Yale NUS College

Formally Verifying Accountable Byzantine Consensus

Karolina Grzeszkiewicz

Capstone Final Report for BSc (Honours) in

Mathematical, Computational and Statistical Sciences

Supervised by: Dr. Ilya Sergey

AY 2022/2023

Yale-NUS College Capstone Project

DECLARATION & CONSENT

1. I declare that the product of this Project, the Thesis, is the end result of my own work and that due acknowledgement has been given in the bibliography and references to ALL sources be they printed, electronic, or personal, in accordance with the academic regulations of Yale-NUS College.

2. I acknowledge that the Thesis is subject to the policies relating to Yale-NUS College Intellectual Property (Yale-NUS HR 039).

ACCESS LEVEL

3. I agree, in consultation with my supervisor(s), that the Thesis be given the access level specified below: [check one only]

[✓ Unrestricted access]
Make the Thesis immediately available for worldwide access.

o Access restricted to Yale-NUS College for a limited period
Make the Thesis immediately available for Yale-NUS College access only from ____________ (mm/yyyy) to ______________ (mm/yyyy), up to a maximum of 2 years for the following reason(s): (please specify; attach a separate sheet if necessary):
____.

After this period, the Thesis will be made available for worldwide access.

o Other restrictions: (please specify if any part of your thesis should be restricted)
____
____

Karolina Grzeszkiewicz, Saga College
Name & Residential College of Student

[illegible]
Signature of Student

03.04.2023
Date

Dr. Ilya Sergey [illegible]
Name & Signature of Supervisor

03.04.2023
Date

Acknowledgements

I am forever grateful to the following people for their support and influence throughout my undergraduate studies.

To my capstone advisor, Prof Ilya Sergey, for his valuable advice, unlimited patience, engaging lectures and office hours, challenging programming assignments, and finally, for piquing my interest in formal verification. His generous guidance and passion have truly inspired my confidence and ambition to pursue computer science.

To Prof Olivier Danvy for showing me that programming is nothing more than proving, and initiating me into the beauty of functional programming.

To George and Qiyuan, for their insights and feedback throughout the process of working on this project.

To *moja miłość* for always radiating kindness, warmth, and energy, for being and listening.

To Yale-NUS friends and communities for many memorable experiences: bhangra, philosophy society, my suitemates.

To my parents for their unwavering support and openness, and to my sister for being such a source of joy.

YALE-NUS COLLEGE

Abstract

B.Sc (Hons)

Verifying Accountable Byzantine Consensus

by Karolina GRZESZKIEWICZ

Modern Byzantine consensus protocols can be used to achieve agreement in the presence of a bounded number of faulty nodes trying to corrupt the network, yet they fail to detect and hence punish or disincentivise Byzantine behaviour. When combined with any byzantine consensus protocol, the Accountable Byzantine Consensus (ABC) transformation guarantees both consensus and accountability. Given the sensitive applications of consensus, we would like to have a formal model of the protocol and of the possible deviations from it, namely the byzantine behaviour. Given such a model, we can formally prove that the desired properties of the protocol hold for the model.

In this report we present two formalisations of the ABC protocol: one in TLA+, a formal modelling tool, and a more general model of the network and protocol semantics, including a faithful representation of Byzantine behaviour. We use the latter to prove the soundness of the protocol by induction. Finally, we draw insights regarding modeling Byzantine behaviour and the notion of accountability offered by ABC.

Contents

Chapter 1

Introduction

1.1 Byzantine Consensus

The problem of Byzantine Consensus goes back to "The Byzantine Generals Problem" coined by Leslie Lamport (Lamport et al., 2019). Suppose a group of Byzantine army generals are camped around a city with their troops and they want to decide whether all of them should invade the city or retreat. They communicate via messengers that can take arbitrarily long to deliver messages, and some of the generals might be traitors intentionally trying to prevent consensus or push through a malicious plan. In computer science terms, the generals correspond to nodes in a network, and the decision to be made is usually related to storing a consistent state or log of events among the honest nodes in the network. We call the dishonest nodes *Byzantine*.

We need a reliable consensus mechanism to build large distributed services in which strangers cooperate with each other according to predefined rules, perform operations, and store logs mirroring the state or history of the system. Examples of such services include distributed databases, distributed filesystems, and blockchains. Clearly, the consensus mechanism has to be immune to the presence of adversaries who attempt to corrupt the system.

The protocols designed in response to the Byzantine Consensus Problem are referred to as BFT protocols (Byzantine Fault Tolerance), and the most famous and widely used ones include PBFT (Castro and Liskov, 1999), HotStuff (Yin et al., 2019), and Nakamoto Conensus (Nakamoto, 2008). The first two protocols are designed to operate under partial synchrony, but remain safe under asynchrony. The state-of-the-art protocols have communication complexity of $O(n^3)$ or $O(n^4)$, where n is the number of nodes in the network and communication complexity measures the amount of communication required.

It is known that consensus can only be achieved if at most $t_0 = \lceil \frac{n}{3} \rceil - 1$ processes are Byzantine (Lamport et al., 2019).[1] Assuming there are at most t_0 Byzantine nodes, a protocol that solves the Byzantine consensus problem satisfies the following:

1. *Termination:* Every correct process eventually decides a value.
2. *Agreement:* All correct processes decide the same value.
3. *Validity:* If all correct processes propose the same value, only that value can be decided by a correct process.

Agreement and *Validity* are *safety* properties, which roughly means that they assert that "nothing bad happens," and *Termination* is a *liveness* property meaning that "something good happens eventually."

1.2 Accountability in Byzantine Consensus

Satisfying the above properties does not necessarily guarantee the safety and trustworthiness of a distributed service. Firstly, the *agreement* property holds only if at most t_0 nodes are malicious. Secondly, the Byzantine consensus problem assumes the network is perfectly connected, which is not always the case.[2]

[1]We will refer to t_0 in ensuing chapters.

[2]For instance, in the Bitcoin blockchain communication is carried out via a peer-to-peer (P2P) network in which a malicious user can isolate a specific node within the network and hijack its communications.

One way to cope with such unlikely yet possible attacks is to disincentivise the attackers by either making attacks more expensive or identifying and punishing the attackers, for instance by excluding them from the network. The latter is what we refer to as accountability.

The research development that is central to this capstone project is the publication of "As easy as ABC: Optimal (A)ccountable (B)yzantine (C)onsensus is easy!" (Civit et al., 2022) which presents a transformation of any BFT protocol into an accountable one. The motivation is to achieve consensus when possible given the known constraints, and to identify at least $t_0 + 1$ processes as guilty whenever correct processes disagree. In other words, in addition to *Agreement*, *Validity*, and *Termination*, an ABC-transformed BFT protocol satisfies *Accountability:* "If two correct processes decide different values, then every correct process eventually detects at least $t_0 + 1$ faulty processes and obtains a proof of culpability of all detected processes." (Civit et al., 2022)

The appeal of ABC is in its simplicity, applicability to any BFT protocol, and optimal communication complexity. ABC incurs only $O(n^2)$ extra communication with at most t_0 faulty processes, and $O(n^3)$ otherwise. Hence, it does not increase the communication complexity of most BFT protocols.

The key part of ABC is the Accountable Confirmer algorithm designed to be executed after the BFT protocol, with the values decided by the processes in the BFT protocol being submitted in the initial phase of the Accountable Confirmer.

1.3 Problem Statement

Distributed systems are known for being notoriously difficult to reason about, as their complexity grows with the number of nodes in the network.[3] Given how critical the applications of ABC are, there are strong motivations to gain

[3]The number of nodes can be counted in millions for blockchain networks.

formal guarantees of its correctness (*safety* and *liveness*) and other important properties by applying formal verification techniques.

First, we model the protocol in TLA+, designed for modelling and lightweight verification of concurrent and distributed systems (Lamport, 2003). The language of TLA+ is very expressive and thus used in both academia and industry. Specifications are written in the formal language of first-order logic where a protocol transition is described by its precondition and postcondition. The TLA toolbox also enables bounded model checking, which explores all possible executions of the protocol up to some depth in a breadth-first manner. Model checking requires setting the parameters of the system, such as the number of nodes, to specific values. For the above-mentioned reasons, model checking is computationally expensive and unsound, possibly failing to find violations of invariants when it does not reach the depth at which they occur. We use TLA+ primarily for initial exploration of the protocol logic and its properties.

However, to enable sound verification of ABC, in particular when the protocol executes in the presence of Byzantine adversaries exhibiting arbitrary behaviour, we need a more abstract model of the protocol and the network in which it executes. Such a model should consist of a global state specified inductively in terms of transitions enabled by the network. The transitions are to model asynchronous message passing, where the messages sent by honest nodes are determined by the protocol. The Byzantine nodes, on the other hand, can send almost arbitrary messages. Such a model shall be parametrised by the number of honest and Byzantine nodes in the system, and the implementations of the cryptographic primitives. The parametrisation serves to enable formal reasoning about the abstract model itself, rather than its particular instantiations. We can then formally prove the key properties that are expected to hold for any instantiation of the system described by the model.

The key insight behind this project is that for the sake of sound verification of the properties of ABC, a comprehensive model of the protocol is just as important as that of the Byzantine behaviour. The advantage of an inductive model over TLA+ is that the specification of Byzantine behaviour can be descriptive, rather than prescriptive. In other words, instead of specifying what a Byzantine node *can* do, we can specify what it *cannot* do, which is a more appropriate way of thinking about Byzantine behaviour. Moreover, attempting to model Byzantine behaviour comprehensively in TLA+ would lead to an exponential growth of the state space and thus, make model checking very expensive.

This project has two primary objectives. First, to model ABC in TLA+, making certain assumptions about the behavior of Byzantine nodes that may not be entirely realistic, and then define its accountability properties. We verify these properties using bounded model checking. Second, to create a more realistic inductive model of ABC that systematically represents the Byzantine behavior with minimal restrictions on Byzantine actions. Once we have developed this model, we can rigorously prove the soundness of Byzantine node detection.

1.4 Contributions

The contributions of this project can be summarised as follows:

1. Specification of the ABC protocol in TLA+.
2. Statement and lightweight verification of the key properties of the Accountable Confirmer in TLA+.
3. Formal model of the ABC semantics and the network in which the protocol is executed, faithfully representing Byzantine behaviour.
4. Statement of the accountability-safety property ("every node detected is Byzantine") and its proof relying on an inductive invariant of the system.

Chapter 2

Background

2.1 Accountability in Distributed Systems

The ideal accountable consensus protocol is one that eventually identifies all guilty nodes as guilty (*completeness*), and never identifies correct nodes as guilty (*soundness*). However, what does it mean to be "guilty"? For instance, given the asynchrony of distributed systems we are not able to tell a deliberate omission error from an indefinite delay.

PeerReview, the first protocol offering an accountability layer for any consensus protocol, invokes the notion of faults that are *observable by a correct node* (Haeberlen et al., 2007). It also distinguishes *detectably faulty* (violating the rules of the protocol in a way that affects a correct node) from *detectably ignorant* (failing to acknowledge that it has received a message sent by a correct node) nodes. Every *detectably ignorant* node is *suspected* forever by a correct node, and for every node that is *detectably faulty* with respect to some message, some node that "propagated" the faulty message will get *exposed* eventually. Thus, it satisfies *completeness* with regards to *suspecting* the *detectably ignorant*, but not with respect to *detecting* the *detectably faulty*. However, it satisfies *soundness* in that no correct node is forever *suspected* by a correct node, and no correct node is ever *exposed* by a correct node. These distinctions let us differentiate between the faults that cannot be observed due to asynchrony (*detectably *ignorant* nodes, which can be subject of suspicion that is never confirmed but can be disproved) and the faults that are observable as there is evidence of protocol violation (usually a message) that can be found in a correct node's log (*detectably faulty* nodes). However, as noted earlier, we cannot always identify all *detectably faulty* nodes, so we do not have the desired guarantee of *completeness* in its full form. Moreover, it should be noted that satisfying these properties comes at a cost – the PeerReview protocol requires that each process maintains a log recording all of its received and sent messages, and each node can request the log of any other node, which incurs a significant communication cost.

Polygraph (Civit et al., 2021), on the other hand, is a Byzantine consensus protocol guaranteeing *agreement, validity, termination, and accountability*, where *accountability* is understood as detecting a set of processes that have contributed to the disagreement. In practice, a process contributes to a disagreement when it sends conflicting messages to different processes. In that sense, the notion of accountability applicable to the Polygraph protocol is weaker than the one from PeerReview, but this is due to the fact that the processes in Polygraph collect and exchange less information than in PeerReview. In Polygraph a process proves the culpability of at least $t_0 + 1$ processes by comparing its *certificate* to its logged *ledgers* (which contain some information received from other processes) and observing that they conflict i.e., were constructed in the same round but contain different values. Then it broadcasts the *ledgers* to others to propagate the information needed to prove culpability. However, Polygraph does not offer detection of omission faults (which are forever *suspected* in PeerReview). Hence, Polygraph has better communication complexity of $O(n^4)$, (which is the communication complexity of some commonly used BFTs), and scales better in exchange for a slightly weaker notion of accountability.

2.2 The ABC protocol

The ABC protocol transforms any BFT protocol by plugging in each node's decision into the *Accountable Confirmer* protocol. Then in the *Accountable Confirmer* a node can *confirm* the decision and possibly detect guilty processes. Given an instance of the *Accountable Confirmer ac*, and an instance of the Byzantine consensus protocol *bc*, we can describe the execution of the ABC protocol for some process P as follows:

1. *Byzantine Consensus* phase: Trigger the execution of *bc*.
2. *Accountable Confirmer* phase: If a value v is decided via *bc* then trigger the execution of *ac* by *submitting* v.
3. *Decision:* If a value v is *confirmed* via *ac* then decide that value.
4. *Detection:* If a set of processes S is detected via *ac* then detect S.

Therefore, the *ABC* transformation enables both consensus and detection.

The *Accountable Confirmer*, which lies at the heart of ABC, consists of the following phases:

1. *Submit:* Broadcast a *submit* message for a value v to all other processes.
2. *Confirm:* Upon receiving $N - t_0$ *submit* messages (where N is the number of nodes) for the submitted value v broadcast a *confirm* message with a *certificate*, which consists of the $N - t_0$ received *submit* messages for v.
3. *Detect:* If two valid *certificates* for two different values have been received, then construct a *proof* of culpability (S, F) from the intersection of the node sets in the two *certificates*, where S is a set of detected nodes and F contains evidence for their misbehaviour.

Intuitively, the *proof* is constructed by finding nodes that sent two *submit* messages for two different values, and this evidence of misbehaviour can be extracted from the set of collected *certificates.* This suggests that the Byzantine behaviour that can be detected with ABC consists of sending *submit* messages for two different values to two different nodes.

Note that for the messages included in a *certificate* to count as valid as evidence, we need the *submit* messages to include *digital signatures* to ensure that a message is not forged or tampered with when in flight. Then any node receiving a *submit* message can verify that the message was sent by the supposed sender using its public key,[4] and if the check passes the node can add the message and the signature to its *certificate.* [5]

The pseudo-code for the *Accountable Confirmer* algorithm (Civit et al., 2022) can be found in Appendix A.

[4]Usually this is done by invoking some *verify* function on the message and the signature.

[5]The original protocol contains a preliminary round of *light certificate* exchange, where *light certificates* contain only a combined signature for the value. This is to avoid sending the *full certificates*, carrying all received *submit* messages for the value, in case no *light certificates* conflict.

Chapter 3

ABC in TLA+

As noted in the previous section, ABC takes as input a BFT protocol, and runs the Accountable Confirmer to confirm the decision reached via the BFT, or in case of lack of decision detect the faulty nodes. Therefore, we would like to model the underlying BFT and the Accountable Confirmer protocol as two modules that can be composed together.[6]

3.1 Modelling Byzantine Consensus

In any consensus protocol there are three states in which a process p can be:

1. *working:* initial state, until p proposes a value v
2. *proposed:* p moves to this state after proposing some value v and stays in the state until it decides some value w
3. *decided:* p moves to this state after deciding some value w

In this general model of a consensus protocol we abstract away the decision process, assuming there is some protocol-specific deterministic mechanism for how the correct processes decide their values.

We begin by defining the constants of the model:

- `replicas` – set of all processes participating in the protocol.
- `values_all` – pool of all possible values to be proposed or decided.

[6]The code for the TLA+ model presented in this chapter is available at https://github.com/karolinagrzeszkiewicz/ABC-in-TLA.

Note that what we call constants in TLA+ are parameters of the model, and they have to be assigned values to restrict the model checking to the model defined by the value assignment.

Then we define the variables of the model:

- `proposals` – a function from `replicas` to `values_all`, assigning each replica the value it proposes, initially in the set of such functions.
- `proposals_set` – set of all proposed values, initially empty.
- `decisions` – a function from `replicas` to `values_all_opt`, assigning each replica the value it decided, initially "none" for all replicas.
- `states` – a function from `replicas` to the three state labels
- `is_Byzantine` – a function from `replicas` to `Bool` indicating whether a given replica is Byzantine.

Note that we can initialise variables with sets of values corresponding to various possibilities for the initial state. The model checker then explores all paths starting at each initial state.

Then we can define the two protocol transitions:

- `propose(r)` – given a non-Byzantine replica `r` in the "working" state, add `proposed(r)` to `proposals_set` and change `states(r)` to "proposed."
- `decide(r)` – given a replica `r` which has not decided yet and is in the "proposed" state, if no other replica is in the "working" state then change `states(r)` to "decided", and furthermore if `r` is honest:
  - if there are at most t_0 Byzantines, decide a deterministic value.[7]
  - else decide an arbitrary value from `proposals_set`.[8]

  If `r` is Byzantine then decide an arbitrary value from `proposals_set`.

[7]We can enforce that all honest replicas choose the same value with the `CHOOSE` syntax.
[8]This can be done by using an existential statement.

The model is defined by its initial state and the `Next` relation by which to move from one state to another some replica either proposes or decides:

$Next \triangleq \exists\, r \in replicas : propose(r) \lor decide(r)$

Note that this specification models not only the protocol executions, but also possible behaviour of Byzantine nodes in the network that deviates from the protocol. Furthermore, it models what happens when there are more than t_0 Byzantine nodes – in such a scenario arbitrary values can be decided since the Byzantine Consensus properties do not apply.

The *validity* and *agreement* properties of Byzantine Consensus should trivially hold given that the model is designed as a generic BFT. However, we state them as invariants and model check. We do not state *termination* – given the finite nature of TLA+ model checking, the liveness properties cannot be verified.

3.2 Modelling the Accountable Confirmer

The parameters of the *Accountable Confirmer* model are the same as in the previous section, since our "environment" is fully defined by the sets of replicas and possible values. To keep track of the state we define the following variables:

- `is_Byzantine` – a function from `replicas` to `Bool` (as above).
- `predecisions` – a function from `replicas` to `values_all` assigning to each replica the value it has decided and is about to submit
- `confirmed` – a function from `replicas` to `Bool` indicating whether a given replica has confirmed a value or not.
- `certificate` – a function from `replicas` to certificates, where a replica's certificate is the set of replicas that have sent it a submit message with the value it has predecided.
- `obtainedCertificates` – a function from replicas to the certificate messages they have collected.

- `proof` – a function from `replicas` to the powerset of `replicas` mapping each replica to the set of replicas it has detected.
- `msgs` – the set of all messages currently in the pool (when a message is received it gets removed from the pool, this is to model asynchrony).
- `rState` – a function from `replicas` to their possible states.
- `submitted` – a function from sender `replicas` to a function from receiver `replicas` to `values_all` i.e., `submitted[sender][receiver]` is the value that the sender has submitted to the receiver.

Then we define the model transitions for a replica `r` as follows:

- `submit` – broadcast a submit message for the value `predecisions(r)` to all other processes, can be taken if `r` is in the intial state, after the broadcast `r` switches to the "submitted" state.
- `receiveSubmit` – if the value in the submit message is the same as `predecisions(r)` then the sender is added to `certificate(r)`. This transition can be taken in the initial or "submitted" state.
- `confirm` – this transition can be taken if `r` has submitted a value but has not confirmed yet, and it has received at least $n - t_0$ submit messages for `predecisions(r)`. If this holds, then `r` can broadcast `certificate(r)` and move to the "confirmed" state.
- `receiveCertificate` – the certificate in the message is appended to `obtainedCertificates(r)`.
- `proveCulpability` – if there are two conflicting certificates in its `obtainedCertificates(r)`, then `r` sets `proof` equal to the intersection of the two certificates.

The key assumption behind this model is that processes can experience arbitrary delays in performing actions and in particular in receiving messages.

This is why we have two separate transitions for sending a message (whereby a message gets added to the pool of all unreceived messages) and for receiving a message (whereby a message is removed from the pool of messages and the receiving replica performs some bookkeeping).

Now we define a single transition that applies only to a Byzantine replica `b`:

- `submitByzantine` – for every `r` in `replicas` a submit message is sent from `b` to `r` with an arbitrary value in `values_all`.

The only other possible deviations from the protocol are failures to take some transition action, or sending arbitrary certificates. Yet the former is equivalent to an arbitrarily long delay and hence will be covered by the model checker. The latter is not possible since we assume that submit messages in a certificate are signed with private keys and thus cannot be forged.

3.3 Properties of the Accountable Confirmer

Ideally, we would like the following to hold for the *Accountable Confirmer*:

1. *Completeness* – If a process behaved Byzantine then it will get caught eventually.
2. *Soundness* – If a process got caught then the process must have behaved Byzantine.

We explore how to formally state these two properties and model check the candidate invariants for a set of four replicas and three possible values.

First, following the insight from chapter 2, we define *Byzantine behaviour* with respect to the *Accountable Confirmer* protocol as follows:

$behavedByzantine(r) \triangleq$

$\exists\, v1, v2 \in \{submitted[r][to] : to \in replicas \setminus \{r\}\},\ : v1 \neq v2$

Then consider the following definition of *completeness*:

$AccountabilityCompleteness \triangleq$

$$
\begin{array}{l}
\forall\, r1,\ r2 \in \mathit{replicas} : \\
(\ \land\ \mathit{is\_byzantine}[r1] = \text{“false”} \\
\quad \land\ \mathit{behavedByzantine}(r2) \\
\quad \land\ \mathit{rState}[r1] = \text{“proved”}) \\
\quad \implies\ r2 \in \mathit{proof}[r1])
\end{array}
$$

As we find out from the model checker, there is an interleaving of transitions which violates the property. This is because the Byzantine process might have submitted values that no honest process has pre-decided, in which case its submit messages are ignored even though they constitute *Byzantine behaviour.* An honest node registers only submit messages for the value it has pre-decided itself, so if a Byzantine node submits a value that no honest node has pre-decided, as far as the protocol is concerned, it has not submitted anything at all.

Furthermore, if the number of Byzantine processes is greater than $t_0 + 1$, sometimes a process might behave Byzantine but not with respect to exactly the two processes whose certificates were found to conflict with each other and used to construct as proof. For instance, suppose a Byzantine node b submits a value v to all but two nodes nodes, call them n_1 and n_2. b submits v' to n_1 (which has pre-decided v) and n_2 (which has pre-decided v'). Then if an honest node h constructs a proof of culpability from the conflicting certificates for v and v', sent by n_1 and n_2 respectively, b will not be in that proof. However, this is only possible when there are more than $t_0 + 1$ nodes, because the intersection of two sets of $N - t_0$ nodes is always of size at least $t_0 + 1$. Therefore, even if we redefine *Byzantine behaviour* as submitting two different values pre-decided by some honest nodes, the model will still fail to achieve *completeness.*

Now consider the *soundness* property:

$\mathit{AccountabilitySoundness} \triangleq$

$\forall\, r1 \in replicas : is\_byzantine[r1] = \text{“false”}$

$\implies \forall\, r2 \in proof[r1] : behavedByzantine(r2)$

Intuitively, *soundness* should hold, since the only way for a process to be in the intersection of two conflicting certificates is by sending two different values. Indeed, the model checker confirms that it holds for finite executions with chosen parameters.

Finally, we also state and model check the accountability property from the original paper (Civit et al., 2022), namely that if two honest processes confirm different values, then eventually every honest process detects at least $t_0 + 1$ faulty processes. This property can be thought of as a substitute for *completeness*, as it states that we can detect a significant number of Byzantine nodes.

$Accountability \triangleq$

$\exists\, p,\ q \in replicas : confirmDifferentVal(p,\ q)$

$\implies \forall\, r \in replicas :$

$(\land\ is\_byzantine[r] = \text{“false”}$

$\land\ rState[r] = \text{“proved”})$

$\implies Cardinality(proof[r]) \geq t0 + 1$

However, the TLA+ definition only asserts that once a replica has a proof of culpability the proof has cardinality at least $t_0 + 1$, which follows trivially from the set intersection size of two sets of at least $N - t_0$ members. Yet, we cannot do better than this since eventuality is a temporal property that cannot be expressed in first-order logic, unless we identify “eventually” with a particular state of the system.

3.4 Byzantine Behaviour and Accountability

We have provided models of a generic BFT together with the Accountable Confirmer, and verified the properties of the latter through finite execution. Our findings indicate that *completeness* does not hold, regardless of how we define *Byzantine behaviour*, and we conjectured that *soundness* holds.

However, the verification was not sound, not just because it relied on bounded model checking, but primarily because of the fact that due to the practical restrictions of model checking with TLC we were not able to model a large class of Byzantine behaviour.[9] This class consists of various ways in which the Byzantine nodes can collude, and hence go beyond merely submitting arbitrary values. Therefore our verification effort risked missing important classes of conceptual safety issues with the protocol. Henceforth, we will pursue a more comprehensive model of Byzantine behaviour, which can only be achieved by inductively modelling the system.

For the next step, we would like to define the system inductively. This will enable us to refine the model of Byzantine behaviour by defining minimal constraints on possible deviations from the protocol. Moreover, the model will be suitable for sound verification through inductive proofs, which can be mechanised in an automated theorem prover, such as Coq Proof Assistant.

[9]TLC is the model checker for TLA+ specifications.

Chapter 4

Modelling the B in ABC

In this section we present a full formalisation of the ABC protocol and the Byzantine behaviour acting against it. This formalisation is suitable for sound verification. We define inductively a global state of a network of nodes, which exchange messages asynchronously and possibly maintain a local state. The honest nodes are required to exchange messages according to the ABC protocol, but they operate in a network where certain messages appear "out of thin air" (independently of the protocol), signed by Byzantine nodes. Then the goal is to prove that only Byzantine nodes can be detected through the protocol (*soundness* of accountability), no matter what the adversaries do and how hard they try to forge evidence for misbehaviour of honest nodes.

4.1 Assumptions, Parameters and Axioms

In this section, we discuss the assumptions, parameters, and axioms of our model. Unlike in the TLA+ model, we can quantify over all possible values of parameters, provided that the axioms hold.

Firstly, we assume asynchrony of the network, meaning that messages can be rearranged, duplicated or take arbitrarily long to be delivered.

Secondly, the system is parametrised by a set of consensus parameters: a set of all addresses of nodes in the network, a subset of all addresses that belong to Byzantine processes, and a function value_bft from processes to values accepted in the BFT. The nodes with non-Byzantine addresses are honest, i.e., follow the Accountable Confirmer protocol, and the nodes with Byzantine addresses are Byzantine, i.e., can exhibit arbitrary behaviour constrained by the network semantics and limitations imposed by the cryptography used.

Thirdly, Public Key Cryptography is used to sign the value that a given node is submitting, and the signature can then be verified by any other node given the address of the signee (which can serve as its public key). We do not adopt any particular scheme, but rather take the secret keys of nodes and the sign and verify functions as parameters of the model, with the minimal requirement that a signature is valid (i.e., verify returns true) if and only if it is equal to one produced with the sign function. We also assume that secret keys are unique, and through the protocol semantics we enforce that the secret keys of honest nodes are not shared.

Finally, since the ABC protocol is parametrised by a BFT, or more concretely the values obtained by each node from its execution, we model these values with the value_bft function. We let value_bft be any function from node addresses to options of values (either "None" or "Some v" where v is a value), subject to the constraint that the Byzantine Consensus properties of *termination* and *agreement* hold. Note that *validity*, which asserts that the value decided by a correct node must have been proposed by a correct node, is not relevant to our model. Unlike in the TLA+ model, we abstract away the "propose" phase of a BFT protocol and only consider the final "decided" values, which have a bearing on the final outcome of executing the Accountable Confirmer protocol.

4.2 System State-Space

Any system configuration is a pair of *global state* $\triangle$ (a mapping from node addresses to their local states) and *packet soup* P (a pool of all packets sent by

$$\begin{aligned}
\mathsf{Addr} &\triangleq \mathbb{N} \\
\mathsf{NodeAddr} &\subseteq \mathsf{Addr} \\
\mathsf{ByzAddr} &\subseteq \mathsf{NodeAddr} \\
\mathsf{HonestAddr} &\triangleq \mathsf{NodeAddr} \setminus \mathsf{ByzAddr} \\
\mathsf{N} &\triangleq |\mathsf{NodeAddr}| \\
\mathsf{t}_0 &\triangleq \lceil \tfrac{N}{3} \rceil - 1 \\
\mathsf{t} &\triangleq |\mathsf{ByzAddr}|
\end{aligned}$$

FIGURE 4.1: Network parameters.

$$\begin{aligned}
\mathsf{Value} &: \mathsf{eqType} \\
\mathsf{Key} &: \mathsf{eqType} \\
\mathsf{Signature} &: \mathsf{eqType} \\
\mathsf{value\_bft} &: \mathsf{Addr} \to \mathsf{option\ Value} \\
\mathsf{verify} &: \mathsf{Value} \to \mathsf{Signature} \to \\
&\quad\ \mathsf{Addr} \to \mathsf{Bool} \\
\mathsf{secret\_key} &: \mathsf{Addr} \to \mathsf{Key} \\
\mathsf{sign} &: \mathsf{Value} \to \mathsf{Key} \to \mathsf{Signature}
\end{aligned}$$

FIGURE 4.2: State parameters.

$$\begin{aligned}
&\textit{termination} &&: \mathsf{t} \le \mathsf{t}_0 \implies \forall \mathsf{n} \in \mathsf{HonestAddr}, \exists\, \mathsf{v} : \mathsf{Value}, \mathsf{value\_bft}(\mathsf{n}) = \mathsf{Some\ v} \\
&\textit{agreement} &&: \mathsf{t} \le \mathsf{t}_0 \implies \forall \mathsf{n}_1, \mathsf{n}_2 \in \mathsf{HonestAddr},\ \mathsf{value\_bft}(\mathsf{n}_1) = \mathsf{value\_bft}(\mathsf{n}_2) \\
&\textit{two\_correct} &&: |\mathsf{HonestAddr}| \ \ge 2 \\
&\textit{valid\_sig} &&: \forall\, v : \mathsf{Value}, sig : \mathsf{Signature},\ n \in \mathsf{Addr}, \mathsf{verify}(v, sig, n) \\
& &&\quad \iff sig = sign(v, secret\_key(n)) \\
&\textit{unique\_keys} &&: \forall\, n_1,\ n_2 \in \mathsf{Addr},\ n_1 \neq n_2 \implies \mathsf{secret\_key}(n_1) \neq \mathsf{secret\_key}(n_2) \\
&\textit{unique\_sig} &&: \forall v_1,\ v_2\ : \mathsf{Value},\ n_1,\ n_2 \in \mathsf{Addr},\ v_1 \neq v_2 \lor n_1 \neq n_2 \\
& &&\quad \implies \mathsf{sign}(v_1, \mathsf{secret\_key}(n_1)) \neq \mathsf{sign}(v_2, \mathsf{secret\_key}(n_2))
\end{aligned}$$

FIGURE 4.3: Axioms of the framework parameters.

nodes in the network). The initial system configuration is defined as an empty *packet soup* and an initial local state for every honest node, since we cannot make any assumptions about the state of Byzantine nodes. By *termination* $\forall\, n \in \mathsf{HonestAddr}, \exists\, v : \mathsf{Value},\ \mathsf{value\_bft}(n) = \mathsf{Some}\ v$, so we can define that local state in terms of a node's pre-decided value.

A *local state* of a node is a quadruple that models both immutable and mutable state of a node. The quadruple includes the address of the node, a Bool value that corresponds to whether the node has already confirmed or not, the node's *certificate*, and the set of *certificates* the node has received from other nodes. The *certificate* consists of a value, and a set of nodes that have *submitted* this value to our node, together with each node's *signature* for the value. A *certificate* serves as evidence for other nodes' actions, and hence *signatures* enable

$$\begin{aligned}
\triangle \in \mathsf{GlobState} &\triangleq \mathsf{NodeAddr} \rightarrow \mathsf{LocalState}\\
P \in \mathsf{PacketSoup} &\triangleq \mathcal{P}(\mathsf{Packet})\\
\sigma \in \mathsf{Config} &\triangleq \mathsf{GlobState} \times \mathsf{PacketSoup}
\end{aligned}$$

FIGURE 4.4: System configurations.

$$\begin{aligned}
&\wedge\ \forall\, n \in \mathsf{HonestAddr}, \exists\, v : \mathsf{Value},\\
&\quad \mathsf{value\_bft}(n) = \mathsf{Some}\ v\\
&\quad \Longrightarrow \triangle_0(n) = \langle n, \mathit{false}, \langle v, \emptyset \rangle, \emptyset \rangle\\
&\wedge\ P_0 = \emptyset
\end{aligned}$$

FIGURE 4.5: Initial system configuration.

$$\begin{aligned}
\delta \in \mathsf{LocalState} &\triangleq \mathsf{Addr} \times \mathsf{Bool} \times \mathsf{Certificate} \times \mathcal{P}(\mathsf{Certificate})\\
c \in \mathsf{Certificate} &\triangleq \mathsf{Value} \times \mathcal{P}(\mathsf{NodeAddr} \times \mathsf{Signature})
\end{aligned}$$

FIGURE 4.6: Local state.

$$\begin{aligned}
p \in \mathsf{Packet} &\triangleq \mathsf{Addr} \times \mathsf{Addr} \times \mathsf{Msg} \times \mathsf{Bool}\\
P_{rcv} &\triangleq \{\langle \mathsf{src}\ p, \mathsf{dest}\ p, \mathsf{msg}\ p \rangle \mid p \in P\ \wedge\ \mathsf{received}\ p = \mathit{true}\}\\
P_{sent} &\triangleq \{\langle \mathsf{src}\ p, \mathsf{dest}\ p, \mathsf{msg}\ p \rangle \mid p \in P\}\\
\mathsf{mark\_rcv}(P, p) &\triangleq P \setminus \{p\} \cup \{\langle \mathsf{src}\ p, \mathsf{dest}\ p, \mathsf{msg}\ p, \mathit{true} \rangle\}\\
m \in \mathsf{Msg} &::= \mathsf{SubmitMsg}(\langle \mathsf{v} : \mathsf{Value},\ \mathsf{sig} : \mathsf{Signature} \rangle)\\
&\ \mid\ \mathsf{ConfirmMsg}(\mathsf{c} : \mathsf{Certificate})
\end{aligned}$$

FIGURE 4.7: Messages and Packets.

verification of the validity of such evidence and detection of forged evidence.

Packets are quadruples containing the address of the sender, the address of the receiver, the message, and a Bool value indicating whether the message has been received by the addressee. The contents of messages are defined by the Msg data type. According to the data type a message can be of two kinds, either a *submit* message for a *value* and a *signature* (which should be the *signature* for the submitted *value* if the node is following the protocol), or a *confirm* message for a *certificate*, by means of which a node can broadcast evidence that a given set of nodes have "voted for" its value.

Note that the *packet soup* stores the history of all messages ever sent, and for a protocol that implements accountability a history of messages sent is all we need to reason about the *safety* properties of the protocol. Henceforth, we refer to the set of all messages in the *packet soup* as P_{sent}, and the set of messages that have been received, a subset of P_{sent}, as P_{rcv}.

4.3 Local Node Semantics

Per-node transitions define how a node operates, namely the atomic transition it undergoes in reaction to receiving a message or without any triggering event. A node's response involves an update of its local state and possibly emitting new packets. The operational semantics presented in this chapter are in line with the Accountable Confirmer protocol and thus concern only honest nodes. We follow a relational style of presenting operational semantics, where a rule describes the relation between the preconditions for the state update (top part) and the state update itself (bottom part). We split the local node transitions into (1) receive step transitions $\delta \xrightarrow{p}_{\rho} (\delta', ps)$ where a node changes its state from δ to δ' upon receiving a packet p, and emits a set of packets *ps*, and (2) internal step transitions $\delta \longrightarrow_{\iota} (\delta', ps)$ where a node changes its state from δ to δ', and emits a set of packets *ps*, where the node is not reacting to any event and hence, can take the step transition at an arbitrary moment.

Receive transitions can be divided into RCVSUBMIT transitions in reaction to receiving a *submit* message, and RCVCONFIRM transitions in response to a *confirm* message. The precondition for the former transition is that the message received is a submit message for the same value that the receiver has in its certificate (i.e., the value it decided in the BFT phase) and that the signature in the message is a valid signature for the value sent. If that precondition holds and the receiver has not finished collecting signatures for its certificate (i.e., not reached the $\mathsf{N} - \mathsf{t}_0$ threshold) then the receiver updates its certificate with the address and signature of the sender. Furthermore, if the $\mathsf{N} - \mathsf{t}_0$ threshold is reached upon the certificate update then the sender broadcasts its certificate to everyone in a *confirm* message. The RCVCONFIRM transition, on the other hand, only requires that all of the signatures in the certificate received via the

Receive-step transitions: $\delta \xrightarrow{p}_{\rho} (\delta', ps)$

RcvSubmit

$$\frac{\begin{array}{c} nsigs' = (\mathit{if}\ (v = v'\ \mathit{and}\ \mathit{verify}(v, sig, from)\ \mathit{and\ not}\ conf)\ \mathit{then}\ \{\langle from, sig\rangle\} \cup nsigs\ \mathit{else}\ nsigs) \\ conf' = |nsigs'| \geq N - t_0 \\ \mathsf{ps} = (\mathit{if}\ \mathsf{conf'}\ \mathit{then}\ \{\langle \mathsf{this}, n, \mathit{ConfirmMsg}\ \langle \mathsf{v}, \mathsf{nsigs'}\rangle\rangle \mid n \in \mathsf{NodeAddr}\}\ \mathsf{else}\ \emptyset \end{array}}{\langle this, conf, \langle v, nsigs\rangle, certs\rangle \xrightarrow{\langle from,\ this,\ SubmitMsg\ \langle v', sig\rangle\rangle}_{\rho} (\langle this, conf', \langle v, nsigs'\rangle, certs\rangle, ps)}$$

RcvConfirm

$$\frac{\forall \langle n, sig\rangle \in nsigs,\ \mathit{verify}(v, sig, n)}{\langle this, conf, cert, certs\rangle \xrightarrow{\langle from,\ this,\ ConfirmMsg\ \langle v,\ nsigs\rangle\rangle}_{\rho} (\langle this, conf, cert, \{\langle v,\ nsigs\rangle\} \cup certs\rangle, \emptyset)}$$

FIGURE 4.8: Local semantics, receive-transitions.

Internal step transitions: $\delta \longrightarrow_{\iota} (\delta', ps)$

INTSUBMIT

$$\frac{\begin{array}{c} \mathsf{value\_bft}(\mathit{this}) = \mathsf{Some}\ v \qquad sig = \mathit{sign}(v, \mathit{secret\_key}(this)) \\ \mathsf{ps} = \{\langle \mathsf{this}, n, \mathit{SubmitMsg}\ \langle v, \mathsf{sig}\rangle\rangle \mid n \in \mathsf{NodeAddr}\} \end{array}}{\langle this, conf, cert, certs\rangle \longrightarrow_{\iota} (\langle this, conf, cert, certs\rangle,\ ps)}$$

FIGURE 4.9: Local semantics, internal transitions.

confirm message are valid, and if that is the case the received certificate is added to the set of certificates collected from other nodes.

There is a unique internal step transition, namely INTSUBMIT, which corresponds to a node broadcasting a *submit* message for the value it has decided in the BFT phase. As a precondition we require that the broadcasted value is the one the node has decided in the BFT, and the signature is a valid signature of the sender for the sent value. Note that this transition can be taken by a node any time, and possibly multiple times. Yet since we model the collection of addresses and signature in a certificate as a set, by the RCVSUBMIT transition if a node and its signature are already in the set, then the set is not updated, hence preventing duplicates from appearing in the signature set.

4.4 Network Transitions

We model our network with five network transition rules which define all possible changes to the system configuration in one system step. The network transition rules are of the form $\langle \Delta, P \rangle \implies \langle \Delta', P' \rangle$, where $\langle \Delta', P' \rangle$ refers to the updated *system configuration*. In this section we focus on the first two transition rules which are non-Byzantine, meaning that they model how packets addressed to honest nodes are picked up from the *packet soup* and processed by the addressee (according to local node transitions). There is also a trivial transition NETIDENTITY which leaves $\langle \Delta, P \rangle$ unchanged.

NETDELIVER corresponds to the global state transition of delivering a randomly picked unread message p from *packet soup* P to its destination address a of a node with state δ, where a must be a non-Byzantine address. Then the node with address a processes the message as specified by the receive step transitions, depending on the type of the message delivered.

NETINTERNAL defines a transition that can be taken by an honest node any time, regardless of the contents of the *packet soup*. The local state change of the node and the emitted messages are defined by the internal step transition INTSUBMIT. This means that the only state change an honest node can undergo without being prompted by any event is broadcasting *submit* messages.

4.5 Byzantine Network Transitions

There are two network transitions corresponding to actions taken by Byzantine adversaries. The key insight behind the transitions is that Byzantine nodes can do anything that is possible in a system. In case of the ABC protocol, this implies being able to send any messages consistent with the Msg data type and being subject to constraints imposed by the public key cryptography. We enforce

Network transitions: $\delta \longrightarrow_\iota (\delta', ps)$

NETDELIVER

$$\frac{\begin{array}{ccc} p \in P & \text{dest } p = a & \text{received } p = \mathit{false} \\ a \in \text{HonestAddr} & \Delta(a) = \delta & \delta \xrightarrow{p}_\rho (\delta', ps) \end{array}}{\langle \Delta, P \rangle \xRightarrow{\text{rcv } a\ p} \langle \Delta[a \mapsto \delta'], \text{mark\_rcv}(P, p) \cup ps\} \rangle}$$

NETINTERNAL

$$\frac{\Delta(a) = \delta \qquad a \in \text{HonestAddr} \qquad \delta \longrightarrow_\iota (\delta', ps)}{\langle \Delta, P \rangle \xRightarrow{\text{int a}} \langle \Delta[a \mapsto \delta'], P \cup ps \rangle}$$

NETIDENTITY

$$\langle \vartriangle, P \rangle \xRightarrow{\text{id}} \langle \vartriangle, P \rangle$$

FIGURE 4.10: Non-Byzantine network transitions.

that at least the honest nodes follow the signature scheme, but the Byzantines might disregard it, by sharing their secret keys or signatures for values with other nodes.

Note that we model the adversarial actions in the network transition level rather than through special local transitions for Byzantine nodes. This is because we are interested in how the Byzantine nodes, as a group, can corrupt the system by sending arbitrary messages. Hence, the local state of a Byzantine node is not modelled. Instead, what we model is adding (almost) arbitrary messages from Byzantine senders to the *packet soup*, which represents the most general scenario of Byzantine messages appearing in the system "out of thin air". The above representation of Byzantine behaviour is complete, rather than underapproximating the space of Byzantine actions by modelling concrete actions like we did with the TLA+ model of ABC.

The NETBYZSUBMIT transition corresponds to an arbitrary *submit* message being sent from an arbitrary Byzantine address to an arbitrary node address. Note that Byzantine nodes can send submit messages with arbitrary values and signatures – even if the signature is invalid. By RCVSUBMIT honest nodes should ignore messages with invalid signatures.

On the other hand, the content of a *confirm* message being sent from a Byzantine address to an arbitrary address cannot be entirely arbitrary, as shown in the NETBYZCONFIRM transition. The confirm messages emitted by Byzantine nodes are subject to some minimal constraints due to the fact that the nodes cannot forge the signatures of other nodes without (1) knowing their secret keys or (2) having overheard the signature for the value being signed. Now we assume that the honest nodes follow the given signature scheme by not sharing their secret keys or signatures, unless the signature is shared via broadcasting a *submit* message. Yet we cannot expect the same from the Byzantine nodes. For instance, the Byzantine nodes might be collaborating and intentionally exchanging their secret keys, or simply be dismissive of the rules of secure communication. Hence, we only require that for all signatures of honest nodes in the certificate prepared by a Byzantine node, if the signature for a given value is valid, then there is a corresponding *submit* message in the *packet soup* with that signature and value. Any such *submit* message must have been sent to the preparing Byzantine node, or to any other node (as the Byzantine node might have been eavesdropping on the communication channel and storing signatures of other nodes). However, there are no requirements on the signatures of Byzantine nodes – given the above-mentioned reasons we cannot assume that it is impossible to forge them.

Network transitions: $\delta \longrightarrow_\iota (\delta', ps)$

NETBYZSUBMIT

$$\frac{p : \mathsf{Packet} \qquad \mathsf{src}\ p\ \in\ \mathsf{ByzAddr} \\ \exists\ v : \mathsf{Value}, sig \in \mathsf{Signature},\ \mathsf{msg}\ p = \mathsf{SubmitMsg}\ \langle v, sig\rangle}{\langle \triangle, P\rangle \xrightarrow{\mathsf{byz}\ p} \langle \triangle, P \cup \{p\}\rangle}$$

NETBYZCONFIRM

$$\frac{p : \mathsf{Packet} \qquad \mathsf{src}\ p\ \in\ \mathsf{ByzAddr} \\ \exists\ \mathsf{v} :\ Value,\ \mathsf{nsigs} \in \mathcal{P}(\mathsf{NodeAddr} \times \mathsf{Signature}),\ \mathsf{msg}\ p\ =\ \mathsf{MConfirm}\ \langle v, nsigs\rangle \\ \forall\ \langle n, \mathsf{sig}\rangle\ \in\ \mathsf{nsigs},\ n \in \mathsf{HonestAddr}\ \wedge\ sig = \mathsf{sign}(v, \mathsf{secret\_key}(n)) \\ \Longrightarrow\ \exists\ n' \in \mathsf{NodeAddr}, b : \mathsf{Bool},\ \langle n, n',\ \mathsf{MSubmit}\ \langle v, sig\rangle,\ b\rangle \in P}{\langle \triangle, P\rangle \xrightarrow{\mathsf{byz}\ p} \langle \triangle, P \cup \{p\}\rangle}$$

FIGURE 4.11: Byzantine network transitions.

4.6 The Accountability-Soundness Property

For the model described above, we would like to prove that the *Accountability – Soundness* property holds. This property asserts that ABC has no false positives i.e, if a node is detected then it must be Byzantine. This property is key if we want to employ ABC to punish malicious nodes or exclude them from the network.

First, we need to formally define how nodes get detected. We do not represent proof as a local state component since this would entail that a proof of culpability is constructed by each node asynchronously. However, the proof of culpability can be defined by the intersection of two conflicting certificates from the set of certificates collected by a node. Then we define proof as a function from a set of certificates to a set of nodes.[10]

Now, we can formally state *Accountability–Soundness* as follows: *"If a correct process obtains a proof of culpability of another processes, then the detected*

[10]Since for *Accountability–Soundness* we only consider proofs constructed by honest nodes, we do not need the original submit messages of detected nodes in the proof.

$$
\begin{aligned}
&\forall\ \text{certs} \in P(\text{Certificate}), n \in \text{proof}(certs) \iff \\
&\quad \exists \text{v}_1, \text{v}_2 : Value, \text{sig}_1, \text{sig}_2 \in \text{Signature}, \\
&\quad \text{nsigs}_1, \text{nsigs}_2 \in \mathcal{P}(\text{NodeAddr} \times \text{Signature}), \\
&\quad \wedge \langle v_1, nsigs_1 \rangle \in certs \\
&\quad \wedge \langle v_2, nsigs_2 \rangle \in certs \\
&\quad \wedge v_1 \neq v_2 \\
&\quad \wedge \langle n, sig_1 \rangle \in nsigs_1 \\
&\quad \wedge \langle n, sig_2 \rangle \in nsigs_2
\end{aligned}
$$

FIGURE 4.12: Definition of the proof function.

process is Byzantine." . However, for a stronger notion of soundness, not only do we want the detected node to be Byzantine, but we also want it to have behaved Byzantine.[11] This could be stated as *"If a correct process obtains a proof of culpability of another processes, then the detected process has behaved Byzantine."* This was the soundness property we formulated in chapter 3, conjecturing that it holds upon running the model checker for fixed parameters.

However, this stronger notion of soundness does not necessarily hold. It is violated by the scenario in which one of the conflicting confirm messages was fabricated by a Byzantine node b, and the signature of the Byzantine node n in the certificate was forged by b (given that b might have had access to n's secret key) rather than being taken from a corresponding submit message in the *packet soup*. Note that our TLA+ model did not account for this possibility. Hence, we consider the former notion of accountability.

In the next two sections we will prove that *Accountability–Soundness* holds for any *global state* $\triangle$ and *packet soup* P satisfying the inductive invariant.

[11] A Byzantine node can behave honestly i.e., follow the protocol.

$$
\begin{aligned}
&\mathsf{behavedByz}(n, P) := \\
&\quad \exists\, v_1,\ v_2 : \mathsf{Value}, sig_1, sig_2 : \mathsf{Signature}, \\
&\quad a_1, a_2 \in \mathsf{NodeAddr}, \\
&\quad \wedge\, v_1 \neq v_2 \\
&\quad \wedge\, \langle n,\ a_1, \mathsf{SubmitMsg}\langle v_1, sig_1\rangle, \mathsf{true}\rangle \in P \\
&\quad \wedge\, \langle n,\ a_2, \mathsf{SubmitMsg}\langle v_2, sig_2\rangle, \mathsf{true}\rangle \in P \\
&\quad \wedge\, sig_1 = sign(v_1, secret\_key(n)) \\
&\quad \wedge\, sig_2 = sign(v_2, secret\_key(n))
\end{aligned}
$$

FIGURE 4.13: Byzantine behaviour.

$$
\begin{aligned}
&AccSoundness(\langle \triangle, P\rangle) := \\
&\quad \forall\, n\ \in \mathsf{NodeAddr},\ h \in \mathsf{HonestAddr}, \\
&\quad n \in \mathsf{proof}(\triangle(h)) \implies n \in \mathsf{ByzAddr} \\
&AccSoundness'(\langle \triangle, P\rangle) := \\
&\quad \forall\, n\ \in \mathsf{NodeAddr},\ h \in \mathsf{HonestAddr}, \\
&\quad n \in \mathsf{proof}(\triangle(h)) \\
&\quad \implies \mathsf{behavedByz}(n, P)
\end{aligned}
$$

FIGURE 4.14: Accountability–Soundness.

4.7 Inductive Invariant

In this section we define an inductive system invariant – a property of the system that holds inductively, i.e., (1) it holds for the initial system configuration $\langle \triangle_0, P_0\rangle$, and (2) if it holds for $\langle \triangle, P\rangle$ then it also holds for $\langle \triangle', P'\rangle$ where $\langle \Delta, P\rangle \implies \langle \Delta', P'\rangle$. Such a property would then hold for every possible system configuration $\langle \triangle, P\rangle$. The inductive invariant is an overapproximation of the system semantics, describing what the semantics entails. We would like to have an inductive invariant that implies the *Accountability–Soundness* property. We find that the invariant in Figure 4.15 is inductive and implies *Accountability–Soundness*. The key insight behind this invariant is that it encapsulates the transitions of the system, tracing back the preconditions that must hold for a node to end up in the proof of culpability constructed by an honest node.

Now we will present a proof sketch for the inductivity of this invariant.

Proof. First, we want to prove that $\textsc{Inv}(\langle \triangle_0, P_0\rangle)$ holds. We know that $\forall n \in \mathsf{NodeAddr},\ certs(\triangle_0(n)) = \emptyset$ so (1) holds trivially. Similarly, (2), (4), and (5) hold trivially since $P_0 = \emptyset$, and (3) holds trivially because $\forall n \in \mathsf{NodeAddr}$, $cert(\triangle_0(n)) = \langle value\_bft(n), \emptyset\rangle$. Then we know that $\textsc{Inv}(\langle \triangle_0, P_0\rangle)$ holds.

Now suppose that $\textsc{Inv}(\langle \triangle, P\rangle)$ holds for some system configuration $\triangle, P$ (inductive hypothesis), and $\langle \Delta, P\rangle \overset{s}{\implies} \langle \Delta', P'\rangle$ where s is one of the network

$$
\begin{array}{l}
\textsc{Inv}(\langle \triangle, P\rangle) := \\
(1)\ \land\ \forall\, h\ \in \text{Honest}, v : \text{Value}, nsigs \in P(\text{NodeAddr} \times \text{Signature}), \\
\qquad \langle v, \textit{nsigs}\rangle \in \textit{certs}(\triangle(h)) \\
\qquad \implies \exists n' \in \textit{NodeAddr}, \langle n', h, \textit{ConfirmMsg}\ \langle v, nsigs\rangle\rangle \in P_{rcv} \\
\qquad\qquad \land \forall \langle n, \textit{sig}\rangle \in \textit{nsigs},\ \textit{sig} = sign(v, secret\_key(n)) \\
(2)\ \land\ \forall\, h\ \in \text{Honest}, n \in \text{NodeAddr}, v : \text{Value}, nsigs \in P(\text{NodeAddr} \times \text{Signature}), \\
\qquad \langle h, n, \textit{ConfirmMsg}\ \langle v, nsigs\rangle\rangle \in P_{sent} \implies \textit{cert}(\triangle(h)) = \langle v, \textit{nsigs}\rangle \\
(3)\ \land\ \forall\, h\ \in \text{Honest}, v : \text{Value}, nsigs \in P(\text{NodeAddr} \times \text{Signature}), \\
\qquad \textit{cert}(\triangle(h)) = \langle v, \textit{nsigs}\rangle \\
\qquad \implies \forall \langle n, \textit{sig}\rangle \in \textit{nsigs}, \langle n, h, \textit{SubmitMsg}\ \langle v, \textit{sig}\rangle\rangle \in P_{rcv} \\
\qquad\qquad \land\ \textit{sig} = sign(v, secret\_key(n)) \\
(4)\ \land\ \forall b \in \text{ByzAddr}, h \in \text{NodeAddr}, v : \text{Value}, nsigs \in P(\text{NodeAddr} \times \text{Signature}), \\
\qquad \langle b, h, \textit{ConfirmMsg}\ \langle v, nsigs\rangle\rangle \in P_{sent} \\
\qquad \implies \forall \langle n, \textit{sig}\rangle \in \textit{nsigs}, \\
\qquad\qquad n \in \text{HonestAddr}\ \land\ sig = \text{sign}(v, \text{secret\_key}(n)) \\
\qquad\qquad \implies \exists n' \in \textit{NodeAddr},\ \langle n, n', \textit{SubmitMsg}\ \langle v, sig\rangle\rangle \in P_{sent} \\
(5)\ \land\ \forall\, h\ \in \text{Honest}, n_1, n_2 \in \text{NodeAddr}, sig_1, sig_2 \in \text{Signature}, v_1, v_2 : \text{Value}, \\
\qquad (\langle h, n_1, \textit{SubmitMsg}\ \langle v_1, sig_1\rangle\rangle \in P_{sent}\ \land\ \textit{sig}_1 = sign(v_1, secret\_key(h)) \\
\qquad \land \langle h, n_2, \textit{SubmitMsg}\ \langle v_2, sig_2\rangle\rangle \in P_{sent}\ \land\ \textit{sig}_2 = sign(v_2, secret\_key(h))) \\
\qquad \implies\ v_1 = v_2
\end{array}
$$

FIGURE 4.15: The inductive invariant.

transitions. Then we consider the five cases corresponding to the five network transition rules:

Case I: s = NETIDENTITY

Then $\langle \triangle', P'\rangle = \langle \triangle, P\rangle$ so by inductive hypothesis $\textsc{Inv}(\langle \triangle', P'\rangle)$ holds.

Case II: s = NETDELIVER (p, δ) and msg p = SubmitMsg $\langle v, sig\rangle$ for some v : Value, sig : Signature

Then if (1) holds for $\langle \triangle, P\rangle$ then it should also hold for $\langle \triangle', P'\rangle$ since $\forall h \in \text{Honest}\ certs(\triangle(h)) = certs(\triangle'(h))$ (*certificates* collected by a node can only be updated through receiving a *confirm* message).

Now for (2), by the inductive hypothesis:

$$\forall\, h\ \in \text{Honest}, n \in \text{NodeAddr}, v : \text{Value}, nsigs \in P(\text{NodeAddr} \times \text{Signature}),$$

$$\langle h, n, \textit{ConfirmMsg}\ \langle v, nsigs\rangle\rangle \in P_{sent} \implies \textit{cert}(\triangle(h)) = \langle v, \textit{nsigs}\rangle$$

We know that $P'_{sent} = P_{sent} \cup \{\langle src\ p, dest\ p, SubmitMsg\ \langle v, sig\rangle\rangle\}$. Note that P'_{sent} does not contain any *confirm* messages that are not in P_{sent}, and by RCV-SUBMIT an honest node's *certificate* is not updated after being sent. Then (2) must hold for $\langle \triangle', P'\rangle$.

For (3), by the inductive hypothesis:

$$\begin{aligned}
&\forall\ h \in \mathsf{Honest}, v : \mathsf{Value}, nsigs \in P(\mathsf{NodeAddr} \times \mathsf{Signature}),\\
&\qquad cert(\triangle(h)) = \langle v, nsigs\rangle\\
&\qquad\quad \implies \forall \langle n, sig\rangle \in nsigs, \langle n, h, SubmitMsg\ \langle v, sig\rangle\rangle \in P_{rcv}\\
&\qquad\qquad \land\ sig = sign(v, secret\_key(n))
\end{aligned}$$

We only need to prove it for $h = \text{dest}\ p$ since for other nodes the property holds by inductive hypothesis. If $cert(\triangle'(h)) = cert(\triangle(h)$ then it also holds trivially. The only remaining case is $cert(\triangle'(h)) = \langle v, \{\langle n', sig'\rangle\} \cup nsigs\rangle$ where $cert(\triangle(h)) = \langle v, nsigs\rangle$, in which case we only need to consider the newly added signature (the remaining ones are covered by the inductive hypothesis). Then by RCVSUBMIT $\langle n', h, SubmitMsg\ \langle v, sig'\rangle\rangle \in P'_{rcv}$ and also $sig' = sign(v, secret\_key(n'))$. Hence, the property holds in $\langle \triangle', P'\rangle$.

Now, (4) and (5) hold in $\langle \triangle', P'\rangle$ by inductive hypothesis, since the transition we are considering does not involve sending *confirm* messages with Byzantine sender address or sending *submit* messages (only receiving).

Case III: $s = \text{NETDELIVER}\ (p, \delta)$ and msg $p = \mathsf{ConfirmMsg}\ \langle v, nsigs\rangle$ for some $v : \mathsf{Value}, nsigs : \mathcal{P}(\mathsf{NodeAddr} \times \mathsf{Signature})$

Consider (1). By RCVCONFIRM clearly if $certs(\triangle'(dest\ p)) = certs(\triangle(dest\ p)) \cup \{cert'\}$ for some $cert' = \langle v, nsigs\rangle$ then $\langle \text{src}\ p, \text{dest}\ p, \mathsf{ConfirmMsg}\ \langle v, nsigs\rangle\rangle \in P_{rcv}$ and $\land \forall \langle n, sig\rangle \in nsigs,\ sig = sign(v, secret\_key(n))$. Hence, if (1) holds for $\langle \triangle, P\rangle$ then it also holds for $\langle \triangle', P'\rangle$.

Now (2), (3), (4), and (5) hold for $\langle \triangle', P' \rangle$ by induction hypothesis alone since these propositions do not assert anything about received *confirm* messages.

Case IV: s = NETINTERNAL (a, δ)

(1), (2), (3), and (4) hold for $\langle \triangle', P' \rangle$ by induction hypothesis alone since these propositions do not assert anything about sent *submit* messages.

For (5) note that the only new submit messages in P' could come from honest address a with state δ. Yet by INTSUBMIT a, if value_bft(a) = Some v for some v : Value, then a always submits v, and $P'_{sent} = P_{sent} \cup \{\langle a, n, SubmitMsg\ \langle v, \text{sig}\rangle\rangle \mid n \in \text{NodeAddr}\}$ where $sig = sign(v, secret\_key(a))$. Therefore, since the values submitted are always the ones contained in value_bft(a), and by *termination* $\forall a \in \text{HonestAddr}, \exists\ v\ :\ \text{Value}, \text{value\_bft}(a)\ =\ \text{Some}\ v$, they are necessarily the same, both in P_{sent} and P'_{sent}. So the property also holds for $\langle \triangle', P' \rangle$.

Case V: s = NETBYZSUBMIT p

(1), (2), (3), (4) and (5) hold for $\langle \triangle', P' \rangle$ by induction hypothesis since thesy do not assert anything about *submit* messages sent from Byzantine addresses.

Case VI: s = NETBYZCONFIRM p

(1), (2), and (3) hold for $\langle \triangle', P' \rangle$ by induction hypothesis since they do not assert anything about *confirm* messages sent from Byzantine addresses.

For (4) we know that $P'_{sent} = P_{sent} \cup \{\langle src\ p, dest\ p, ConfirmMsg\ \langle v, \textbf{nsigs}\rangle\rangle\}$ for some v : $Value$, nsigs $\in \mathcal{P}(\text{NodeAddr} \times \text{Signature})$, and src $p \in$ ByzAddr. Then by NETBYZCONFIRM:

$$\forall \langle n, \text{sig}\rangle \in \text{nsigs}, n \in \text{HonestAddr}\ \wedge\ sig = \text{sign}(v, \text{secret\_key}(n))$$

$$\implies\ \exists n' \in NodeAddr,\ \langle n, n', SubmitMsg\ \langle v, sig\rangle\rangle \in P_{sent} \subseteq P'_{sent}$$

Hence, the assertion holds for all messages in P_{sent} and for the new message $\langle \text{src}\ p, \text{dest}\ p, \text{ConfirmMsg}\ \langle v, nsigs\rangle\rangle$, so the property must hold for $\langle \triangle', P' \rangle$.

(5) also holds trivially for $\langle \triangle', P' \rangle$ since it does not concern *confirm* messages sent by Byzantine nodes.

□

Note that the proof is structured such that each of the five conjuncts of the invariant is *validated* by one network transition (or one network and one local node transition combined), and *not invalidated* by the remaining transitions (in which case it holds by induction hypothesis alone). This demonstrates how the invariant is nothing more than a slight overapproximation of the ABC semantics, encapsulating the logic behind the transitions that makes the *Accountability–Soundness* property "true." In the light of the above-mentioned, the inductive invariant can be thought of a logical intermediate layer between the formal specification of ABC and the *Accountability–Soundness* property – the system specification implies the invariant, and the invariant implies the *Accountability–Soundness* property.

4.8 Proving Accountability–Soundness

Given the that the inductive invariant holds for any system configuration $\langle \triangle, P \rangle$, we can prove that *AccSoundness*$(\langle \triangle, P \rangle)$ holds for any system configuration $\langle \triangle, P \rangle$ simply as a consequence of the validity of the invariant, without the need to use induction. For the sake of brevity, in the proof we will refer to the five conjuncts of the invariant by their respective identifiers (1)-(5).

Proof. Suppose a node n is in the proof extracted from the certificates of an honest node, i.e., let $n \in \mathsf{NodeAddr}, h \in \mathsf{Honest}$ and $n \in \mathsf{proof}(\mathsf{certs}(\triangle(\mathsf{h})))$. Then by the specification of proof, $\exists\, c_1, c_2 \in \mathit{certs}(\triangle(h))$ such that:

$$c_1 = \langle v_1, \mathsf{nsigs}_1 \rangle \text{ for some } \mathsf{nsigs}_1 \in \mathcal{P}(\mathsf{NodeAddr} \times \mathsf{Signature}), v_1 : \mathsf{Value} \quad (4.1)$$

$$c_2 = \langle v_2, \mathsf{nsigs}_2 \rangle \text{ for some } \mathsf{nsigs}_2 \in \mathcal{P}(\mathsf{NodeAddr} \times \mathsf{Signature}), v_2 : \mathsf{Value} \quad (4.2)$$

$$v_1 \neq v_2 \tag{4.3}$$

$$\exists sig_1, sig_2 : Signature, \text{ such that } \langle n, sig_1 \rangle \in nsigs_1 \text{ and } \langle n, sig_2 \rangle \in nsigs_2 \tag{4.4}$$

Then by (1) both c_1 and c_2 must have been sent by some other nodes, call them n_1 and n_2 i.e:

$$\exists\ n_1 \in NodeAddr, \langle n_1, h, ConfirmMsg\ \langle v_1, nsigs_1 \rangle\rangle \in P_{rcv} \tag{4.5}$$

$$\exists\ n_2 \in NodeAddr, \langle n_2, h, ConfirmMsg\ \langle v_2, nsigs_2 \rangle\rangle \in P_{rcv} \tag{4.6}$$

and by (1) and equation (4.4):

$$sig_1 = sign(v_1, secret\_key(n)) \tag{4.7}$$

$$sig_2 = sign(v_2, secret\_key(n)) \tag{4.8}$$

Now the invariant logic branches out depending on whether each of n_1 and n_2 is Byzantine, so we consider four cases.

Case 1: n_1 and n_2 are honest: Note that $P_{rcv} \subseteq P_{sent}$. Then by (2):

$$cert(\triangle(n_1)) = \langle v_1, nsigs_1 \rangle \tag{4.9}$$

$$cert(\triangle(n_2)) = \langle v_2, nsigs_2 \rangle \tag{4.10}$$

Then by (3) and equation (4.4):

$$\langle n, n_1, SubmitMsg\ \langle v_1, sig_1 \rangle\rangle \in P_{rcv} \wedge sig_1 = sign(v_1, secret\_key(n)) \tag{4.11}$$

$$\langle n, n_2, SubmitMsg\ \langle v_2, sig_2 \rangle\rangle \in P_{rcv} \wedge sig_2 = sign(v_2, secret\_key(n)) \tag{4.12}$$

Now suppose for the sake of contradiction that $n \in$ Honest. Then by (5), and equations (4.9) and (4.10), it follows that $v_1 = v_2$ which contradicts equation (4.3) (hence contradicts the specification of proof). Therefore, it must be that $n \in$ ByzAddr.

Case 2: n_1 is honest, n_2 is Byzantine.

Then by the same logic as in **Case 1**, by (2):

$$cert(\triangle(n_1)) = \langle v_1, nsigs_1 \rangle \tag{4.13}$$

Then by (3) and equation (4.4):

$$\langle n, n_1, SubmitMsg\ \langle v_1, sig_1 \rangle\rangle \in P_{rcv} \wedge sig_1 = sign(v_1, secret\_key(n)) \tag{4.14}$$

Suppose n is honest. Then by (4), since n is in the certificate c_2 fabricated by a Byzantine node, by equation (4.8) its signature is valid, and by assumption n is honest, there must be a corresponding submit message from n to some other node n' for the value v_2 in the *packet soup*:

$$\exists n' \in \mathit{NodeAddr},\ \langle n, n', \mathit{SubmitMsg}\ \langle v_2, sig_2\rangle\rangle \in P_{sent} \tag{4.15}$$

From equations (4.15), (4.8) and (4.14), and by (5), it follows that $v_1 = v_2$ which contradicts equation (4.3). Therefore, it must be that $n \in \mathsf{ByzAddr}$

Case 3: n_1 is Byzantine, n_2 is honest. Holds by symmetry with **Case 2**. Simply swap n_1 and n_2 in the proof above.

Case 4: n_1 and n_2 are Byzantine. Suppose n is honest. Then by (4) and equations (4.7), (4.8):

$$\exists n' \in \mathit{NodeAddr},\ \langle n, n', \mathit{SubmitMsg}\ \langle v_1, sig_1\rangle\rangle \in P_{sent} \tag{4.16}$$

$$\exists n'' \in \mathit{NodeAddr},\ \langle n, n'', \mathit{SubmitMsg}\ \langle v_2, sig_2\rangle\rangle \in P_{sent} \tag{4.17}$$

Now by (4.7), (4.8), the assumption that n is honest, and (5), it must be that $v_1 = v_2$. However, this contradicts equation (4.3). Hence, $n \in \mathsf{ByzAddr}$.

□

The proof is structured such that it traces back the "actions" of the accused node n, which is enabled by the structure of the invariant. The possibilities for "precedent actions" branch out depending on whether the node that sent either of the conflicting certificates to h is Byzantine or honest. This makes the proof repetitive and suitable for mechanisation.

Furthermore, note that in **Case 1** we could prove the stronger property *AccSoundness'* because clearly, we get a direct proof for the fact that n sent messages for two different values with valid signatures to two (not necessarily different) nodes, which is how we defined Byzantine behaviour. However, in all other cases at least one of the senders of conflicting certificates is Byzantine, which means that the sender could have included a forged signature of another Byzantine node in its certificate, without the latter sending a corresponding *submit* message. This way, a Byzantine node can be detected through a proof of culpability without having behaved Byzantine. The detected node, if its address comes from the set of Byzantines, might have even conscientiously followed the ABC protocol, but having failed to keep its secret key secret, its signature might have been forged by other Byzantines. By this argument, we could refine the notion of Byzantine behaviour for the ABC protocol to include a node sharing its secret key with other nodes (possibly only Byzantine nodes).[12] We could conjecture that for Byzantine behaviour defined in such way *Acc-Soundness'* should hold, yet we would need to redefine our network semantics to model key sharing.

[12]In addition to the Byzantine behaviour of sending conflicting messages (i.e., failing to follow the rules of the Accountable Confirmer).

Chapter 5

Conclusion and Future Work

Byzantine Consensus protocols are notoriously difficult to reason about, yet we need formal guarantees of their *safety*, ideally even in the worst-case scenario of Byzantine nodes overpopulating the network and exhibiting arbitrary behaviour. The motivation of this project is to obtain the key *safety* guarantees for the ABC protocol by rigorously modelling a system in which a set of correct nodes follow the protocol in the presence of Byzantine adversaries. The latter can violate the protocol rules, collude with each other, and behave in arbitrary ways subject to the constraints imposed by the cryptographic methods.

We began with a TLA+ model, which included a single Byzantine transition and did not model the cryptography used, hence underapproximating the space of Byzantine actions. However, a more accurate model would be unfeasible due to concerns about the exponential growth of state space, which we should avoid to enable model checking. Yet the simple TLA+ model allowed us to formulate the relevant properties and model check them for fixed parameters.

To obtain a more accurate model suitable for sound verification we chose to represent ABC semantics as an inductively specified system with asynchronous message passing and digital signatures from Public Key Cryptography. This enabled more rigorous modelling of Byzantine behaviour, and a refined statement of the *Accountability-Soundness* property. We presented a proof of the inductive invariant and a proof of *Accountability-Soundness* from the invariant.

However, although we chose to focus on the most essential *safety* property which guarantees that the accountability mechanism never accuses correct nodes, there are other important properties we would expect from an accountable protocol. Some of these properties, such as the ones presented in chapter 1, in particular the original *accountability* property from (Civit et al., 2022), could be proved given an appropriate logic for reasoning about *liveness*. However, the project shows that certain desirable *accountability* properties do not hold for ABC. This raises the question of the feasibility of stronger notions of *accountability*. We briefly discuss some future work that could address the above-mentioned problems.

5.1 Mechanised Proofs in Coq Proof Assistant

The model presented in chapter 4 was designed with a Coq Proof Assistant formalisation in mind.[13] The transitions, modeled in relational style in this project, can be translated to executable functions, and we can define possible system configurations inductively with a system_step proposition asserting a relation between a valid configuration w and a subsequent configuration w'. Then we can write machine-checked proofs for the inductive invariant (from the system_step proposition) and for *Accountability-Soundness* (from the inductive invariant). While the Coq Proof Assistant encoding is not part of our contribution to this project, it has been done independently by following our formalisation.

Furthermore, there is work being done to prove the *Accountability* property from (Civit et al., 2022). The property asserts that if two honest nodes confirm different values, then eventually all honest nodes prove the culpability of at least

[13]Coq Proof Assistant provides a formal language to write definitions and theorems as well as an environment for machine-checked proofs.

$t_0 + 1$ nodes. We can define "eventually" as "once all messages in the *packet soup* sent from and addressed to honest nodes have been delivered," and show that if the latter is the case then any honest node can extract a proof of culpability of at least $t_0 + 1$ nodes from the certificates it has collected. To prove the above, we would need to extend the inductive invariant with a few more propositions, including a proposition asserting that if an honest node has confirmed, then it must have broadcasted a certificates with $N - t_0$ signatures.

Finally, work is also being done to generalise the framework for modelling Byzantine behaviour and apply it to other accountable distributed system protocols, such as PeerReview (Haeberlen et al., 2007).

5.2 Faithfully Modelling Cryptographic Schemes

As demonstrated in chapter 4, assumptions about the cryptographic methods can play a key role in determining what the adversaries can do, and hence which *safety* properties hold. Furthermore, some actions that violate a signature scheme based on Public Key Cryptography, such as sharing one's secret key, could be considered Byzantine. Ideally, we would like to have a more comprehensive model of ABC where the message exchanges intended for setting up the scheme, and the message exchanges intended to compromise its security, are included.

However, in reality the security guarantees of most cryptographic schemes used in practice are only probabilistic. For instance, with negligible though non-zero probability the adversary can guess the sender's secret key and use it to impersonate the sender. Therefore, the *safety* properties of any distributed protocol that uses digital signatures or encryption will only hold with large probability. For mechanised and composable proofs of the (probabilistic) security of underlying schemes one could use the SSProve library in Coq Proof

Assistant (Haselwarter et al., 2021).

5.3 Stronger Notions of Accountability

Finally, a question that naturally arises from the conclusion of this verification effort is whether there exists an accountable protocol that satisfies *completeness*. Intuitively, *completeness* of an accountable protocol means that we can detect all nodes that have misbehaved, and hence prove the honesty of the remaining nodes, which seems to be impossible. If that is the case, it would be interesting to model a general accountability protocol and prove that it cannot be *complete*.

Furthermore, *completeness* is always relative to our definition of Byzantine behaviour; for instance the definition adopted by the authors of ABC does not include *delay* or *omission*, i.e., nodes deliberately refusing to send messages. If the majority of nodes were to behave this way, the ABC protocol execution would hang forever. It would be interesting to research for which definitions of Byzantine behaviour (if any) we can achieve *completeness*.

Bibliography

Castro, Miguel and Barbara Liskov (1999). “Practical Byzantine Fault Tolerance”. In: *OsDI*. Vol. 99. 1999, pp. 173–186.

Civit, Pierre, Seth Gilbert, and Vincent Gramoli (2021). “Polygraph: Accountable Byzantine Agreement”. In: *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, pp. 403–413.

Civit, Pierre, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, and Jovan Komatovic (2022). “As easy as ABC: Optimal (A)ccountable (B) yzantine (C) onsensus is easy!” In: *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, pp. 560–570.

Haeberlen, Andreas, Petr Kouznetsov, and Peter Druschel (2007). “PeerReview: Practical Accountability for Distributed Systems”. In: *ACM SIGOPS operating systems review* 41.6, pp. 175–188.

Haselwarter, Philipp G, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Catalin Hritcu, Kenji Maillard, and Bas Spitters (2021). “SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq”. In: *Cryptology ePrint Archive*.

Lamport, Leslie (2003). *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional.

Lamport, Leslie, Robert Shostak, and Marshall Pease (2019). “The Byzantine Generals Problem”. In: *Concurrency: The Works of Leslie Lamport*. New York, NY, USA: Association for Computing Machinery, 203–226. ISBN: 9781450372701. Available from: https://doi.org/10.1145/3335772.3335936.

Nakamoto, Satoshi (2008). *Bitcoin: A Peer-to-Peer Electronic Cash System.* https://bitcoin.org/bitcoin.pdf. Accessed: March 3, 2023.

Yin, Maofan, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham (2019). "HotStuff: BFT Consensus with Linearity and Responsiveness". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pp. 347–356.

Appendix A

Accountable Confirmer

Given N processes, $t_0 = \lceil \frac{n}{3} \rceil - 1$, and $i \in \{1, \ldots, N\}$ the full code for a process P_i, as described by Civit et al. (2022), is:

```
upon event ⟨Init⟩ do
        $value_i \leftarrow \top$
        $confirmed_i \leftarrow false$
        $from_i \leftarrow \emptyset$
        $lightCertificate_i \leftarrow \emptyset$
        $fullCertificate_i \leftarrow \emptyset$
        $obtainedLightCertificates_i \leftarrow \emptyset$
        $obtainedFullCertificates_i \leftarrow \emptyset$
upon event ⟨Submit | v⟩ do
        $value_i \leftarrow v$
        trigger $\langle Broadcast \mid [\text{SUBMIT}, v, ShareSign_i(v)]_{\sigma_i} \rangle$
```

```
upon event ⟨Deliver | $P_j$, $[\text{SUBMIT}, value, share]_{œ_j}$⟩ do
        if $ShareVerify_j(value, share) = \top$ and $value = value_i$ and $P_j \notin from_i$ then
            $from_i \leftarrow from_i \cup \{P_j\}$
            $lightCertificate_i \leftarrow lightCertificate_i \cup \{share\}$
            $fullCertificate_i \leftarrow fullCertificate_i \cup \{[\text{SUBMIT}, value, share]_{\sigma_j}\}$
upon event $|from_i| \geq n - t_0$ do
        $confirmed_i \leftarrow true$
        trigger ⟨$Confirm$ | $value_i$⟩
        trigger ⟨$Broadcast$ | [LIGHT – CERTIFICATE, , $value_i$, $Combine(lightCertificate_i)$]⟩
upon event ⟨Deliver | $P_j$, [LIGHT – CERTIFICATE, , $value_j$, $lightCertificate_j$]⟩ do
        if $lightCertificate_j$ is a valid light certificate then
            $obtainedLightCertificates_i \leftarrow obtainedLightCertificates_i \cup \{lightCertificate_j\}$
upon event $c_1, c_2 \in obtainedLightCertificates_i$, where $c_1$ conflicts with $c_2$
        trigger ⟨$Broadcast$ | [FULL – CERTIFICATE, , $value_i$, $fullCertificate_i$]⟩
upon event ⟨Deliver | $P_j$, [FULL – CERTIFICATE, , $value_j$, $fullCertificate_j$]⟩ do
        if $fullCertificate_j$ is a valid light certificate then
            $obtainedFullCertificates_i \leftarrow obtainedFullCertificates_i \cup \{fullCertificate_j\}$
upon event $c_1, c_2 \in obtainedFullCertificates_i$, where $c_1$ conflicts with $c_2$
        $proof \leftarrow$ extract a proof of culpability of (at least) $t_0 + 1$ processes from $c_1$ and $c_2$
        $F \leftarrow$ set of processes detected via $proof$
        trigger ⟨$Detect$ | $F$, $proof$⟩
```

Note that the *Accountable Confirmer* splits *certificates* into *light certificates* and *full certificates*. The latter contain the received messages together with their respective signatures, while the former are collections of *threshold signatures*, included in each *submit* message, which can then be combined into a single signature before broadcasting the certificate to other nodes. Then there is a preliminary *light certificate* exchange round, with communication complexity of $O(N)$, and only if two conflicting *light certificates* are found the node broadcasts its *full certificate*, which has communication complexity of $O(N^2)$. The reason why a round of *full certificate* exchange is necessary to find evidence for misbehaviour is that nodes detected from a conflict of *light certificates* might be prone to false positives. This is because *threshold signatures* might be forged in case the number of byzantine processes exceeds t_0.