, 1999.

[21] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

[22] Benjamin Y Chan and Rafael Pass. Simplex consensus: A simple and fast consensus protocol. In *Theory of Cryptography Conference*, pages 452–479. Springer, 2023.

[23] Victor Shoup, Jakub Sliwinski, and Yann Vonlanthen. Kudzu: Fast and simple high-throughput bft. *arXiv preprint arXiv:2505.08771*, 2025.

[24] Brendan Kobayashi Chou, Andrew Lewis-Pye, and Patrick O'Grady. Minimmit: Fast finality with even faster blocks. *arXiv preprint arXiv:2508.10862*, 2025.

[25] Andrew Lewis-Pye and Tim Roughgarden. Permissionless consensus. *arXiv preprint arXiv:2304.14701*, 2023.