Consensus in 50 pages (rough draft)

ANDREW LEWIS-PYE

The literature on consensus protocols can seem impenetrable despite the fact that modern consensus protocols are quite simple. The aim of these notes is to give clear explanations of the results from the classical literature that are most relevant to the modern study of blockchains.

The state of this version. These notes are under construction, but I'll only post versions that have been carefully proofread. At present, I get as far as the 'synchronous' and 'partially synchronous' settings, but have not yet treated the asynchronous setting. If you don't know what those terms mean, don't worry. All will be explained below! In the partially synchronous setting, we get as far as describing protocols like Tendermint and Casper FFG. In the next few weeks, I'll be adding notes on Hotstuff, PBFT, Gasper and DAG-based protocols. I'll add material on asynchronous protocols after that. I'll also be uploading a sequence of videos to my newly created YouTube channel (just search under my name). Please do write to me at a.lewis7@lse.ac.uk with questions or comments.

1 REQUIRED BACKGROUND

We assume the reader has some basic background in computer science. In particular, we suppose the reader is happy to talk about such things as 'computing devices' and has some familiarity with the ways in which such things can be formalised. We blackbox the use of any cryptographic methods and presume the reader is familiar with the idea that cryptographic schemes may be used to produce unforgeable digital signatures. Later on, we will assume some familiarity with Bitcoin.

2 THE PROBLEM (INFORMALLY)

(1) What do we mean by a 'consensus protocol'? What is the problem a consensus protocol has to solve? There are three main consensus problems considered in the literature: Byzantine Agreement, Byzantine Broadcast and State Machine Replication. We start with an informal introduction to the Byzantine Agreement problem.

(2) **The Byzantine Agreement problem (informal version).** We give a description similar to Lamport, Shostak and Pease [**?**], who introduced the problem in the early 1980s. 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:

  (a) **Termination.** Each honest general must eventually reach a decision (retreat or attack).
  (b) **Agreement.** All honest generals must reach the same decision.
  (c) **Validity**. If all honest generals start with the same opinion, then that common opinion must be the same as their final decision.

(3) **The need for validity**. The description of the Byzantine Agreement problem above is informal. To reason about it formally we will need to define a precise version of the problem.

Emails. Andrew Lewis-Pye: a.lewis7@lse.ac.uk, andy@lewis-pye.com.

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 set things up. Even with this informal version of the problem, though, there are certain observations that can be made. Note, for example, that without the requirement for 'validity' the problem above is certainly trivial because the protocol could just have all honest generals ignore their initial opinion and always decide to attack.

(4) **If all are honest.** To understand the problem better, it is best to start by testing trivial cases. Suppose that there are n generals of which at most f are dishonest. For clarity, let us suppose the honest generals know f (but not which are the dishonest generals). Note that if $f = 0$ then the problem is trivial. For example, we can just have each general communicate their initial opinion to the others and then decide on the majority vote (with ties broken in favour of "retreat", say). So, it is clear that one of the basic questions we should be interested in is "what values of n and f can a protocol handle?".

(5) **If $f \geq n/2$.** It is also easy to see that no protocol can work if $f \geq n/2$. To see this, let us consider the case $n = 2$. Then the reader can easily extend the argument to the general case. 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 output "attack" if they behave honestly (a similar argument will apply if they both output "retreat"). Now consider what happens when the both generals have initial opinion "retreat" but the first is dishonest and behaves during the protocol exactly like an honest general with initial opinion "attack", while the second general behaves honestly. In this case, the protocol must cause both generals to output "attack", just as in the first case considered, even though all honest generals started with the opinion "retreat". This contradicts the validity requirement.

(6) **Is it trivial when $f < n/2$?** We saw above that the problem is trivially *non-solvable* when $f \geq n/2$. It is also initially tempting to think that the problem might be trivially *solvable* when $f < n/2$. Can we not just implement a majority vote argument of the sort described in paragraph 4 above, for example? The problem with this approach stems from the allowed form of communication between generals, which is intended to accurately reflect communication between processors 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 would work. In the setting described above, however, communication occurs by messenger between one pair of generals at a time. The problem now is that dishonest generals can tell different things to different generals. Suppose, for example, that $n = 3$ and $f = 1$. Suppose one honest general initially wants to attack, while the other wants to retreat. If we carry out a simple majority vote protocol as described above, and if the dishonest general sends a 'retreat' message to the general who wants to retreat and an 'attack' message to the general who wants to attack, then the honest generals will see different majority votes and so will decide differently.

(7) 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. To establish impossibility results of this kind, though, we certainly need a formal framework. So, that is what we will set up in the next section.

3 THE FORMAL FRAMEWORK

(1) The decisions we have to make in formally defining the setup include such things as how long messages take to arrive (or whether they are certain to arrive), and what sort of behaviour 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 model and the particular problem asked (be it the Byzantine Agreement problem or some variant). This means one has to keep track of a large number of different scenarios and the relationships between them.

(2) **Formalising the generals as processors.** We formalise each general as a processor. The particular computational model used is not important for the results discussed here. For example, one could formalise each general as an Interactive Turing Machine,[1] in which case the required protocol will be a program for the Turing machine. We take instead the same approach as most of the early consensus literature and consider each processor to be specified by a state transition diagram. This means the execution of the protocol is divided into discrete time units, beginning at time $t = 0$. At each time t, each processor i begins in a certain state x and receives a certain set of messages M from the other processors. Given x and M (where M includes information as to who sent which message), the state transition diagram then determines: (a) The set of messages that i sends at time t, and; (b) The state that i is in at time $t + 1$. So, according to this model, the state transition diagram specifies the protocol.[2]

(3) **Inputs and outputs**. Certain states of the state transition diagram for i are labelled as 'input states', and i begins the execution of the protocol at time 0 in one of these input states. The input to i determines which input state i starts in. Certain states may also labelled as 'output' states and may be used to determine i's output.

(4) **Thinking of processors as Turing Machines instead.** If you prefer to think of processors as Interactive Turing Machines (or RAM machines, or whatever your favourite model is) that is fine and it will not impact any of the proofs in these notes, but you will have to allow that processors can carry out multiple basic instructions in a single timeslot. For example, the instructions in a single timeslot might say, "Check all the messages received and if $f + 1$ of them are 1s then send a 0". Clearly, instructions of this form consist of a number of basic Turing machine operations. If you are thinking of processors as Turing Machines then the input will be given on the input tape, rather than determining the choice of initial state (as for processors specified by state transition diagrams).

(5) **Names for processors**. We suppose there are n processors given names 0 to $n - 1$. Each processor is told n as well as their own name i, i.e. this information is given as part of their input.

(6) **Authenticated channels.** There exists a two-way communication channel $\{i, j\}$ for each pair of distinct processors i and j. The communication channel $\{i, j\}$ is *authenticated* in the sense that:

- Only i can send messages to j and only j can send messages to i on the channel $\{i, j\}$, and;
- When i receives messages it is aware of the channel by which the messages were sent, i.e. the instructions for i can depend not only on the messages received at any given timeslot but also which channels the messages arrived on.

At each timeslot, the instructions for processor i determine which messages it should send along each of its channels $\{i, j\}$.

[1] The machine being interactive just means that it can send and receive messages from other processors during the execution.

[2] There is no requirement that the state transition diagram be finite, or even computable.

(7) **Message delay and the synchronous setting**. There are a number of different possible assumptions regarding the reliability of message delivery. To keep things simple, we start by considering what is known as the *synchronous* setting. This means that if i sends j a message at time t then j receives that message[3] from i at time $t+1$.[4] This is quite a strong assumption, so later we will consider two other weaker assumptions regarding the reliability of message delivery (each giving a different set of corresponding results).

(8) **How can dishonest generals behave?** We suppose some of the processors may be *faulty*. Each processor is given an upper bound f for the number of faulty processors as part of its input. Generally, in the context of the 'blockchain' literature, we are most interested in analysing settings where the faulty processors can display arbitrary (and potentially malicious) behaviour. In this case, we say that the processors display *Byzantine* faults. This is the case we will focus on in these notes. Formally, this means that faulty processors can have any state transition diagram.

(9) **The adversary**. In the paragraph above, we have formally specified how the faulty processors may behave. When we reason informally, it will be often be convenient to think of the faulty processors as being coordinated by an *adversary* who may choose their actions so as to be as troublesome as possible for the protocols we design.

(10) **Crash faults.** Sometimes we will also be interested in a more benign form of faulty behaviour known as *crash faults*. In the crash fault setting, faulty processors must follow the protocol precisely until such a point as they crash, whereupon they execute no further instructions.[5]

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

(12) Sometimes we will assume that a signature scheme is available and sometimes we will not. Where we make use of a signature scheme, we entirely blackbox the necessary cryptography. When we say that a **Public Key Infrastructure (PKI)** is given this simply means that:

- Each processor is provided with a public/private key pair and is told the public key of each of the other processors as part of its input, i.e. it is told the public key for each processor i.
- We simply *decree* that only i can produce its own signatures, i.e. we restrict the set of executions considered to those in which it holds that if part of a message sent by j is signed by a different processor i, then j must have previously received that part of the message signed by i.

[3] For the sake of complete clarity, we add that messages sent at different timeslots are regarded as distinct and that each message is received only once.

[4] Another possibility is to allow that there is some finite bound $\Delta \geq 1$ such that if i sends a message at time t then it arrives by time $t+\Delta$ at the latest. We will not need to differentiate between these models because: (a) Any protocol that can handle the latter setting can clearly also handle a more reliable setting in which messages are always delivered at the next timeslot, and; (b) All of the protocols we consider in these notes for the synchronous setting can easily be modified to handle the weaker assumption that messages arrive within time Δ by executing the same instructions only at timeslots that are multiples of Δ (and modifying the labelling of timeslots for the instructions accordingly).

[5] For the sake of complete clarity, we add that (in the crash fault setting) it is assumed processors may crash at any point while executing the instructions for a given timeslot. So, if the instructions require them to send five messages, the processor may crash after sending the first three.

Of course, those readers coming from a security background may prefer to assume the more sophisticated setup used in that context, which assumes that the processors are Polynomial-Time Bounded Interactive Turing Machines (this being a setup that actually allows one to analyse and prove results about the cryptographic schemes concerned).

(13) **The research program**. 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 synchronous setting or in a setting for which message delivery is less reliable?
- Does the answer depend on whether a PKI is given?

(14) The order in which we address these issues is as follows. We start in the synchronous setting. We consider Byzantine faults and we determine for which n and f Byzantine Agreement is possible, depending on whether a PKI is given. Then we repeat all of the same questions for settings that make weaker assumptions about message delivery. Along the way, we will also consider two other variants of the Byzantine Agreement problem, called ‘Byzantine Broadcast’ and ‘State Machine Replication’.

4 BYZANTINE AGREEMENT (BA) AND BYZANTINE BROADCAST (BB)

(1) In Section 2 we considered a version of the Byzantine Agreement problem which was binary in the sense that processors had to decide between two output options. In fact, it is sometimes convenient to consider a more general form of the problem:
- We consider a set of n processors, of which at most f display Byzantine faults.[6]
- For some set V, each processor is given an input in V (different processors potentially receiving different inputs). V is told to the processors[7] and could be of any finite size ≥ 2.
- The protocol must satisfy the following conditions:
  - **Termination**. All non-faulty processors must give an output in V.
  - **Agreement**. All non-faulty processors must give the same output.
  - **Validity**. If all non-faulty processors have the same input v, then v must be their common output.

(2) On seeing the more general version of the problem outlined above, it is natural to ask 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 [], 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 these notes 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).

[6] Recall from section 3 that processors are told n and f.

[7] A small detail (whose purpose will become clear later) is that we will assume V is given in such a way that some generic algorithm allows all non-faulty processors to pick the same ‘default’ value $v \in V$ given only V as input. This will be the case if V has a first element, or if all elements of V are represented by finite binary codes, or if V is just given directly with a distinguished element. The same assumption is also made for Byzantine Broadcast.

(3) 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 known as *Byzantine Broadcast*:
- We consider a set of n processors, of which at most f display Byzantine faults.
- One processor is designated the 'broadcaster'. All processors are given 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 non-faulty processors must give an output in V.
  - **Agreement**. All non-faulty processors must give the same output.
  - **Validity**. If the broadcaster is not faulty and has input v, then all non-faulty processors must output v.

(4) What is the relationship between the Byzantine Agreement (BA) problem and the Byzantine Broadcast (BB) problem? In paragraph 5, we noted that BA cannot be solved when $f \geq n/2$. It is easy to see, though, that the argument given there does not apply to BB (the reader is invited to check). In Section 5 we will see that, if a PKI is given and we work in the synchronous setting, 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.

(5) On the other hand, if we work in the synchronous setting 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 0. Each processor then takes the value received at time 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.

(6) So far, it might seem that BB is strictly easier than BA. As a word of caution, we will later see settings such as the *partially synchronous* and *asynchronous* settings, in which BB is not possible although protocols do exist to solve BA. So, the two problems are not strictly comparable in a general sense.

5 BA AND BB IN THE SYNCHRONOUS SETTING WITH PKI

5.1 A positive result – The Dolev Strong protocol

In this section, we prove the following theorem.

Theorem 5.1. *Consider the synchronous setting with PKI given. There exists a protocol that solves the Byzantine Broadcast problem for any number of faulty processors.*

(1) **This also deals with BA**. By the reductions discussed in Section 4, Theorem 5.1 also suffices to show that we can solve BA when working in the synchronous setting with PKI iff $f < n/2$.

(2) **Why isn't it trivial?** Before describing the proof, let us explore why a trivial solution does not work. An obvious way to try solving BB when given a PKI 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, then they realise the broadcaster is faulty, so they give some 'default' value as output.

(3) Of course, the idea behind this approach is that if the broadcaster is non-faulty then they will only produce a single signed value and all non-faulty processors will output that. If the broadcaster produces two different signed values and shows them to non-faulty processors then everyone will eventually see those values and give the default output.

(4) **The problem**. The problem with this approach is clear. At what point should the processors stop sharing values and terminate? If they share until time t, then the adversary[8] can choose to show one signed value to all non-faulty processors until time t, and then show some subset of the non-faulty processors a second signed value at time t (when it is too late to share anymore), causing the 'agreement' requirement of BB to be violated. We could require processors to ignore new values seen at the last timeslot t, but this only takes things one step back – now the adversary just has to show some subset of the non-faulty processors a second signed value at time $t-1$.

(5) **The trick (informal)**. What we need is a clever mechanism to ensure that if any non-faulty processor 'recognises' a certain signed value produced by the broadcaster, then all non-faulty processors will also 'recognise' that value. That way, either they all recognise a single value and give that as output, or they all recognise multiple values and so give the default output. The mechanism used to achieve this was initially described by Dolev and Strong[9] [] and is quite elegant:

  (a) At time 0 the broadcaster sends signed versions of their input to each processor.

  (b) At time 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 send this new version of the message to all processors.

  (c) Then we stipulate that if a processor is to 'recognise' a new value at any time t, the message must have been signed by t distinct processors. If they recognise a new value at time t, then they add their signature to the list and send that message (now with $t+1$ distinct signatures) to all other processors.

  (d) At time $f+1$ 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 a default value.

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

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

[8] Recall the informal concept of an adversary from Section 3

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

(7) **A more 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. For $v \in V$, and for distinct processors $i_1, \dots, i_t$, we let $v_{i_1,\dots,i_t}$ be v signed by $i_1, \dots, i_t$ in order, i.e., for the empty sequence $\emptyset$, $v_\emptyset$ is v, and for each $k \in [1, t]$, $v_{i_1,\dots,i_k}$ is $v_{i_1,\dots,i_{k-1}}$ signed by i_k. Let M_t be the set of all messages of the form $v_{i_1,\dots,i_t}$ such that $v \in V$ and $i_1, \dots i_t$ are all distinct processors. Each processor i maintains a set O_i, which can be thought of as the set of values that i recognises, and which is initially empty. We let $\perp$ denote a 'default' element of V.

(8) The instructions for processor i are as follows.

Time 0. If i is the broadcaster and if i's input is v, then i sends v_i to all processors and enumerates v into O_i.
Time t **with** $1 \leq t \leq f + 1$. Consider the set of messages $m \in M_t$ that i receives at time t. For each such message $m = v_{i_1,\dots,i_t}$, if $v \notin O_i$, proceed as follows: Enumerate v into O_i and if $t < f + 1$ send m_i to all processors.

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

(9) It is then easy to re-express the informal proof of paragraph 6 in the more formal language of paragraphs 7 and 8 above.

(10) **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 messages sent by honest parties, note that it suffices for each processor to recognise and pass on at most *two* values. That way, either all processors recognise no values at all, or they all recognise a single value, or they all recognise two values and give the default output. In any case, the protocol outputs correctly.

5.2 A negative result: $f + 1$ rounds of communication are necessary

The protocol described in Section 5.1 involved $f + 1$ rounds of communication. In the same paper that Dolev and Strong described the protocol above [], they proved that this is optimal, i.e. $f + 1$ rounds are required. In this section we describe a simpler proof of this result due to Marcos Aguilera and Sam Toueg. The proof demonstrates an important technique, but readers might choose to skip it on a first reading.

TO BE ADDED

5.3 A negative result regarding message complexity: Dolev and Reischuk

TO BE ADDED

6 BA AND BB IN THE SYNCHRONOUS SETTING WITHOUT PKI

6.1 The negative result

In this section, we prove Theorem 6.1 below. It's a beautiful proof and a classic of the literature. On the other hand, it's probably okay to skip it if your aim is just to get up to date with modern protocols and if you don't intend to do research in the area.

Theorem 6.1. *Consider the synchronous setting without PKI. No protocol can solve BA or BB when* $f \geq n/3$.

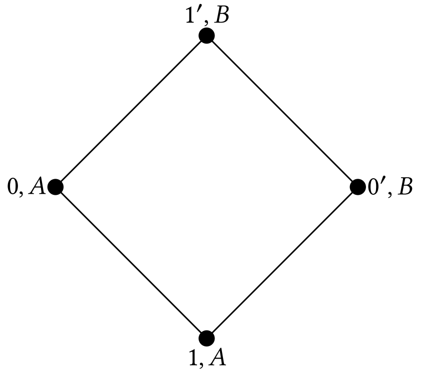

Fig. 1

(1) Note that, by the reductions discussed in Section 4, it suffices to prove the theorem for BA. Theorem 6.1 was originally proved by Lamport, Shostak and Pease [], but the more elegant proof we present here is due to Fischer, Lynch and Merritt [].
(2) According to the setup described in Section 3, each processor $i \in \{0, \dots, n-1\}$ knows which processor is at the other end of each communication channel $\{i, j\}$, i.e. if i receives a message on communication channel $\{i, j\}$ then i knows this message must be from j. To present a proof of Theorem 6.1 for which it is entirely clear why all elements of the proof are necessary, however, 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. We will come back to the proper setup soon.
(3) In such a context, we can describe a simple proof by contradiction. The approach taken might seem a bit unusual, because what we do is to suppose that a working protocol exists and then we deduce certain things about how the protocol must behave in scenarios which are different than the intended application. This might seem a strange thing to do, but the analysis allows us to quickly deduce that there exist valid executions of the protocol in which it does not behave correctly.
(4) 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 1. In Figure 1, each node represents a non-faulty processor. We refer to them by the names 0, 1, $0'$, and $1'$ and suppose that each processor i or i' for $i \in \{0, 1\}$ is told their name is i. Next to each processor is also indicated its input, either A or B. There are communication channels between processors 0 and 1, between processors 0 and $1'$, between processors $1'$ and $0'$ and between processors $0'$ and 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 timeslot. 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).

(5) What can we deduce about the outputs of the processors in this setup? Processors 0 and 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 0 and 1 both have input A, and in which the third processor is faulty and simulates $0'$ and $1'$ as arranged as in the figure (i.e the faulty processor sends to processor 1 at each stage what $0'$ sends to 1 in the four processor execution, and sends to 0 what $1'$ sends to 0 in the four processor execution). 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 timeslot in the two executions.

(6) Similarly (and symmetrically), processors $0'$ and $1'$ must output B.

(7) Now we get to our contradiction. We have already deduced that 0 must output A and that $1'$ must output B. The problem is that this execution is indistinguishable as far as processors 0 and $1'$ are concerned from a valid execution in which the third processor is faulty and simulates $0'$ and 1 as arranged in the figure. The fact that 0 and $1'$ give different outputs therefore violates the 'agreement' requirement of BA.

(8) **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 0 and 1 in Figure 1. If processors 0 and 1 both treat the channel between 0 and 1 indicated in the figure as the channel $\{0, 1\}$, and if each processor $i \in \{0, 1\}$ treats the channel between i and $(1 - i)'$ indicated in the figure as the channel $\{i, 2\}$, then we can again deduce that processors 0 and 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 $0'$ and $1'$ output B. The problem now is that this execution does not look indistinguishable from any valid execution as far as processors 0 and $1'$ are concerned, because in any valid execution they would both treat the communication channel between them as the channel $\{0, 1\}$ rather than $\{0, 2\}$.

(9) There is a simple way to remedy this problem. We just introduce two further processors, as in Figure 2. We suppose that each processor with the name i or i' in the figure is told that its name is i, and that it treats the channel in the figure to any processor j or j' as the channel $\{i, j\}$. The argument now works much as before. Processors 0 and 1 must output A because for them the execution is indistinguishable from a three processor execution in which the processor 2 is faulty and simulates processors $2', 0', 1'$ and 2 in the figure. Processors $2'$ and $0'$ must output B because for them the execution is indistinguishable from a three processor execution in which the processor 1 is faulty and simulates processors $1, 0, 2$ and $1'$ in the figure. Now we get to the contradiction because processors 1 and $2'$ give different outputs, but the execution is indistinguishable for them from a three node execution in which processor 0 is faulty and simulates processors $0, 2, 1'$ and $0'$ as arranged in the figure.

(10) 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 as follows. We divide $\{0, \dots n - 1\}$ into three disjoint subsets I_0, I_1, I_2, so that each subset has at most one more element than the others. Then we have each processor i simulate the set of processors I_i in the n processor execution, giving as output the common output of all processors in I_i.

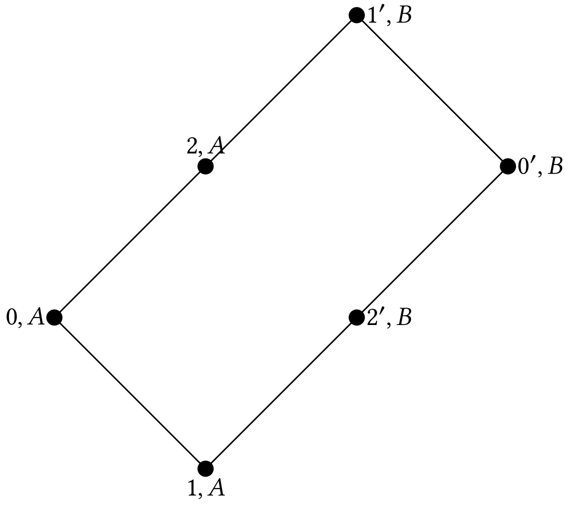

Fig. 2

6.2 The positive result

In this section, we prove the following theorem. The proof is instructive, but it's also possible to skip it and understand all later material.

Theorem 6.2. *Consider the synchronous setting without PKI. There exist protocols solving BA and BB whenever $f < n/3$.*

(1) **Initial considerations**. Note that the proof of Section 5.1 will not work without a PKI. Without a PKI, faulty processors will be able to pretend they have seen an arbitrary set of values produced by the broadcaster. If we run the same protocol without PKI, then the adversary will always be able to ensure non-faulty processors output the default value.
(2) By the reductions discussed in Section 4, it suffices to prove the theorem for BA.
(3) Theorem 6.2 was originally proved by Lamport, Shostak and Pease []. Here we describe (a slight modification) of a simple and more instructive proof by Berman, Garay and Perry [], which is known as the *Phase-King protocol*.
(4) **Gradecast**. To explain the main protocol, we first consider a simple two step subprotocol called *Gradecast*. Just as for BA, each processor receives an input $v \in V$. Now, though, they have to 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.

Gradecast: The instructions for processor i.

Step 1. Send v to all processors.

Step 2. If $n - f$ distinct processors sent b to i in Step 1, then send b to all processors.

Output: If $n - f$ distinct parties sent b to i in Step 2, then output b with grade 2. Otherwise, if $f + 1$ distinct parties sent b to i in Step 2, then output b with grade 1. Otherwise, output v with grade 0.

(5) The protocol is certainly simple. Let us examine its properties. First, it is easy to see that it satisfies validity:

- **Validity$^+$**. If all non-faulty processors have the same input value v, then they all output v with grade 2.

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

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

So, if a non-faulty processor outputs b with grade 2, then it *knows* all non-faulty processors have output b.

(7) To prove Knowledge of Agreement, we first prove a weak agreement property:

- **Weak Agreement**. If i is non-faulty and sends b in Step 2, then no non-faulty processor sends $b' \neq b$.

Proof. It cannot be the case that one non-faulty party is sent b by $n - f$ processors in Step 1, while another is sent $b' \neq b$ by $n-f$ processors, because any two sets of $n-f$ processors have least $n - 2f \geq f + 1$ processors in the intersection. At least one of those must be non-faulty, but non-faulty processors send only one value in Step 1.

(8) **Proof of Knowledge of Agreement**. If a non-faulty processor outputs b with grade 2, then it is sent b by at least $n - f$ processors in Step 2. This means all non-faulty processors are sent b by at least $n - 2f \geq f + 1$ processors in Step 2. Moreover, from the Weak Agreement property, no non-faulty processors can be sent $b' \neq b$ by $f + 1$ processors in Step 2.

(9) **Where has this got us?** So far, we have a simple protocol which, if all non-faulty processors start with the same input, produces a situation in which they all know that was the case. 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 basic idea is as follows:

- We run $f + 1$ *rounds*, each of which has a different 'leader'.
- Processors start each round with a certain value – at the start of the first round this is just their input.
- Each round 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 shares their value v. Each non-faulty processor changes their value to v unless they outputted the Gradecast instance for this round this grade 2.

(10) If we proceed in this way, then it is easy to see Validity will be satisfied. If all non-faulty processors start with the same value v, then it follows inductively that they will output each Gradecast instance with $(v, 2)$, and will not change their value upon hearing from the leader.

(11) It is also easy to see that after any round with a non-faulty leader (and then at all subsequent rounds), all non-faulty processors will have the same value: If a non-faulty processor does not change their value to that of the non-faulty 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.

(12) **The instructions for processor** i.

```
$v[-1] = v$
For $t = 0$ to $f$:
    Timeslots $4t, 4t + 1$:
        Run Gradecast with input $v[t - 1]$.

    Timeslot $4t + 2$:
        $(v[t], \text{grade}[t]) := \text{Gradecast}(v[t - 1])$.
        If $i == t$: send $v[t]$ to all processors.

    Timeslot $4t + 3$:
        If grade[$t$] < 2 then $v[t]$ := processor $t$'s reported value.
    Output $v[f]$.
```

(13) **Proof of Validity**: This follows from the Validity property of Gradecast and the fact that, in each round, the grade will be 2, meaning that the leader's value will be ignored.

(14) **Proof of Agreement**: Consider the first round with a non-faulty leader. From the Knowledge of Agreement property, all non-faulty processors will either switch to the leaders's value, or already have that value with grade 2.

(15) **An important takeaway**. 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 setting*, where solving BA is impossible when $f \geq n/3$ even when a PKI is given. In this context, a version of Weak Agreement becomes one of our central tools:

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

7 STATE MACHINE REPLICATION

7.1 Defining SMR

(1) So far, we have considered the consensus problems BA and BB. Here, we introduce another consensus problem called *State Machine Replication* (SMR). Very roughly, SMR is the problem that blockchain protocols are designed to solve: Clients send in a sequence of transactions of their choosing and the processors implementing the SMR protocol have to agree on an order in which to implement those transactions.

(2) In this section, we describe one formalisation of SMR, but the reader should be aware that there are a number of ways SMR can be formalised, and that (unlike BA and BB) SMR is often treated somewhat informally in the literature, i.e. it is often left unspecified precisely which formalisation is being used. Later, we will discuss other approaches to formalising SMR, and how the approach taken impacts questions such as the sense in which blockchain protocols like Bitcoin solve SMR.

(3) In the previous sections, we have considered protocols for solving BA and BA, both with and without a PKI. In part, interest in the case that no PKI is available is historical – the results of Section 6 are classics of the literature and it would seem a serious omission not to include them in any introduction to the area. Permissioned[10] SMR protocols, on the other hand, are normally considered in the context that a PKI is given. So, in the case of SMR protocols, we will deal only with the case that a PKI is available.

(4) Two principal differences between SMR and BA or BB are:
- SMR involves a second kind of protocol participant, referred to as *clients*. Roughly, clients do not actively participate in the process of reaching consensus, but are able to submit *transactions* (sometimes called *requests*) to the processors carrying out the consensus protocol, and must be reliably informed by the processors when those transactions have been implemented.
- SMR does not require processors to give a single output, but rather to produce an output sequence that grows in length as clients submit more transactions.

[10] By a 'permissioned' protocol we mean one such as those considered here, in which all participants are known to each other from the start of the protocol execution. This stands in contrast to 'permissionless' protocols, such as Bitcoin.

(5) **SMR - the setup**. Some of the choices below are somewhat arbitrary and have little impact on the consensus problem defined.
- We consider two kinds of protocol participant: a set of n *processors* $\{0,\dots,n-1\}$ and a finite set of *clients* $\{0,\dots,m-1\}$.
- We assume given a PKI for the set of all clients and processors, i.e. each client and each processor begins the protocol knowing the names and public keys of all participants, together with their own private key.
- Processors and the authenticated communication channels between them are formalised as before. All pairs of processors have communication channels between them.
- For each client a and for each processor i, there may exist an authenticated communicaiton channel, i.e. there may exist communication channel $\{a,i\}$ such that only i can send messages to a and only a can send messages to i on the channel $\{a,i\}$. In the synchronous setting, messages sent at time t are received at time $t+1$. We assume that for each client a there exists at least one non-faulty processor i for which there exists a channel $\{a,i\}$.
- The only messages that clients can send are *transactions*. We assume every transaction sent by a client is signed by the client.
- Clients can send arbitrary (signed) transactions on their communication channels (i.e. the protocol cannot dictate which transactions are sent by clients and must be able to deal with arbitrary signed transactions sent by clients), subject only to the constraint that clients send each transaction at most once to any given processor – one can suppose different instances of what would be the same transaction are numbered to distinguish between them.
- A *confirmation* for the transaction tx is the message tx signed by one of the processors (a client will require $f+1$ confirmations for a given transaction – see below).

(6) **SMR - the requirements**. Non-faulty processors must maintain an ordered log (sequence) of client transactions. Let $\texttt{log}_i^t$ denote the log of processor i at timeslot t. We write $\texttt{log}_i^t \preceq \texttt{log}_j^{t'}$ to denote that the first log is a prefix of the second (or the same). We require:
- **Consistency**. For any non-faulty i,j and any t,t', either $\texttt{log}_i^t \preceq \texttt{log}_j^{t'}$ or $\texttt{log}_j^{t'} \preceq \texttt{log}_i^t$.
- **Liveness.** If a non-faulty i receives the transaction tx, then tx must eventually be included in the logs of all non-faulty processors.

(7) Processor i regards a transaction as *confirmed* once the transaction enters its log. So, roughly, the consistency and liveness conditions say that all non-faulty processors must agree on the set of confirmed transactions (although confirmation for one may be slower), and any transaction that is received by a non-faulty processor must eventually be confirmed.

(8) **Lazy clients**. There is one final condition that an SMR protocol must satisfy according to our present formalisation (later we will consider relaxations of this condition). Roughly, we want to formalise the idea that clients should not need to actively observe the entire protocol over time to know when their transactions are confirmed. Clients should be able to go offline, then return to check what messages they have received and be sure of their confirmed transactions. To formalise this, we require that clients should receive $f+1$ confirmations for a given transaction (signed by distinct processors) iff the transaction is confirmed by a non-faulty processor.

7.2 For which f is SMR possible in the synchronous setting?

(1) An immediate consequence of the *lazy client* condition in paragraph 8 of Section 7.1 is that SMR cannot be solved when $f \geq n/2$.

(2) When $f < n/2$, however, it is easy to modify the Dolev-Strong protocol described in Section 5 to give a protocol for SMR:
- We run repeated instances of the Dolev-Strong protocol with processor i mod n acting as broadcaster (or 'leader') in the ith instance.
- When a non-faulty processor is leader, they order the set of all client transactions they have received but which are not yet in their log, and broadcast this ordered sequence of transactions T as their value for agreement.
- Non-faulty processors ignore suggested values for agreement that are not sequences of client transactions that have not yet been added to their log.
- If the Dolev-Strong protocol decides $\perp$, then non-faulty processors do not extend their log. If the Dolev-Strong protocol agrees on the value T, then non-faulty processors append T to their log.
- When a non-faulty processor adds any transaction to its log, it produces a confirmation for that transaction and sends that confirmation to all processors and all clients with which it has a communication channel.
- Non-faulty processors also resend all transaction confirmations received: When i first sees a confirmation for tx produced by j (i.e. i receives the client transaction tx signed by processor j), i sends that confirmation to all processors and all clients with which it has a communication channel.

(3) The protocol above might not seem like the most efficient way to solve SMR. Later, we will see a very elegant protocol (essentially Tendermint []) which is designed to operate in the *partially synchronous* setting, where message delivery is less reliable than in the synchronous setting, and which requires fewer rounds of communication per 'leader'. Since this protocol works in the partially synchronous setting, it also works in the synchronous setting.

(4) Note that a solution to SMR (for which we assume a PKI and require $f < n/2$) in the synchronous setting can also be used to solve BB (and thus BA):
- Let $\perp$ denote a 'default' element of V.
- We run an instance of SMR in which the number of clients is equal to the number of processors, and in which clients have channels to all processors. For each $i \in \{0, \dots, n-1\}$, we have processor i also control client i.
- In the first timeslot we have the broadcaster, i say, send a single transaction from client i, which is a signed version of its input.
- In the second timeslot, each non-faulty j sends a signed 'did not receive' transaction from the client they control if they do not receive a transaction from client i.
- We wait until the SMR protocol confirms a signed value from the broadcaster, in which case all non-faulty processors output that value, or else confirms $f + 1$ 'did not receive' transactions (signed by distinct processors), in which case all non-faulty processors output the default value $\perp$.

8 DEFINING THE PARTIALLY SYNCHRONOUS SETTING

(1) There are two standard ways of relaxing the synchronous setting. These are the *partially synchronous* and the *asynchronous* settings. In this section, we describe the partially synchronous setting.

(2) In the version of the synchronous setting we considered in previous sections, a message sent at time t was always delivered at time $t + 1$. This is sometimes called the 'lock step model'. As mentioned in Section 3, an alternative is to suppose processors are told a value Δ such that any message sent at time t will be delivered by time $t + \Delta$. We did not differentiate between

these models, because all of the proofs we described are easily modified to work for either model.

(3) In the partially synchronous setting, the rough idea is that processors are told a bound Δ which will hold when network conditions are good. Sometimes, however, network failure may occur, meaning that messages may be delayed for time longer than Δ. Protocols now have to survive extended periods of network failure and produce outputs whenever good network conditions hold for sufficiently long. In the context of SMR, for example, we cannot demand that new transactions be confirmed during periods of network failure, but we can demand that 'consistency' is maintained at such times. Then we require that new transactions be confirmed whenever network conditions are good for sufficiently long.

(4) **A technical trick**. A technically slick (and standard) way to formalise this [], is to posit the existence of some unknown *Global Stabilisation Time* (GST), and specify that the bound Δ holds after GST: Any message sent at time t is received by time $\max\{t, \text{GST}\} + \Delta$.

(5) **Justifying the trick**. At first sight, the condition above may seem too strong – why should there be some point after which message delivery is always reliable? Did we not want to consider a more general scenario in which network failures may come and go, and in which protocols are required to produce outputs whenever network conditions are good for *sufficiently long*. It is not hard to see, though, that any protocol which functions under the apparently easier GST model, will also function under the latter more general conditions. To see this, we give an explanation which is somewhat informal because we have not formally defined the more general model. To make things concrete, consider the case of a protocol for BA. If the protocol ever violated agreement or validity under the more general model, then it would also do so under the GST model (by an appropriately large choice of GST). If the protocol satisfies termination under the GST model, then for any choice of n, any choice of inputs, any choice of behaviour for the faulty processors and any choice of GST, there exists T, such that all non-faulty processors terminate within time $GST + T$. So, the protocol also terminates under the more general model so long as network conditions are good for sufficiently long.

(6) **Clock synchronisation**. In the synchonous model, all processors began with perfectly synchronised clocks which ran at identical speeds, meaning that all processors could execute the instructions for some timeslot at the same time as each other. We consider two versions of the partially synchronous model with respect to clock synchronisation: Clocks can be synchronised or not. In both versions, we'll assume that all processors have identical clock speeds (of course it is also interesting to relax this condition). When clocks are synchronised this means that all processors begin the protocol at the same time as each other. When clocks are not synchronised, this means that processors might begin the protocol at different times and do not have clocks that agree on some global notion of what time it is (or what timeslot should presently be executed). When clocks are not synchronised, we will assume that all processors begin before GST.

(7) **Another formalisation – unknown** Δ. In fact, there are two versions of the partially synchronous setting that are standard (beyond issues of clock synchronisation). An alternative approach is to suppose that the bound Δ always holds but is not given to the processors as input (so that the protocol must function for any finite Δ without knowing this value). Let us call this the 'unknown Δ' setting.

(8) **The equivalence**. So long as we are not concerned with efficiency (just when protocols are possible), these two versions of the partially synchronous setting are equivalent. If a protocol works for the unknown Δ model, then it works under the GST assumption because, for any execution of the protocol, messages will always be delivered within time $\Delta' :=\text{GST}+\Delta$ after

sending. In the other direction, suppose we have a protocol which functions under the GST model. To turn this into a protocol that functions under the unknown Δ model, we can give processors the input $\Delta = 1$, and then insert increasing intervals between the timeslots at which processors execute their instructions, e.g. processors might implement the instructions for time t at time 2^t (efficiency considerations aside). Although the processors do not know the actual value Δ, the time between timeslots at which instructions are implemented is eventually larger than Δ.

(9) Unless stated otherwise, when we refer to the partially synchronous model we will mean the GST version of the model from this point forward.

(10) **BB is not possible when $f \geq 1$ in the partially synchronous model**. Suppose clocks are synchronised. Towards a contradiction, suppose we have a working protocol:
- Let us say two executions of the protocol are *indistinguishable as far as i is concerned* (until a certain timeslot) if that processor receives the same inputs and receives the same messages on the same channels at the same timeslots in both executions.
- Suppose $n \geq 2$ and consider an execution of the protocol in which all processors are non-faulty. Suppose all messages are delivered at the timeslot after sending, except that we withhold delivery of messages from the broadcaster (and delay GST) until some other processor terminates. Some processor other than the broadcaster must eventually terminate because such an execution is indistinguishable (for as long as we withhold the broadcaster's messages) as far as they are concerned from an execution in which GST= 0 but the broadcaster is faulty and does not send messages.
- Let t be the first timeslot at which a processor i other than the broadcaster terminates. We define GST:= $t + 1$.
- Let v be i's output. Now we reach a contradiction, because there are executions such as those described above in which the broadcaster's input is not v.

9 FOR WHICH f ARE BA AND SMR POSSIBLE IN THE PARTIALLY SYNCHRONOUS SETTING WITH PKI?

9.1 The negative result

In this section, we prove the following theorem of Dwork, Lynch and Stockmeyer [].

Theorem 9.1. *Consider the partially synchronous setting (with or without PKI) with Byzantine faults. No protocol can solve BA or SMR when $f \geq n/3$.*

(1) We give a proof for BA, which is easily modified to deal with SMR. In fact, we will show in section 9.2 that any protocol solving SMR in the partially synchronous setting (with PKI) can be used to solve BA.

(2) Suppose clocks are synchronised (to give the strongest possible impossibility result). Suppose $f \geq n/3$ and that we have a working protocol:
- Suppose $n = 3$, $f = 1$.
- Recall the definition of *indistinguishable* executions from paragraph 10 of Section 8.
- Suppose processors 0 and 1 are non-faulty, and have inputs A and B respectively. Suppose processor 2 is faulty.
- Suppose any message sent at time t is received at time $t + 1$, except that we withhold delivery of any messages between processors 0 and 1 (and delay GST) until the non-faulty processors have both terminated.
- Suppose that, in determining the messages it sends to 0, processor 2 simulates a non-faulty processor with input A that does not receive any messages from 1. In determining the

  messages it sends to 1, processor 2 simulates a non-faulty processor with input B that does not receive any messages from 0.
  - Processor 0 must eventually terminate and output A because (for as long as we withhold messages from 1) this execution is indistinguishable as far as they are concerned from an execution in which GST= 0 but 1 is faulty and does not send messages.
  - Similarly (and symmetrically), 1 must eventually terminate and output B.
  - Now we reach our contradiction because we can define GST to be the first time at which 0 and 1 have both terminated and observe that agreement is violated for this execution.

(3) In the above, we supposed $n = 3$, $f = 1$. To obtain a proof for the general case, proceed as in paragraph 10 of Section 6.1.

9.2 Reducing BA to SMR in the partially synchronous setting

Suppose we have a protocol solving SMR (for $f < n/3$). Then we solve BA as follows:

- We run an instance of SMR in which the number of clients is equal to the number of processors, and in which clients have channels to all processors. For each $i \in \{0, \dots, n-1\}$, we have processor i also control client i.
- At time $t = 0$, each non-faulty processor i sends a single transaction from client i, which is a signed version of its input.
- Each non-faulty processor carries out the SMR protocol until it has confirmed at least $n - f$ signed values from distinct processors. At this point, it outputs the majority value (breaking ties in some arbitrary fixed fashion).

9.3 Simplified Tendermint: Solving BA and SMR for $f < n/3$ in the partially synchronous setting with PKI and synchronised clocks

In this section, we prove the following theorem.

Theorem 9.2. *Consider the partially synchronous setting with PKI, synchronised clocks and Byzantine faults. There exist protocols solving SMR and BA whenever $f < n/3$.*

(1) A protocol solving BA in the partially synchronous setting with PKI was first given by Dwork, Lynch and Stockmeyer [] (who also proved the same result without PKI, but with a more complex proof). The route we take is to describe a protocol for SMR. By the reduction described in Section 9.2, this suffices to give the result for BA.

(2) There is a long and complicated history of SMR protocols in the partially synchronous setting. The most famous protocol is probably PBFT []. We will describe a version of PBFT later. To begin with, however, we describe an easy protocol which is a simplified version of Tendermint [].

The informal intuition.

(1) To make the protocol as simple as possible, we will not worry about efficiency considerations, such as the number of messages sent and message length. We also assume synchronised clocks (this is not true for the original Tendermint protocol). Later we will show how to relax this condition. We will also subdue mention of clients. We will just assume that clients send in transactions and that non-faulty processors send transaction confirmations to clients as described in the example of Section 7.2.

(2) **Rounds and leaders**. We describe a protocol which is rather like that from Section 6.2, in the sense that the instructions are divided into successive rounds, each with a different *leader*. The job of the leader is to suggest a new set of transactions to be added to the common log.

(3) **Transaction blocks**. To present the protocol in the same flavour as Tendermint, we consider *blocks* of transactions. A block of transactions is an ordered sequence of transactions, but also contains other data such as a single hash[11] pointer to a unique *parent* block.

(4) **Ancestors**. The parent B' of a block B is also called an *ancestor* of B, and all ancestors of B' are also ancestors of B. If B' is the parent of B, then we also say that B is a *child* of B'. All processors begin with an agreed 'genesis' block, which contains no transactions and has no ancestors. All other blocks must have the genesis block as an ancestor, i.e. non-faulty processors will not recognise blocks that are not of this form. If B' is an ancestor of B, we also say that B *extends* B'. Two blocks are *incompatible* if neither is an ancestor of the other.

(5) **Quorum certificates**. We also consider (signed) *votes* on blocks at various stages of the protocol. If $n-f$ processors produce votes for a particular block (and corresponding to a specific round of voting), we call this set of votes a *quorum certificate* (QC). We only ask for $n-f$ votes, because faulty processors cannot be relied on to vote. Note that a trick similar to that used in Section 6.2 (paragraph 15) applies here: If non-faulty processors only vote for one block in each round of voting, then two different blocks cannot both receive QCs in the same round. This follows since:

$$\boxed{(\dagger_0)\ \text{Two sets of } n-f \text{ processors must have at least one non-faulty processor in common .}}$$

To see that $(\dagger_0)$ holds, note that any two sets of $n-f$ processors must have at least $n-2f$ processors in common. They must then have at least one non-faulty processor in common, since $n-2f>f$.

(6) **A simple (but bad) attempt.** One approach would be to proceed as follows:

(a) The leader for round r is processor r mod n.

(b) Each round is of length 2Δ. When the round begins (at time t say), the leader produces and sends a new block of transactions to all processors.[12] This block includes the round number and should extend the highest confirmed block (i.e. that corresponding to the highest round number rather than that with the most ancestors) amongst those they have seen.

(c) At time $t+\Delta$, non-faulty processors each consider the first (signed) block[13] (if any) they have received from the round leader. If it extends the highest confirmed block they have seen, then they produce a (signed) vote for that block corresponding to round i and send that vote to all processors.

(d) A processor considers a block confirmed once they see a QC for that block.

(7) In the partially synchronous setting, however, it is especially easy to see that the method above does not produce consistency, i.e. incompatible blocks might become confirmed. A block B might be confirmed in round r, for example, but it might be the case that fewer than $f+1$ of the non-faulty processors see the QC . Then a block B', which is incompatible with B, might be proposed at round $r+1$ and (potentially with the help of faulty votes) might also become confirmed.

(8) **Two stages of voting**. What happens if we consider two stages of voting on the block instead? Let us suppose that, if they see a QC on the block B produced in 'stage 1' (say) of round r, processors then produce a 'stage 2' vote, and it is a QC of stage 2 votes that means

[11] We assume the reader has some familiarity with hash functions. In these notes, hash functions will only be used to provide short pointers. Rather than worrying about how to model them, we will simply *decree* that hash pointers are unique, i.e. we will restrict the set of executions considered to those in which all hash values considered are unique.

[12] As in Section 6.2, it is convenient here to suppose that whenever i sends a message to all the other processors, it also sends that message to itself. While this is not formally possible according to the framework described in Section 3, it is clear that processors can simulate such a scenario in the instructions they execute.

[13] We'll assume throughout that processors ignore messages that aren't properly formed, with the right round number etc.

confirmation. The problem with the protocol from (6) was that a block might be confirmed, but non-faulty processors might not see the relevant QC. With our new approach, it is at least the case that if B is confirmed then at least $n - 2f$ $(\geq f + 1)$ non-faulty processors must have seen the stage 1 QC for the block (because we need at least $n - 2f$ non-faulty processors to vote in the second round to produce confirmation). Suppose we now instruct processors who have seen the stage 1 QC to 'lock' on the block B, which means that they do not yet consider it confirmed, but now will not vote for blocks incompatible with B (while the lock is in place). It is easy to see that this new rule produces consistency:

> ($\dagger_1$) If the block B is confirmed in round r, then at least $n-2f$ non-faulty processors become locked on B. Since at least $n - 2f$ non-faulty processors are required to produce any QC, this makes it impossible for any block incompatible with B to receive a QC in subsequent rounds.

(9) **Liveness**. The only danger now is that we threaten liveness. When a non-faulty processor becomes locked on a block B, it is possible that $\leq f$ other non-faulty processors are already locked on a previous and incompatible (and necessarily unconfirmed) block B'. With the non-faulty processors split, we now cannot ensure that any future blocks become confirmed. This does not seem like much of a problem, though, because, once the global-stabilisation-time (GST) comes, we can ensure that the processors that are locked on B' will see the stage 1 QC for B. So, we can just agree that, when a processor is locked on a specific block corresponding to a certain round, they will unlock if they subsequently see a QC corresponding to a greater round. Note that this does not harm the argument in (†) above in any way, so the consistency argument still holds.

The formal details.

(1) **Blocks**. Let $\langle m \rangle_i$ denote the message m signed by i. A block is a message of the form $\langle \mathsf{T}, r, d \rangle_i$, where T is a sequence of transactions, $i \equiv r \bmod n$ and d is a hash value. We say that r is the round corresponding to the block. Ancestors and children etc. are defined as in paragraph 4 above.

(2) **Votes**. A vote v for the block B produced by i in round r is a message of the form $\langle d, r, j \rangle_i$, where d is the hash of B and j is the stage of voting. We say v is stage j vote by i. A stage j QC for a block B is a set of $n - f$ stage j votes for B, each by a different processor.

(3) **Well-formed blocks**. A processor will regard a block as *well-formed* if it is of the form described in paragraph 1 above and if it is able to verify that d is the hash of either the genesis block or a well-formed block which has received a QC (stage 1, 2, or both). Non-faulty processors will ignore blocks which they cannot verify are well-formed, so we will assume that all blocks referred to in the instructions below are are well-formed from this point on.

(4) **Automated message passing**. To ensure all processors receive the blocks they need to, even when those blocks are produced by the adversary, we suppose that (in addition to the instructions listed in paragraph 5 below) processors pass on messages when they first receive them, e.g. when processor i receives a vote by processor j, i sends that vote to all processors.[14]

(5) **The protocol**. All processors begin with their 'lock' set to be the genesis block, which corresponds to round 0. We consider the obvious ordering on QCs, in terms of round number

[14] This approach is clearly subject to abuse by the adversary who can then force non-faulty processors to send an unbounded number of messages. While we will not be concerned with matters of efficiency until we consider Hotstuff, the following approach could also be taken to prevent this issue. Processors only pass on messages in the following contexts: (a) When any non-faulty processor first sees that a block B has received a QC, it sends B to all processors; (b) For $j \in \{1, 2\}$, when any non-faulty processor first observes a stage j QC for B, it sends that QC to all processors.

and then stage number within the round.[15] By the 'highest block', we mean that with the greatest corresponding round number. The 'highest quorum' below a block B is the highest QC that corresponds to any block amongst B and its ancestors.

The instructions for i **at round** $r > 0$**, starting at** time $t = 3r\Delta$.
Time t. If i is the leader for round r (if $i \equiv r \bmod n$), then carry out the following. Let B' be the highest block that i has received and such that i has received a QC for B' (or let B' be the genesis block, if no QC has been seen). Produce and send to all processors a block B which is a child of B'.
Stage 1. Time $t + \Delta$. Consider the first block B corresponding to round r received by i (if any such block exists). If B extends i's lock, or else if the highest QC below B is as high or higher than i has previously seen (if it is higher they release their lock), then i sends a stage 1 vote for B to all processors.
Stage 2. Time $t + 2\Delta$. If i has seen a stage 1 QC for some block B corresponding to round r, then i sets B as their lock and sends a stage 2 vote for B to all processors.

(6) We have already proved consistency for this protocol. Liveness is not hard to prove either. Consider a round with non-faulty leader which begins at time t such that $t - \Delta$ is after the global-stabilisation-time (GST). Let B be the highest lock (i.e. with greatest round number) amongst all non-faulty processors at $t - \Delta$. The non-faulty leader will see B and a QC for B by time t (due to the automated message sending), and will then propose B' which has highest QC below it at least that of B. All non-faulty processors will produce votes for B' in stages 1 and 2 of the round, and the block will then be confirmed.

9.4 Chained Tendermint (basically Casper FFG [])

(1) If that protocol seemed simple, the next realisation is that we can use a trick to make it even simpler. In fact, we can get the extremely simple protocol from (6) in the previous section to work, if we just apply a new notion of confirmation together with locking.

(2) The basic idea is as follows. Rather than have two stages of voting in each round r, we have a single stage but we have the votes for that stage count in two ways. First of all, they count as a stage 1 vote for the block at round r. If that block extends a round $r - 1$ block with a QC, however, we *also* consider those votes as a stage 2 vote for that previous block. So now, when we see QCs corresponding to two compatible blocks at two successive rounds r and $r + 1$, we consider the block at round r confirmed.

(3) Terminology and conventions (on automated message passing etc) are the same as in Section 9.3. Here is the new protocol.

The instructions for round $r > 0$**, starting at** $t = 2r\Delta$.
Time t. The leader sends to all processors a block which is a child of the block with the highest QC it has seen.
Time $t + \Delta$. Each processor considers the first block B they receive corresponding to round r (if any). If B extends their lock, or else if the highest QC below B is as high or higher than they have previously seen, they: (a) Set their lock to be the block B' corresponding to the highest QC below B, (b) Send a vote for B to all processors.

(4) As stated above, when i sees QCs corresponding to two compatible blocks at two successive rounds r and $r + 1$, it considers the block at round r confirmed (together with all ancestors). The proofs for liveness and consistency are essentially the same as before, but now one requires two non-faulty leaders in a row for the liveness proof. [ADD PROOF]

[15] It is worth stressing that it is round number rather than chain length (i.e. number of ancestors) that we care about here.

9.5 Simplified PBFT: Solving BA and SMR for $f < n/3$ in the partially synchronous setting with PKI and without synchronised clocks

The version of Tendermint we presented in Section 9.3 was particularly simple because, unlike implementations of Tendermint in the wild, we assumed that processors have synchronised clocks. In this section, we consider how to remove this assumption. We will describe the method used by PBFT (which stands for Practical Byzantine Fault Tolerance), but the reader is invited to figure out for themselves how to implement a similar method that works for Tendermint.

The informal intuition.

(1) The original PBFT actually functions by having a single leader remain in charge of pushing through transaction confirmations until such a time as other processors vote to change leader (e.g. because they haven't been receiving messages from the leader). In times of synchrony this means that a single leader can remain in charge for an unlimited duration. We present a version here which is more in the spirit of 'blockchain', whereby leader rotation occurs every round. Specifically, the leader for round r is processor r mod n. As is common in the context of blockchain, we will also have the transactions be divided up into blocks (as in the protocol of Section 9.3), although the original PBFT does not work this way. Generally, our intention here is not to precisely reproduce PBFT, but rather to give a version which is as simple as possible and which illustrates the fundamental features of PBFT.

(2) We will use the same terminology and notiation as in Section 9.3, so see that section for the definition of a QC etc.

(3) **How to make processors execute rounds together?** If processors do not have synchronised clocks, how can we ensure that they come together to execute the instructions for a given round correctly? Let us suppose that a processor i is in round $r-1$ and 'wishes to enter' round r at some point of the protocol execution – we will consider the conditions under which a processor will wish to enter a round later. One way to proceed is as follows:
- When i wishes to enter round r, we have i send a signed '`round` r' message to all processors, which indicates that it wishes to enter round r. We also assume the same 'automated message passing' system as in Section 9.3. With respect to `round` r messages, this means that, for each j, the first time that i receives a `round` r message signed by j, i sends that message to all processors.
- At any time when i is in a round r' (such as $r-1$) and receives round $r > r'$ messages signed by $2f+1$ different processors it immediately enters round r.

If we proceed this way, then when i enters round r it knows that all non-faulty processors will also receive at least $2f+1$ `round` r messages within time Δ if we are after GST. If r was the highest round that any non-faulty processor had yet wished to enter, this means that all non-faulty processors will now be able to execute round r together.

(4) **An alternative to locks**. Upon entering round r, how does the leader know what block to produce? Using locks like we did in our presentation of Tendermint is one approach, but PBFT uses a different method:
- Processors include in their `round` r message the highest round number for which they have seen a stage 1 QC, together with a copy of the corresponding QC.
- Once the leader begins round r, they take the highest round number r' specified in any of the $n-f$ `round` r messages received, and let B' be the block for round r' which has received a QC. They produce a child of B' and send this block B to all processors, attaching the $n-f$

round r messages. We call σ which is the set of $n - f$ round r messages a 'justification' for B.[16]

- We then carry out two stages of voting. Processors will vote for B in the first stage if (it is well-formed and) the attached justification σ is 'correct': The justification σ is correct for B if B is a child of the highest block B' that is proved to have a QC by σ. As in Section 9.3, by 'highest' block, we mean that with the greatest round number. Processors will vote in the second stage if they see a stage 1 QC for B.

(5) **Consistency**. The details above already suffice to establish consistency. Suppose a block B for round r becomes confirmed, i.e. receives a stage 2 QC. This means there exists a set X of at least $n - 2f$ non-faulty processors who see a stage 1 QC for B while in round r. Towards a contradiction, suppose there exists a least $r' > r$ at which a leader sends out a justification which is correct for a block B' that is incompatible with B. Since this justification contains $n - f$ round r messages, it must contain a message from a processor in X, which means that σ cannot be correct for B'.

(6) **Timeouts**. When should a processor 'wish to enter' round r? Suppose that i is in round r. Since the leader may be faulty and might not send any messages, i will eventually have to try and enter round $r + 1$. We proceed as follows:

- When i enters round r it sets a timer. If the timer reaches 4Δ before i wishes to enter a higher round, then i wishes to enter round $r + 1$ at this point.
- If i is in round r and sees a stage 2 QC for any block corresponding to a round $r' \geq r$, then i wishes to enter round $r' + 1$. This condition is not necessary, but makes the protocol *optimistically response* – we will discuss this later.

The formal details.

(1) Blocks, well-formed blocks and votes are all defined as in Section 9.3, and we make the same 'automated message passing' assumptions as there. Again, $\langle m \rangle_i$ denotes the message m signed by i.

(2) All processors begin with the same agreed 'genesis block', which corresponds to round 0. It is convenient to regard the empty set as a stage 1 QC for the genesis block.

(3) A round r message is of the form $\langle r, r', \rho \rangle_i$, where ρ is a stage 1 QC for round r'. A set of $n - f$ round r messages σ is called a round r *justification.* Given a round r justification σ, let r' be the greatest such that there exists a round r message in σ of the form $\langle r, r', \rho \rangle_i$. If ρ is a QC for B' and if B is a round r block which is a child of B', then we say that σ is *correct* for B.

(4) **The protocol**.

The instructions for entering rounds.

All processors begin in round 0 and start by *wishing to enter* round 1.

A processor i wishes to enter round $r > 1$ the first time it is in a round $< r$ and either: (a) i sees a stage 2 QC for a block corresponding to round $r - 1$, or (b) i is in round $r - 1$ and its timer reaches 4Δ.

When i first wishes to enter round r it sends a round r message $\langle r, r', \rho \rangle_i$, where r' is the greatest round for which i has seen a block with a stage 1 QC, and where ρ is a stage 1 QC for round r'.

Processor i enters round r the first time it is in a round $< r$ and it has received $n - f$ round r messages.

The instructions for i in round $r \geq 1$.

[16] It is clear that the protocol could be made more efficient here by only including the highest QC in the justification, but we will not worry about matters of efficiency until later.

Start the timer from 0.

Block production. If i is the leader for round r (if $i \equiv r \bmod n$), then carry out the following. Let σ be the round r justification received by i. Form a block B such that σ is correct for B. Send B with σ attached to all processors.

Stage 1 voting. Wait to receive a first block B corresponding to round r and let σ be the justification attached to B. If σ is correct for B, then send a stage 1 vote to all processors.

Stage 2 voting. Wait to receive a stage 1 QC for B. Then send a stage 2 vote for B to all processors.

(5) **Proving liveness**. We have already proved consistency for this protocol. Liveness is not hard to prove either. Let r be a round with non-faulty leader such that no non-faulty processor wishes to enter any round $\geq r$ before GST. The use of timeouts ensures that there exists a first $t_0 \geq GST$ at which some non-faulty processor wishes to enter round r. Let $t_1 \geq t_0$ be the first point at which any non-faulty processor enters round r. The leader for round r will enter round r by time $t_1 + \Delta$ and will send a block B to all processors, with a justification σ attached which is correct for B. All non-faulty processors will send stage 1 votes for B by time $t_1 + 2\Delta$, and will send stage 2 votes for B by time $t_1 + 3\Delta$. This means that all non-faulty processors will see a stage 2 QC for B by time $t_1 + 4\Delta$.