The Government of the Russian Federation

Federal State Autonomous Institution for Higher Professional Education
National Research University Higher School of Economics

St. Petersburg Branch

St. Petersburg School of Physics, Mathematics and Computer Science

Andrei Tonkikh

BYZANTINE FAULT-TOLERANT DISTRIBUTED COMPUTING: RECONFIGURATION AND CONSENSUS

Master dissertation

Area of studies *01.04.02 «Applied Mathematics and Informatics»*

Master Program “Software Development and Data Analysis”

Reviewer
PhD, Senior Researcher,
VMware Research
Ittai Abraham

Research Supervisor
PhD, Associate Professor,
Department of Informatics
Denis Moskvin

Consultant
PhD, Professor,
Télécom Paris,
Institut Polytechnique de Paris
Petr Kuznetsov

Saint Petersburg – 2021

Contents

Introduction **5**

1. Asynchronous Reconfiguration **10**

1.1. Model assumptions 10
1.1.1. Processes and channels 10
1.1.2. Configuration lattice 11
1.1.3. Forward-secure digital signatures 12
1.2. Reconfiguration examples and challenges 13
1.2.1. Reconfiguration example 13
1.2.2. The “I still work here” attack 15
1.2.3. The “slow reader” attack 18
1.3. Abstractions and definitions 18
1.3.1. Access control and object composition 18
1.3.2. Definition of Byzantine lattice agreement 19
1.3.3. Definition of reconfigurable objects 20
1.3.4. Definition of dynamic objects 21
1.3.5. Quorum system assumptions 23
1.3.6. Broadcast primitives 23
1.4. Dynamic Byzantine Lattice Agreement 24
1.4.1. Client implementation 26
1.4.2. Replica implementation 29
1.4.3. Implementing other dynamic objects 31
1.5. Reconfigurable objects 32
1.5.1. Implementation 32
1.5.2. Proof of correctness 34
1.5.3. Discussion 35
1.6. Access control 37
1.6.1. Sanity-check approach 38
1.6.2. Quorum-based approach (“on-chain governance”) . . . 39
1.6.3. Trusted administrators 40
1.6.4. Combining Access Control with other objects 40

1.7. Related work 41
1.8. Discussions 43

2. Efficient Byzantine Fault-Tolerant Consensus 46
2.1. Preliminaries 46
2.1.1. Model assumptions 46
2.1.2. The consensus problem 47
2.2. Algorithm 48
2.2.1. Normal case 49
2.2.2. View change 50
2.2.3. Proof of consistency 52
2.2.4. Generalized version 55
2.3. Lower bound 56
2.3.1. Preliminaries 56
2.3.2. Optimality of the proposed algorithm 60
2.3.3. Optimality of FaB Paxos 66
2.4. Related work 67

Conclusion 69

References 70

A. Reconfiguration Appendix 78
A.1. Pseudocode for the DBLA implementation 78
A.2. Correctness proof of the DBLA implementation 83
A.2.1. Safety 83
A.2.2. Liveness 89
A.3. Possible optimizations for the DBLA implementation 93
A.4. Max Register 94
A.4.1. Dynamic Max-Register implementation 94
A.4.2. Proof of correctness 98

Abstract

Byzantine fault-tolerance is a critical mechanism allowing software engineers to build distributed systems resilient to numerous kinds of hardware failures, software bugs, and security breaches. Depending on the environment, different models are used to design distributed protocols. Notably, there is a huge gap between the so-called asynchronous and partially-synchronous models. The latter allows for deterministic solutions of the famous consensus problem whereas the former does not. In this thesis, one important problem for each of the two models is addressed, the ultimate goal being to make both types of algorithms more applicable in practice.

In the asynchronous model, the problem of reconfiguration is considered. Most known Byzantine fault-tolerant algorithms are designed with the assumption that the set of processes executing the protocol is fixed and known a priori. While being very convenient for algorithm designers, this assumption is not very realistic. In practice, servers often need to be replaced, repaired, or added for greater fault tolerance. In this thesis, the first asynchronous reconfiguration protocol tolerating Byzantine failures is presented.

In the partially-synchronous model, on the other hand, reconfiguration can be performed much easier thanks to the possibility to solve consensus. However, Byzantine fault-tolerant consensus algorithms have a reputation of being much slower than their crash fault-tolerant counterparts. To close this gap, the class of fast Byzantine consensus algorithms has emerged. These algorithms aim at solving the Byzantine consensus with the same latency as crash fault-tolerant solutions. However, this improved latency comes at the cost of decreased fault tolerance.

In this thesis, a new fast Byzantine consensus algorithm that has better resilience than any of the previously existing ones is presented. Moreover, the algorithm comes with proof that its resilience is optimal. The surprising discovery is that, in the special yet very important in practice case when only one Byzantine failure is tolerated, the algorithm presented in this thesis requires only 4 processes, which remains optimal even if we allow for suboptimal latency. All previously known fast Byzantine consensus algorithms required at least 6 processes in this case.

Introduction

A long-lasting vision in distributed computing is to build reliable systems out of unreliable components. Hence, the notion of fault tolerance was invented. The premise is that faults are uncorrelated and it is very unlikely that more than a certain number of processes will fail at the same time.[1] This number is usually denoted by f and is called *the resilience threshold*. The total number of processes is usually denoted by n. The minimum value of n that is required to tolerate f failures characterizes the *resilience* of the protocol (typical values are $n = 2f + 1$ and $n = 3f + 1$).

However, there are many kinds of possible errors that may occur. Starting from a minor slowdown due to garbage collection all the way to a security breach that allows a hacker to fully control one or multiple processes. The simplest and most common assumption is that a faulty process simply stops taking any actions and does not send messages not prescribed by the protocol. Algorithms that are designed with this assumption in mind are called *crash fault-tolerant*. It works well for simple and easily detectable malfunctions such as hard drive or network failures.

However, in practice, much more sophisticated problems occur sometimes. In order to incorporate these kinds of problems, the notion of *Byzantine fault tolerance* was invented [65]. A Byzantine fault-tolerant algorithm assumes that up to f processes are completely controlled by a malicious adversary that is trying to break the system (i.e., to force it to violate one of its safety or liveness properties [11]). The processes under the control of the adversary (called *faulty* or *Byzantine*) are allowed to violate the protocol in arbitrary ways, including coordinated attacks. However, they also may choose to act as correct processes in order to conceal themselves. It is, therefore, impossible to reliably determine whether a specific process is Byzantine unless this process violates the protocol in a detectable way. The non-Byzantine processes are called *correct* and are assumed to follow the prescribed protocol at all times.

In the recent years, under the name of Blockchain, Byzantine fault-tolerance has been widely deployed as a mechanism for building peer-to-peer systems

[1] To avoid correlated faults, multiversion programming is sometimes used [26, 25, 2].

connecting thousands of people without requiring the users to trust each other or any third party.

Apart from the processes themselves, a distributed system is also defined by the network that interconnects them. The type of network used in a distributed system often has an even greater impact on the algorithms and the set of problems that can be solved in that system then the types of process failures that may occur. There is an especially large gap between the so-called asynchronous and partially-synchronous network models. As was proven by Fischer, Lynch, and Paterson in the famous paper [35], if the network connecting the processes is *asynchronous*, meaning that there is no finite upper bound on how long it may take for a message to arrive, then it is impossible to solve the so-called consensus problem in finite time even if just one process may fail by crashing.

The *consensus* problem [65] allows the processes to unambiguously agree on a single value. More formally, every correct process starts with some input value, executes the prescribed protocol, and has to output a single output value. All correct processes have to output the same value, and this value has to be suggested by at least one process. Consensus turns out to be an extremely important problem as it allows to build a *replicated state machine* [52], which, in turn, allows to implement a linearizable distributed version of any data type that has a sequential specification [41].

Reconfiguration. Traditionally, distributed algorithms are designed with the assumption that the set of processes executing the protocol is fixed and is known a priori. This assumption contradicts one of the main goals of fault tolerance: to build systems that will be able to run for very long periods of time (or even indefinitely).

One of the main appeals of state machine replication is that it can be easily reconfigured (at least, in crash fault-tolerant systems) [57]. For a long time it was believed that asynchronous systems cannot be reconfigured without using consensus. However, it was later refuted by DynaStore [8]. A long line of research on asynchronous reconfiguration protocols followed [8, 42, 69, 70, 49]. However, the only reconfiguration protocol capable of withstanding Byzantine faults [61] required a central trusted administrator to execute the reconfigura-

tion and to generate new private keys for all replicas on each reconfiguration. Of course, with the assumption of a trusted administrator, this solution cannot be considered fully Byzantine fault-tolerant, neither does it capture the spirit of asynchronous reconfiguration as it relies on a totally ordered sequence of configurations.[2]

In this thesis, a novel reconfiguration technique capable of tolerating Byzantine failures is presented. The algorithm has optimal resilience ($n \geq 3f+1$) and asymptotically optimal running time ($O(k)$ message delays for k concurrently proposed reconfiguration requests), and it does not rely on a trusted administrator.

Synchronous and partially-synchronous models. In the *synchronous* model, there is a known upper bound Δ on the maximum time that it takes for a message to travel from one correct process to another and be processed. However, in practice, even small and carefully crafted networks may have occasional partitions, downtime, or periods of instability. An algorithm designed in the synchronous model may break in such a case unless Δ is large enough so that any breakdown is repaired within that time. Since the dependency of synchronous algorithms usually depends on Δ, this creates an unpleasant trade-off: one has to either choose Δ very large to account for potential network issues and significantly slow down the protocol, or choose Δ closer to the normal network delay and risk system breakdown in case of an unanticipated large network delay.

One way to mitigate this trade-off is to design the algorithm in the *partially-synchronous* model, which also assumes the existence of a known upper bound Δ on the maximum message delay, but allows for periods of instability when this upper bound does not hold. More formally, in the partially-synchronous model, it is assumed that there is an unknown moment in time, called *the global stabilization time* (*GST* for short), such that after that moment every message sent by a correct process to another correct process is delivered and processed within the time period Δ.

Formally, this assumption requires an infinitely large period of network sta-

[2]Note that [61] was published before the groundbreaking work of Aguilera et al. [8] that demonstrated the possibility of asynchronous reconfiguration.

bility after GST. However, in practice, it is sufficient that the period of stability is large enough for the protocol to make progress. Note that the processes do not know and have no way to detect whether the period of stability has already started. In other words, if a process sends a message to another process and does not receive a reply within time period 2Δ, the process has no way to determine whether the addressee is faulty or the period of stability just have not started yet.

Fast Byzantine consensus. Consensus is one the most frequently-used building blocks for partially–synchronous distributed systems. It is therefore desirable for the consensus algorithms to be as efficient as possible. One of the key performance characteristics is the *good-case latency*. While the worst-case latency of consensus is bound to be at least $\Omega(f)$ [9, 33], most practical partially-synchronous algorithms are capable of terminating in just a constant number of round-trips if some optimistic conditions are met. In particular, Paxos [54, 55] and Viestamped Replication [64, 58] are capable of terminating in just two message delays (i.e., one round-trip) if all correct processes agree on the same *leader* process and this leader is correct. This good-case latency is optimal even for fault-free executions [43].

However, most practical Byzantine fault-tolerant consensus algorithms [24, 71, 21] have good-case latency of 3 or more message delays. This gap leads to an extensive line of research [47, 62, 45, 3, 4, 40] towards *fast Byzantine consensus algorithms* – the class of Byzantine fault-tolerant partially-synchronous consensus algorithms that can reach agreements with the same delay as their crash fault-tolerant counterparts.

FaB Paxos [62] achieves the optimal good-case latency of 2 message delays at the expense of resilience: in order to tolerate f failures, the protocol needs $5f+1$ replicas, which is significantly worse than the optimal number of $3f+1$ replicas for Byzantine consensus [22]. The same paper [62] claims that $5f+1$ is the optimal resilience for a fast Byzantine consensus protocol. This lower bound implies a trade-off between the optimal resilience and the optimal good-case latency of Byzantine consensus. However, the lower bound proof contains a mistake. In fact, as shown in this thesis, the lower bound of $5f+1$ processes only applies to a restricted class of algorithms that assume that the processes are

split into disjoint sets of *proposers* and *acceptors*.

Surprisingly, as shown in this thesis, if the roles of proposers and acceptors are performed by the same set of processes, there exists a fast f-resilient Byzantine consensus protocol that requires only $5f - 1$ processes. By adding a "slow path" [47, 62, 4] one can obtain a generalized version of the protocol that requires $n = \max\{3f + 2t - 1, 3f + 1\}$ processes in order to be able to tolerate f Byzantine failures and remain fast (terminate in two message delays) in the presence of up to t Byzantine failures. In particular, this is the first protocol that is able to remain fast in presence of a single Byzantine failure ($t = 1$) while maintaining the optimal resilience ($n = 3f + 1$).

In this thesis, it is also shown that $n = 3f + 2t - 1$ is the true lower bound for the number of processes required for a fast Byzantine consensus algorithm. While for large values of f and t the difference of just two processes may seem insignificant, practical deployments with small f and t can benefit a lot.

In order to avoid a single point of failure in a system while maintaining the optimal good-case latency ($f = t = 1$), the protocol presented in this thesis requires only 4 processes (as opposed to 6, required by prior protocols), which coincides with the $3f + 1$ lower bound on the number of processes for any partially-synchronous Byzantine consensus algorithm.

Goals and objectives. The goal of this thesis was to study important problems in different models of Byzantine fault-tolerant distributed computing.

The following objectives were addressed:

- Formalize the problem of asynchronous reconfiguration in the model with Byzantine faults and find a solution;
- Rigorously prove the correctness of the resulting protocol;
- Implement reconfigurable Byzantine fault-tolerant atomic Max-Register;
- Create a fast Byzantine consensus algorithm that requires just $n = 5f - 1$ replicas to tolerate f failures and prove its correctness;
- Prove a matching lower bound on the resilience of fast Byzantine consensus.

1 Asynchronous Reconfiguration

In this chapter, the solution for the problem of asynchronous Byzantine fault-tolerant reconfiguration is presented. The main results presented in this chapter were previously published in the proceedings of DISC 2020 [50].

The rest of the chapter is organized as follows. The model assumptions are formally stated in Section 1.1. Definitions for the principal abstractions are given in Section 1.3. In Section 1.4, the implementation of the Dynamic Byzantine Lattice Agreement is described, and, in Section 1.5, it is shown how to use Dynamic Byzantine Lattice Agreement to implement reconfigurable objects. Possible implementations of access control are discussed in Section 1.6. Related work is discussed in Section 1.7. Section 1.8 contains discussions on the applicability of the presented algorithms, further research that has been done partially on the basis of this work, and interesting open problems about asynchronous Byzantine fault tolerant reconfiguration.

To make the presentation easier to follow, complete pseudocode and proofs of correctness for the Dynamic Byzantine Lattice Agreement abstraction are delegated to Appendices A.1 and A.2. Potential directions for optimizations of the Dynamic Byzantine Lattice Agreement implementation are suggested in Appendix A.3 Finally, as an application of the proposed constructions, an implementation of a dynamic Max-Register is provided in Appendix A.4.

1.1 Model assumptions

1.1.1 Processes and channels

In this chapter, two types of processes are considered: *replicas* and *clients*. Let Φ and Π denote the (possibly infinite) sets of replicas and clients, resp., that potentially can take part in the computation. At any point in a given execution, a process can be in one of the four states: *idle*, *correct*, *halted*, or *Byzantine*. A process is *idle* if it has not taken a single step in the execution yet. A process stops being idle by taking a step, e.g., sending or receiving a message. A process is considered *correct* as long as it respects the algorithm it is assigned. A process is *halted* if it executed the special "halt" command and stopped taking further

steps. Finally, a process is *Byzantine* if it prematurely stops taking steps of the algorithm or takes steps that are not prescribed by it. A correct process can later halt or become Byzantine. However, the reverse is impossible: a halted or Byzantine process cannot become correct. A process that remains correct forever is said to be *forever-correct*.

In this chapter, the network consisting of asynchronous reliable authenticated point-to-point links between all pairs of processes is considered [22]. If a forever-correct process p sends a message m to a forever-correct process q, then q eventually delivers m. Moreover, if a correct process q receives a message m from a process p at time t, and p is correct at time t, then p has indeed sent m to q before t.

The adversary is assumed to be computationally bounded so that it is unable to break the cryptographic techniques, such as digital signatures, forward security schemes [18] and one-way hash functions.

1.1.2 Configuration lattice

A *join semi-lattice* is a tuple $(\mathcal{L}, \sqsubseteq)$, where $\mathcal{L}$ is a set partially ordered by the binary relation $\sqsubseteq$ such that for all elements $x, y \in \mathcal{L}$, there exists the *least upper bound* for the set $\{x, y\}$, i.e., the element $z \in \mathcal{L}$ such that $x, y \sqsubseteq z$ and $\forall\, w \in \mathcal{L}$: if $x, y \sqsubseteq w$, then $z \sqsubseteq w$. The least upper bound for the set $\{x, y\}$ is denoted by $x \sqcup y$. Operator $\sqcup$ is called the *join operator*. It is an associative, commutative, and idempotent binary operator on $\mathcal{L}$. $x \sqsubset y$ is used to denote $x \sqsubseteq y$ and $x \neq y$. Elements $x, y \in \mathcal{L}$ are said to be *comparable* iff either $x \sqsubseteq y$ or $y \sqsubset x$. For conciseness, in the rest of this thesis, the word "*lattice*" will be used as a synonym for "join semi-lattice".

For any (potentially infinite) set A, $(2^A, \subseteq)$ is the lattice of all subsets of A, called *the powerset lattice of* A. Note that $\forall Z_1, Z_2 \subseteq A$: $Z_1 \sqcup Z_2 = Z_1 \cup Z_2$.

A configuration is an element of a lattice $(\mathcal{C}, \sqsubseteq)$. Every configuration must be associated with a finite set of replicas via a map *replicas* : $\mathcal{C} \to 2^{\Phi}$, and a *quorum system* via a map *quorums* : $\mathcal{C} \to 2^{2^{\Phi}}$, such that $\forall C \in \mathcal{C}$: *quorums*$(C) \subseteq 2^{\mathit{replicas}(C)}$. Additionally, there must be a function *height* : $\mathcal{C} \to \mathbb{Z}$, such that $\forall C \in \mathcal{C}$: *height*$(C) \geq 0$ and $\forall C_1, C_2 \in \mathcal{C}$: if $C_1 \sqsubset C_2$, then

height$(C_1) <$ *height*(C_2). A configuration C is said to be *higher* (resp., *lower*) than a configuration D iff $D \sqsubset C$ (resp, $C \sqsubset D$).[3]

The set *quorums*(C) is said to be a *dissemination quorum system* at time t iff every two sets (also called *quorums*) in *quorums*(C) have at least one replica in common that is correct at time t and there is at least one quorum $Q \in$ *quorums*(C) such that all replicas in Q are correct at time t.

Consider the following as an example of a configuration lattice: let *Updates* be $\{+, -\} \times \Phi$, where tuple $(+, p)$ means "add replica p" and tuple $(-, p)$ means "remove replica p". Then $\mathcal{C}$ is the powerset lattice $(2^{Updates}, \subseteq)$. The mappings *replicas*, *quorums*, and *height* are defined as follows: $replicas(C) \triangleq \{s \in \Phi \mid (+, s) \in C \land (-, s) \notin C\}$, $quorums(C) \triangleq \{Q \subseteq replicas(C) \mid |Q| > \frac{2}{3}\,|replicas(C)|\}$, and $height(C) \triangleq |C|$. It is straightforward to verify that *quorums*(C) is a dissemination quorum system when strictly less than one third of replicas in *replicas*(C) are faulty.

Note that, if this configuration lattice is used, once a replica is removed from the system, it cannot be added back with the same identifier. In order to add such a replica back to the system, a new identifier must be used. This, however, does not imply that for any configuration lattice, adding a replica back necessarily involves generating a new identifier.

1.1.3 Forward-secure digital signatures

In a *forward-secure digital signature scheme* [18, 60, 19, 31], the public key of a process is fixed while the secret key can evolve. Each signature is associated with a *timestamp*. To generate a signature with timestamp t, the signer uses secret key sk_t. The signer can *update its secret key* and get sk_{t_2} from sk_{t_1} if $t_1 < t_2$. However, "downgrading" the key to a lower timestamp is computationally infeasible. Hence, if a process removes all copies of its private key with timestamps smaller than t, it will never be able to sign new messages with any timestamp lower than t, even if the process turns Byzantine.

For simplicity, a forward-secure digital signature scheme is modelled as an oracle that associates every process p with a timestamp st_p (initially, $\forall p : st_p =$

[3]Notice that "C is higher than D" implies "*height*$(C) >$ *height*(D)", but not vice versa.

0). The oracle provides p with three operations:

1. UpdateFSKey(t) sets st_p to $t \geq st_p$;

2. FSSign(m, t) returns a signature for message m and timestamp t if $t \geq st_p$, otherwise it returns $\perp$;

3. FSVerify(m, p, s, t) returns *true* iff $s \neq \perp$ was generated by invoking FSSign(m, t) by process p.[4]

From the implementation point of view, one can easily obtain a forward-secure signature scheme from any ordinary signature scheme by using one of the generic constructions [18, 60]. In particular, for applications with potentially large number of reconfiguration requests, the d-ary certificate tree approach due to Bellare and Miner [18] with large values of d may be most suitable. In applications with large quorum sizes, a recently proposed forward-secure *multi-signature* scheme [31] can be used to improve performance and lower the communication cost.

The way forward-secure digital signatures can be used for implementing Byzantine fault-tolerant reconfiguration is described in Section 1.2

1.2 Reconfiguration examples and challenges

Before diving into the technical details, let us first consider a few example executions in order to build the intuition about the desirable properties of the reconfigurable objects and the fundamental challenges that we will have to face in order to implement those properties.

For simplicity, all examples in this section use the configuration lattice $(2^{Updates}, \subseteq)$ described in Section 1.1.2.

1.2.1 Reconfiguration example

As an example of a reconfigurable distributed object, let us consider a single-writer multi-reader Byzantine regular register [22]. One designated client

[4] It is assumed that anyone who knows the id of a process also knows its public key. For example, the public key can be directly embedded into the process identifier.

(called *writer* and denoted as w) has access to the operation Write(v), and every other client has access to the operation Read(). In case Read is not concurrent with a Write, it is supposed to return the last value written before the read. Otherwise, it is allowed to return either the last value written or a concurrently written value.

Suppose that at some moment in time the system is in configuration C and is maintained by a set of 4 replicas: $\{r_1, r_2, r_3, r_4\}$. Replica r_2 is Byzantine, but pretends to be correct for the time being. Now suppose that replica r_1 needs to be temporarily turned off for maintenance. Simply turning off this replica is risky: if r_2 stops processing requests, the system will cease to make progress. Hence, an administrator tries to replace r_1 with replica r_5 and issues a reconfiguration request $R_1 = \{(-, r_1), (+, r_5)\}$. At the same time, due to the lack of coordination, another administrator is trying to replace r_1 with another replica (r_6), and issues a reconfiguration request $R_2 = \{(-, r_1), (+, r_6)\}$.

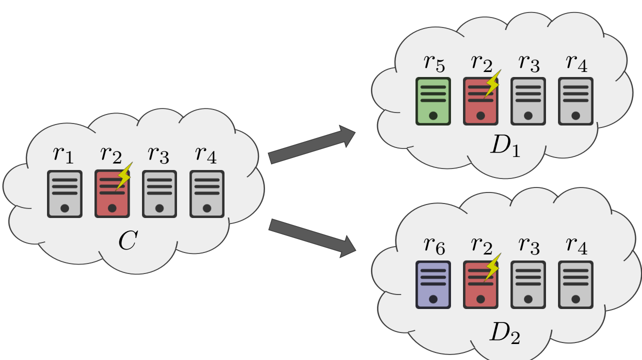


Figure 1.1: Example of a fork.

One imaginable outcome for this scenario (depicted in Figure 1.1) would be for both requests to be processed independently, causing a *fork*: two configurations ($D_1 = C \sqcup R_1$ and $D_2 = C \sqcup R_2$) existing and operating at the same time. It is easy to see why this outcome is undesirable. For example, the quorum $Q_1 = \{r_2, r_3, r_5\}$ in configuration D_1 has no correct processes in the intersection with the quorum $Q_2 = \{r_2, r_4, r_6\}$ in configuration D_2. If w thinks that D_1 is the relevant configuration and writes some value by communicating exclusively with processes in quorum Q_1, and some client q thinks that D_2 is the relevant configuration and reads a value by communicating exclusively with processes in quorum Q_2, then q may not discover the latest value written by w.

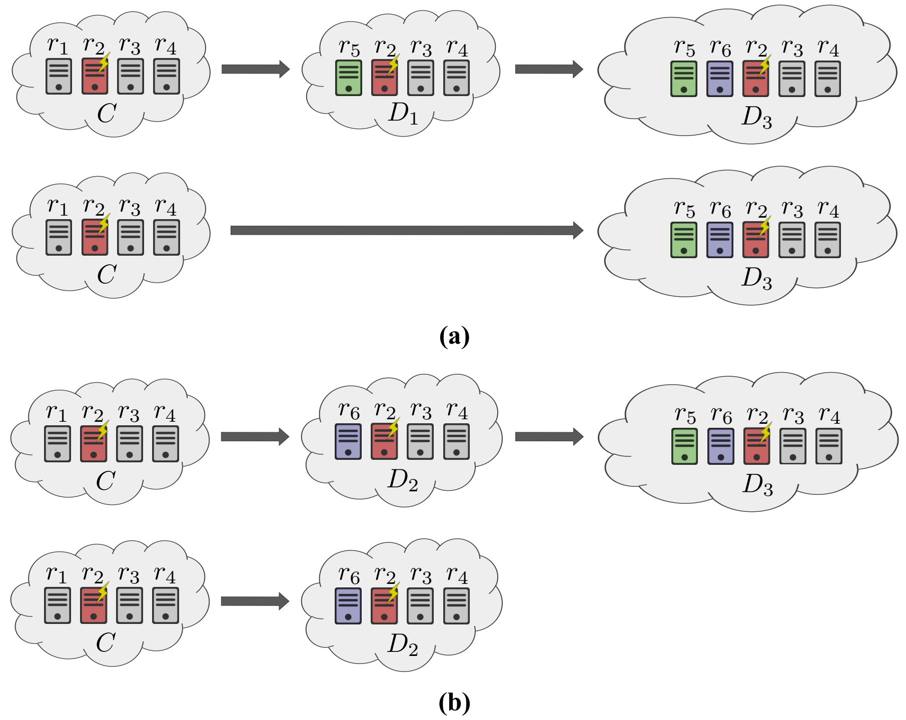


Figure 1.2: Examples of valid combinations of local histories of processes. $D_3 = C \sqcup R_1 \sqcup R_2$.

Ideally, the reconfiguration mechanism should provide an illusion of a single chain of configurations that starts from the initial configuration C^{init}. However, unambiguously agreeing even on a non-trivial prefix of this chain among multiple processes would involve solving consensus, which is impossible in an asynchronous system [35]. Instead, in the system proposed in this thesis, each process has only partial knowledge (a subsequence) of the chain of configurations. These subsequences are called the *local histories* of processes and must be in some sense compatible. In particular, we want to avoid forks. Thanks to the famous lattice agreement abstraction [16, 34], it is possible to guarantee that the histories of correct processes are *related by containment* (i.e., subsets of one another). As long as each individual history is sequential (i.e., all configurations in it are comparable with $\sqsubseteq$), it is sufficient to prevent forks. Some examples of possible combinations of local histories of correct processes are depicted in Figure 1.2.

1.2.2 The "I still work here" attack

A Byzantine fault tolerant system may remain functioning only as long as some assumptions on the set of Byzantine processes are satisfied. In static systems, it

is a common assumption that the fraction of Byzantine replicas does not exceed one-third. More generally, one can assume a dissemination quorum system (as defined in Section 1.1.2). If a dynamic system remains static (i.e., no new reconfiguration requests are made for some time), ideally, we would like to rely only on the correctness of the last installed configuration. More generally, a configuration is said to be *superseded* after a higher configuration is installed. One of the biggest challenges in building reconfigurable Byzantine fault-tolerant systems is to avoid relying on the correctness of superseded configurations.

In the example given before, after r_1 has been removed from the system, we should no longer rely on its correctness. For example, imagine that replica r_1 was replaced because it became known that a hacker attack on this replica was being prepared or because a vulnerability was found in its hardware. Imagine that some client is not aware that configuration C is superseded and tries to execute a Read request in C.[5] Imagine also that replica r_1 had turned Byzantine by that moment and that replica r_3 is not aware of the reconfiguration and still thinks that C is relevant (it may happen due to asynchrony). In this case, Byzantine replicas r_1 and r_2 may collude in order to fool the client into believing that C is still relevant and serve outdated information to the client.

This scenario demonstrates one of the many flavors of the so-called **"I still work here"** attack [7]. In this particular example, it is sufficient to destroy the private key on r_1 after it is removed from the system, before finishing the reconfiguration. However, in general, a more elaborate mechanism is required.

Imagine, for example, that some object is initially maintained by 4 replicas. However, in order to increase fault-tolerance, 6 new replicas are added (see Figure 1.3). This raises the resilience threshold (i.e., the number of Byzantine processes that the system can tolerate) from 1 to 3. However, if 3 out of 4 initial replicas turn Byzantine and some client tries to execute a request in the initial configuration, the Byzantine replicas will be able to fool the client into believing that the initial configuration is still relevant and return an outdated value to the client. This makes our attempts to increase fault-tolerance by adding new replicas futile. Note that there are no correct replicas that can remove their private

[5]Recall that in the asynchronous model we make no assumptions about the relative speed of processes. Hence, it is impossible to make sure that all clients are always caught up with the latest configuration updates.

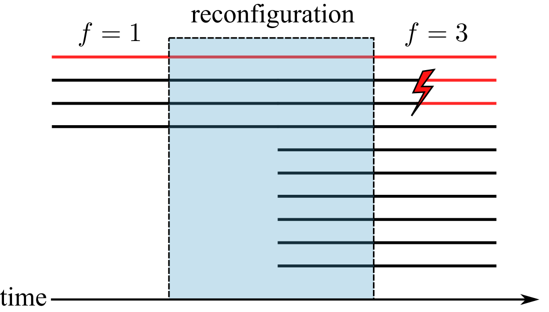


Figure 1.3: The “I still work here” attack.

keys at any point in the execution because no replicas were removed from the system.

In order to prevent this issue, before installing a new configuration, one needs to make sure that the configurations that will be superseded are no longer capable of processing requests from clients. In this thesis, a novel method to implement this functionality is presented. It is based on the same ideas as the “forgetting protocol” described in prior works [61, 7], but it utilizes forward-secure digital signatures in order to avoid relying on a trusted administrator or a total order of configurations.

In the proposed protocol, the height of the configuration is used as the timestamp. When a replica answers requests in configuration C, it signs messages with timestamp $\mathit{height}(C)$. When a higher configuration D is installed, the replica invokes UpdateFSKey$(\mathit{height}(D))$. Therefore, even if the replicas are to become Byzantine in the future, they will not be able to pretend that configuration C is still active. The superseded configuration simply becomes non-responsive, as in crash-fault-tolerant reconfigurable systems.

Unfortunately, in an asynchronous system it is impossible to make sure that replicas of *all* superseded configurations remove their private keys as it would require knowing the prefix of the global history of configurations, which is equivalent to solving consensus. However, as is shown in this thesis, it is possible to make sure that the configurations in which replicas do not remove their keys are never accessed by correct clients and are incapable of creating cryptographic proofs for any statements.

1.2.3 The “slow reader” attack

Finally, there is a more subtle **“slow reader”** attack, which has not been previously studied in the literature. Intuitively, the slow reader attack is a certain way how a reconfiguration request may intertwine with a concurrent client’s request so that the client will be server stale data. As the explanation of this attack involves many technical details, it is postponed to Section 1.4.1, where a detailed example is provided.

1.3 Abstractions and definitions

1.3.1 Access control and object composition

Some abstractions are parametrized by boolean functions VerifyInputValue(v, σ) and / or VerifyInputConfig(C, σ), where σ is called a *certificate*. Moreover, some objects also export a boolean function VerifyOutputValue(v, σ), which lets anyone to verify that the value v was indeed produced by the object. This helps to deal with Byzantine clients. In particular, it achieves three important goals.

First, the parameter VerifyInputConfig allows to prevent Byzantine clients from reconfiguring the system in an undesirable way or flooding the system with excessively frequent reconfiguration requests. In Section 1.6, three simple implementations of this functionality are proposed.

Second, the parameter VerifyInputValue(v, σ) allows to formally capture the application-specific notions of well-formed client requests and access control. For example, in a key-value storage system, each client may be permitted to modify only the key-value pairs that were created by this client. In this case, the certificate σ is just a digital signature of the client.

Finally, the exported function VerifyOutputValue allows to compose several distributed objects in such a way that the output of one object is passed as input for another one. For example, in Section 1.5, one object (*HistLA*) operates exclusively on sets of outputs of another object (*ConfLA*). The parameter function VerifyInputValue of *HistLA* and the exported function VerifyOutputValue of *ConfLA* are used to guarantee that a Byzantine client cannot send to *HistLA*

a value that was not produced by *ConfLA*.

1.3.2 Definition of Byzantine lattice agreement

In this section, the *Byzantine Lattice Agreement* abstraction (BLA for short) is defined. It serves as one of the main building blocks for constructing reconfigurable objects.

Byzantine Lattice Agreement is an adaptation of Lattice Agreement [16, 34] that can tolerate Byzantine failures of processes (both clients and replicas). It is parameterized by a join semi-lattice $\mathcal{L}$, called the *object lattice*, and a boolean function VerifyInputValue : $\mathcal{L} \times \Sigma \to \{true, false\}$, where Σ is a set of possible certificates. The certificate σ is said to be a *valid certificate for input value* v iff VerifyInputValue$(v, \sigma) = true$.

Value $v \in \mathcal{L}$ is said to be a *verifiable input value* in a given run iff at some point in time in that run, some process knows a certificate σ that is valid for v. The adversary must be unable to invert VerifyInputValue by computing a valid certificate for a given value. This is the case, for example, when σ must contain a set of unforgeable digital signatures.

The Byzantine Lattice Agreement abstraction exports one operation and one function:[6]

- Operation Propose(v, σ) returns a response of the form (w, τ), where $v, w \in \mathcal{L}$, σ is a valid certificate for input value v, and τ is a certificate for output value w;

- Function VerifyOutputValue(v, σ) returns a boolean value.

Similarly to input values, τ is said to be a *valid certificate for output value* w iff VerifyOutputValue$(w, \tau) = true$, and w is said to be a *verifiable output value* in a given run iff at some point in that run, some process knows τ that is valid for w.

Implementations of Byzantine Lattice Agreement must satisfy the following properties:

[6]The difference between a function and an operation is that a function can be computed locally, without communicating with other processes, and the result only depends on the function's input.

- *BLA-Validity*: Every verifiable output value w is a join of some set of verifiable input values;
- *BLA-Verifiability*: If Propose$(\ldots)$ returns (w, τ) to a correct process, then VerifyOutputValue$(w, \tau) =$ *true*;
- *BLA-Inclusion*: If Propose(v, σ) returns (w, τ) to a correct process, then $v \sqsubseteq w$;
- *BLA-Comparability*: All verifiable output values are comparable;
- *BLA-Liveness*: If the total number of verifiable input values is finite, every call to Propose(v, σ) by a forever-correct process eventually returns.

For the sake of simplicity, liveness is only guaranteed when there are finitely many verifiable input values. In practice, this assumption boils down to providing liveness iff there are sufficiently large periods of time when no new values are proposed. The abstraction that provides unconditional liveness is called *Generalized Lattice Agreement* [34].

1.3.3 Definition of reconfigurable objects

It is possible to define a *reconfigurable* version of every static distributed object by enriching its interface and imposing some additional properties. In this section, the notion of a reconfigurable object is defined in a very abstract way. By combining this definition with the definition of a Byzantine Lattice Agreement, one can obtain a formal definition of a Reconfigurable Byzantine Lattice Agreement. Similar combination can be performed with the definition of any static distributed object (e.g., with the definition of a Max-Register from Appendix A.4).

A reconfigurable object exports an operation UpdateConfig(C, σ), which can be used to reconfigure the system, and must be parameterized by a boolean function VerifyInputConfig : $\mathcal{C} \times \Sigma \to \{$*true*,*false*$\}$, where Σ is a set of possible certificates. Similarly to verifiable input values, configuration $C \in \mathcal{C}$ is said to be a *verifiable input configuration* in a given run iff at some point in that run, some process knows σ such that VerifyInputConfig$(C, \sigma) =$ *true*.

The total number of verifiable input configurations is required to be finite in any given infinite execution of the protocol. In practice, this boils down to assuming sufficiently long periods of stability when no new verifiable input configurations appear. Similar requirements are imposed by all asynchronous reconfigurable storage systems [7, 70, 49, 10].

When a correct replica r is ready to serve user requests in a configuration C, it triggers callback InstalledConfig(C). Replica r is said to *install* configuration C. At any given moment in time, a configuration is called *installed* if some correct replica has installed it, and it is called *superseded* if some higher configuration is installed.

Each reconfigurable object must satisfy the following properties:

- *Reconfiguration Validity*: Every installed configuration C is a join of some set of verifiable input configurations. Moreover, all installed configurations are comparable;
- *Reconfiguration Liveness*: Every call to UpdateConfig(C, σ) by a forever-correct client eventually returns. Moreover, C or a higher configuration will eventually be installed;
- *Installation Liveness*: If some configuration C is installed by some correct replica, then every correct replica will eventually install C or a higher configuration.

1.3.4 Definition of dynamic objects

Reconfigurable objects are hard to build because they need to solve two problems at once. First, they need to order and combine concurrent reconfiguration requests. Second, the state of the object needs to be transferred across installed configurations. The two problems are decoupled by the notion of a *dynamic* object. Dynamic objects solve the second problem while "outsourcing" the first one.

In Section 1.1, the configuration lattice $\mathcal{C}$ was introduced. A finite set $h \subseteq \mathcal{C}$ is called a *history* iff all elements of h are comparable (i.e., $\forall C_1, C_2 \in h$: $C_1 \sqsubseteq C_2$ or $c_2 \sqsubseteq C_1$). Let HighestConf(h) be $C \in h$ such that $\forall\, C' \in h : C' \sqsubseteq C$.

By definition of a history, HighestConf(h) is unambiguously defined for any history h.

Dynamic objects must export an operation UpdateHistory(h, σ) and must be parameterized by a boolean function VerifyHistory : $\mathcal{H} \times \Sigma \to \{$*true*,*false*$\}$, where $\mathcal{H}$ is the set of all histories and Σ is the set of all possible certificates. History h is said to be a *verifiable history* in a given execution iff at some point in that execution, some process knows σ such that VerifyHistory$(h, \sigma) =$ *true*. A configuration C is called *candidate* iff it belongs to some verifiable history. Also, a candidate configuration C is called *active* iff it is not superseded by a higher configuration.

As with verifiable input configurations, the total number of verifiable histories is required to be finite. Additionally, all verifiable histories must be related by containment (i.e., comparable w.r.t. $\subseteq$). Formally, if VerifyHistory$(h_1, \sigma_1) =$ *true* and VerifyHistory$(h_2, \sigma_2) =$ *true*, then $h_1 \subseteq h_2$ or $h_2 \subseteq h_1$. The technique that can be used to build such histories is discussed in Section 1.5.

Similarly to reconfigurable objects, a dynamic object must have the InstalledConfig(C) callback. The object must satisfy the following properties:

- *Dynamic Validity*: Only a candidate configuration can be installed by a correct replica;

- *Dynamic Liveness*: Every call to UpdateHistory(h, σ) by a forever-correct client eventually returns. Moreover, HighestConf(h) or a higher configuration will eventually be installed;

- *Installation Liveness* (the same as for reconfigurable objects): If some configuration C is installed by some correct replica, then C or a higher configuration will eventually be installed by all correct replicas.

Note that Dynamic Validity implies that all installed configurations are comparable, since all verifiable histories are related by containment and all configurations within one history are comparable.

While reconfigurable objects provide general-purpose reconfiguration interface, dynamic objects are weaker, as they require an external service to build

comparable verifiable histories. In this thesis, it is shown how to build *dynamic* objects in the Byzantine model and how to create *reconfigurable* objects using dynamic objects as building blocks. This technique is applicable to a large class of objects.

1.3.5 Quorum system assumptions

Most fault-tolerant implementations of distributed objects impose some requirements on the subsets of processes that can be faulty. A configuration C is said to be *correct at time* t iff *replicas*(C) is a dissemination quorum system at time t (as defined in Section 1.1). Correctness of the implementation of *dynamic* objects relies on the assumption that active candidate configurations are correct. Once a configuration (i.e., a higher configuration is installed), no assumptions are made about the correctness of the replicas in that configuration.

For *reconfigurable* objects, a slightly more conservative requirement is imposed: every combination of verifiable input configurations that is not yet superseded must be correct. Formally:

Quorum availability: Let $C_1, \ldots, C_k$ be verifiable input configurations such that $C = C_1 \sqcup \ldots \sqcup C_k$ is not superseded at time t. Then *quorums*(C) must be a dissemination quorum system at time t.

Correctness of the *reconfigurable* objects relies solely on correctness of the dynamic building blocks. Formally, when k configurations are concurrently proposed, all possible combinations, i.e., $2^k - 1$ configurations, are required to be correct. However, in practice, at most k of them will be chosen to be put in verifiable histories, and only those configurations will be accessed by correct processes. A more conservative requirement is imposed because it is not known a priori, which configurations will be chosen by the algorithm.

1.3.6 Broadcast primitives

To make sure that no process is "left behind", a variant of *reliable broadcast primitive* [22] is required. The primitive must ensure two properties:

(1) If a forever-correct process p broadcasts a message m, then p eventually delivers m;

(2) If some message m is delivered by a *forever-correct* process, every forever-correct process eventually delivers m.

Note that no assumptions are made about any processes that are not forever-correct. In practice such a primitive can be implemented by a gossip protocol [44]. This primitive is "global" in a sense that it is not bound to any particular configuration. In pseudocode, "**RB-Broadcast** $\langle \ldots \rangle$" is used to denote a call to the "global" reliable broadcast.

Additionally, a "local" *uniform reliable broadcast* primitive [22] is assumed to be available as well. The primitive is "local" because every message sent via this primitive is associated with some configuration. If this configuration is superseded at some point in time, no assumptions are made about the delivery of the message (for example, it is okay if no process ever delivers the message in this case). However, if the configuration is *never superseded*, then a stronger liveness property is required:

(2') if some correct process $p \in$ *replicas*(C) delivered some message m via the instance of the broadcast primitive associated with configuration C, then every forever-correct process $q \in$ *replicas*(C) will eventually deliver m, even if p later turns Byzantine.

This semantic can be easily achieved by making sure that every message is replicated to a quorum of replicas in C before delivering it [22]. In pseudocode, "**URB-Broadcast** $\langle \ldots \rangle$ **in** C" is used to denote a call to the "local" uniform reliable broadcast in configuration C.

1.4 Dynamic Byzantine Lattice Agreement

Dynamic Byzantine Lattice Agreement abstraction (DBLA for short) is the main building block in the construction of reconfigurable objects. Its specification is a combination of the specification of Byzantine Lattice Agreement (Section 1.3.2) and the specification of a dynamic object (Section 1.3.4). The interface of a

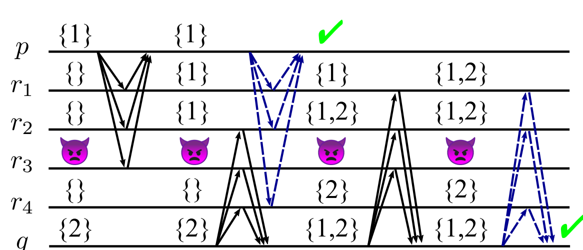
Figure 1.4: The interface of the DBLA object. Parameters are depicted on the left side, operations and functions are on the right side, and callbacks are at the top. The parts of the interface that are inherited from BLA are depicted as solid black arrows, while the parts of the interface that are inherited from the specification of a dynamic object are depicted as dashed blue arrows.

DBLA object is depicted in Figure 1.4. To make the concept easier to grasp, here, a high-level description of the DBLA implementation is given. The complete pseudocode, proofs of correctness, and additional discussions on possible optimizations are delegated to Appendices A.1, A.2, and A.3 respectively.

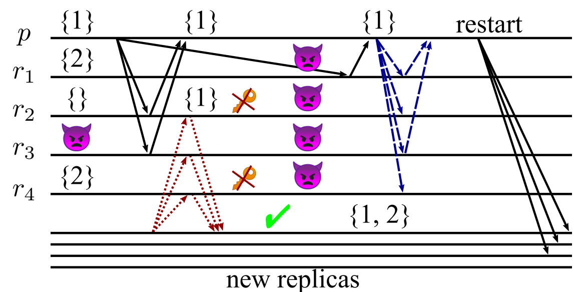
As mentioned earlier, forward-secure digital signatures are used to guarantee that superseded configurations cannot affect correct clients or forge certificates for output values. Ideally, before a new configuration C is installed (i.e., before a correct replica triggers InstalledConfig(C) upcall), one needs to make sure that the replicas of all candidate configurations lower than C invoke UpdateFSKey$(\mathit{height}(C))$. However, this would require the replica to know the set of all candidate configurations lower than C. Unambiguously agreeing on this set would require solving consensus, which is known to be impossible in a fault-prone asynchronous system [35].

Instead, all candidate configurations are classified in two categories: *pivotal* and *tentative*. A candidate configuration is called *pivotal* if it is the *highest* configuration in some verifiable history. Otherwise it is called *tentative*. A nice property of pivotal configurations is that it is impossible to "skip" one in a verifiable history. Indeed, if $C_1 = \text{HighestConf}(h_1)$ and $C_2 = \text{HighestConf}(h_2)$ and $C_1 \sqsubset C_2$, then, since all verifiable histories are related by containment, $h_1 \subseteq h_2$ and $C_1 \in h_2$. This allows the protocol to make sure that, before a configuration C is installed, the replicas in all pivotal (and, possibly, some tentative) configurations lower than C update their keys. As for tentative configurations, the protocol ensures that they are harmless, in a sense that correct clients never interact with them and they cannot create certificates for output values.

In order to reconfigure a DBLA object, a correct client must use reliable



(a) An example execution of two concurrent Propose operations.



(b) An example execution of a Propose operation concurrent with a reconfiguration.

Figure 1.5: Example executions of the DBLA protocol. Solid black arrows (resp., dashed blue arrows) correspond to the messages exchanged during the first (resp., the second) stage of the Propose protocol. Dotted red lines correspond to the messages exchanged during reconfiguration. The numbers represent the sets of verifiable input values known to the processes. Replica r_3 is Byzantine and always responds to PROPOSE messages with the same set of verifiable input values as in the message itself. In **(b)**, replicas r_1, r_2, and r_4 also become Byzantine after the reconfiguration.

broadcast to distribute the new verifiable history. Each correct process p maintains, locally, the largest (with respect to $\subseteq$) verifiable history it delivered so far through reliable broadcast. It is called *the local history of process* p and is denoted by $history_p$. Notation $C_p^{highest}$ is used to denote the most recent configuration in p's local history (i.e., $C_p^{highest} = \text{HighestConf}(history_p)$). Whenever a replica r updates $history_r$, it invokes $\text{UpdateFSKey}(height(C_r^{highest}))$.

Similarly, each process p keeps track of all verifiable input values it has seen $curVals_p \subseteq \mathcal{L} \times \Sigma$, where $\mathcal{L}$ is the object lattice and Σ is the set of all possible certificates. Sometimes, during the execution of the protocol, processes exchange these sets. Whenever a process p receives a message that contains a set of values with certificates $vs \subseteq \mathcal{L} \times \Sigma$, it checks that the certificates are valid (i.e., $\forall\ (v, \sigma) \in vs : \text{VerifyInputValue}(v, \sigma) = true$) and adds these values and certificates to $curVals_p$.

1.4.1 Client implementation

The client's protocol is simple. As was mentioned earlier, the operation $\text{UpdateHistory}(h, \sigma)$ is implemented as **RB-Broadcast** $\langle \text{NEWHISTORY}, h, \sigma \rangle$. The rest of the reconfiguration process is handled by the replicas. The protocol for the operation $\text{Propose}(v, \sigma)$ consists of two stages: *proposing* a value and *confirming* the result.

The first stage (proposing) mostly follows the implementation of lattice agreement by Faleiro et al. [34]. Client p repeatedly sends message $\langle$PROPOSE, $curVals_p$, $seqNum_p$, $C\rangle$ to all replicas in $replicas(C)$, where PROPOSE is the message descriptor, $curVals_p$ is the set of verifiable input values known to the client along with the corresponding certificates, $C = C_p^{highest}$, and $seqNum_p$ is a sequence number used by the client to match sent messages with replies.

After sending these messages to $replicas(C)$, the client waits for responses of the form $\langle$PROPOSERESP, *vs*, *sig*, *sn*$\rangle$, where PROPOSERESP is the message descriptor, *vs* is the set of all verifiable input values known to the replica with valid certificates (including those sent by the client), *sig* is a forward-secure signature with timestamp $height(C)$, and *sn* is the same sequence number as in the message from the client.

During the first stage, three things can happen: (1) the client learns about some new verifiable input values from one of the PROPOSERESP messages; (2) the client updates its local history (by delivering it through reliable broadcast); and (3) the client receives a quorum of valid replies with the same set of verifiable input values. In the latter case, the client proceeds to the second stage. In the first two cases, the client simply restarts the operation. Recall that, according to the BLA-Liveness property, termination of client requests is only guaranteed when the number of verifiable input values is finite. Additionally, the number of verifiable histories is assumed to be finite. Hence, the number of restarts will also be finite. This is the main intuition behind the liveness of the client's protocol.

In the second (confirming) stage of the protocol, the client simply sends the acknowledgments it has collected in the first stage to the replicas of the same configuration. The client then waits for a quorum of replicas to reply with a forward-secure signature with timestamp $height(C)$. The value returned from the Propose operation is simply the join of all verifiable input values in $curVals_p$. This second stage is needed to make sure that during the execution of the first stage the configuration was not superseded.

The example in Figure 1.5a illustrates how the first stage of the algorithm ensures the comparability of the results when no reconfiguration is involved. In this example, clients p and q concurrently propose values $\{1\}$ and $\{2\}$, re-

spectively, from the lattice $(\mathcal{L}, \sqsubseteq) = (2^{\mathbb{N}}, \subseteq)$. Client p successfully returns the proposed value $\{1\}$ while client q is forced to refine its proposal and return the combined value $\{1, 2\}$. The quorum intersection prevents the clients from returning incomparable values (e.g., $\{1\}$ and $\{2\}$).

The example in Figure 1.5b illustrates how reconfiguration can interfere with an ongoing Propose operation in what is called the **"slow reader" attack**, and how the second stage of the protocol prevents a safety violation. Imagine that some correct client completed the Propose operation and received value $\{2\}$ before client p started the execution. This is reflected on the picture by the fact that all correct replicas in quorum $\{r_1, r_3, r_4\}$ store value $\{2\}$. Then client p executes Propose$(\{1\}, \sigma)$, where σ is a valid certificate for input value $\{1\}$. Due to the BLA-Comparability, BLA-Inclusion, and BLA-Validity properties of Byzantine Lattice Agreement, the only valid output value for client p is $\{1, 2\}$ (assuming that there are no other verifiable input values). The client successfully reaches replicas r_2 and r_3 before the reconfiguration. None of them tell the client about the input value $\{2\}$, because r_2 is outdated and r_3 is Byzantine. The message from p to r_1 is delayed. Meanwhile, a new configuration is installed, and all replicas of the original configuration become Byzantine. When the message from p finally reaches r_1, the replica is already Byzantine and it can pretend that it has not seen any verifiable input values other than $\{1\}$. The client then finishes the first stage of the protocol with value $\{1\}$. Returning this value from from the Propose operation would violate BLA-Comparability.

Luckily, the second stage of the protocol prevents the safety violation. Since replicas r_2 and r_4 updated their private keys during the reconfiguration, they are unable to send the signed confirmations with timestamp *height*(C) to the client. Hence, the client will not be able to complete the operation in configuration C and will wait until it receives a new verifiable history via reliable broadcast and will restart the operation in a higher configuration.

The certificate for the output value $v \in \mathcal{L}$ produced by the Propose protocol in a configuration C consists of:

1. the set of verifiable input values (with certificates for them) from the first stage of the algorithm (the join of all these values must be equal to v);

2. a verifiable history (with a certificate for it) that confirms that C is a pivotal configuration;

3. the quorum of signatures from the first stage of the algorithm;

4. the quorum of signatures from the second stage of the algorithm.

Intuitively, the only way for a Byzantine client to obtain such a certificate is to benignly follow the Propose protocol.

1.4.2 Replica implementation

Each replica r maintains, locally, its *current configuration* (denoted by C_r^{curr}) and *the last configuration installed by this replica* (denoted by C_r^{inst}). The following invariant is maintained: $C_r^{inst} \sqsubseteq C_r^{curr} \sqsubseteq C_r^{highest}$. Intuitively, $C_r^{curr} = C$ means that replica r knows that there is no need to transfer state from configurations lower than C, either because r already performed the state transfer from those configurations, or because it knows that sufficiently many other replicas did. $C_r^{inst} = C$ means that the replica knows that sufficiently many replicas in C have up-to-date states, and that configuration C is ready to serve user requests.

Each client message is associated with some configuration C. The replica only answers the message when $C = C_r^{inst} = C_r^{curr} = C_r^{highest}$. If $C \sqsubset C_r^{highest}$, the replica simply ignores the message. Due to the properties of reliable broadcast, the client will eventually learn about $C_r^{highest}$ and will repeat its request there (or in an even higher configuration). If $C_r^{inst} \sqsubset C$ and $C_r^{highest} \sqsubseteq C$, the replica waits until C is installed before processing the message. Finally, if C is incomparable with C_r^{inst} or $C_r^{highest}$, then, since all candidate configurations are required to be comparable, the message is sent by a Byzantine process and the replica should ignore it.

When a correct replica r receives a PROPOSE message, it adds the newly learned verifiable input values to $curVals_r$ and sends $curVals_r$ back to the client with a forward-secure signature with timestamp $height(C)$. When a correct replica receives a CONFIRM message, it simply signs the set of acknowledgments in it with a forward-secure signature with timestamp $height(C)$ and sends the signature to the client.

Algorithm 1 DBLA state transfer, code for replica r

```
1:  upon C^curr ≠ HighestConf({C ∈ history | r ∈ replicas(C)})
2:      let C^next = HighestConf({C ∈ history | r ∈ replicas(C)})
3:      let S = {C ∈ history | C^curr ⊑ C ⊏ C^next}
4:      seqNum ← seqNum + 1
5:      for each C ∈ S do
6:          send ⟨UpdateRead, seqNum, C⟩ to replicas(C)
7:          wait for (C ⊏ C^curr) ∨ (responses from any Q ∈ quorums(C) with s.n. seqNum)
8:      if C^curr ⊏ C^next then
9:          C^curr ← C^next
10:         URB-Broadcast ⟨UpdateComplete⟩ in C^next

11: upon receive ⟨UpdateRead, sn, C⟩ from replica r′
12:     wait for C ⊏ HighestConf(history)
13:     send ⟨UpdateReadResp, curVals, sn⟩ to r′

14: upon receive ⟨UpdateReadResp, vs, sn⟩ from replica r′
15:     if VerifyInputValues(vs \ curVals) then curVals ← curVals ∪ vs

16: upon URB-deliver ⟨UpdateComplete⟩ in C from quorum Q ∈ quorums(C)
17:     wait for C ∈ history
18:     if C^inst ⊏ C then
19:         if C^curr ⊏ C then C^curr ← C
20:         C^inst ← C
21:         trigger upcall InstalledConfig(C)
22:         if r ∉ replicas(C) then halt
```

A very important part of the replica's implementation is *the state transfer protocol*. The pseudocode for it is presented in Algorithm 1. Note that the subscript r is omitted in the pseudocode because each process can only access its own variables directly. Let C_r^{next} be the highest configuration in $history_r$ such that $r \in replicas(C_r^{next})$. Whenever $C_r^{curr} \neq C_r^{next}$, the replica tries to "move" to C_r^{next} by reading the current state from all configurations between C_r^{curr} and C_r^{next} one by one in ascending order (line 5). In order to read the current state from configuration $C \sqsubset C_r^{next}$, replica r sends message ⟨UpdateRead, $seqNum_r$, C⟩ to all replicas in $replicas(C)$. In response, each replica $r_1 \in replicas(C)$ sends $curVals_{r_1}$ to r in an UpdateReadResp message (line 13). However, r_1 replies only after its private key is updated to a timestamp larger than $height(C)$ (line 12). For any correct replica q, the following invariant is maintained: $st_q = height(\text{HighestConf}(history_q))$, where st_q is the timestamp of the private key of q.

If r receives a quorum of replies from the replicas of C, there are two distinct cases:

- C is still active. In this case, the quorum intersection property still holds for C, and replica r can be sure that (1) if some Propose operation has either completed in configuration C or reached the second stage, $v \sqsubseteq \text{JoinAll}(\mathit{curVals}_r)$, where v is the value returned by the Propose operation and $\text{JoinAll}(\mathit{curVals}_r)$ is the join of all verifiable input values in the set $\mathit{curVals}_r$; and (2) if some Propose operation has not yet reached the second stage, it will not be able to complete in configuration C (see the example in Figure 1.5b).
- C is already superseded. In this case, by definition, a higher configuration is installed, and, intuitively, replica r will get the necessary state from that higher configuration.

It may happen that configuration C is already superseded and r will not receive sufficiently many replies from the replicas of C. However, in this case r will eventually discover that some higher configuration is installed, and it will update C_r^{curr} (line 19).

When a correct replica completes transferring the state to some configuration C, it notifies other replicas about it by broadcasting message UPDATECOMPLETE in configuration C (line 10). A correct replica *installs* a configuration C if it receives such messages from a quorum of replicas in C (line 16). Because the protocol must satisfy the Installation Liveness property (if one correct replica installs a configuration, every forever-correct replica must eventually install this or a higher configuration), the UPDATECOMPLETE messages are distributed through the local uniform reliable broadcast primitive that was introduced in Section 1.3.6.

1.4.3 Implementing other dynamic objects

In order to adopt other asynchronous Byzantine fault-tolerant static algorithms to the dynamic model, the same set of techniques can be applied, including:

1. the state transfer protocol (relying on forward-secure signatures);

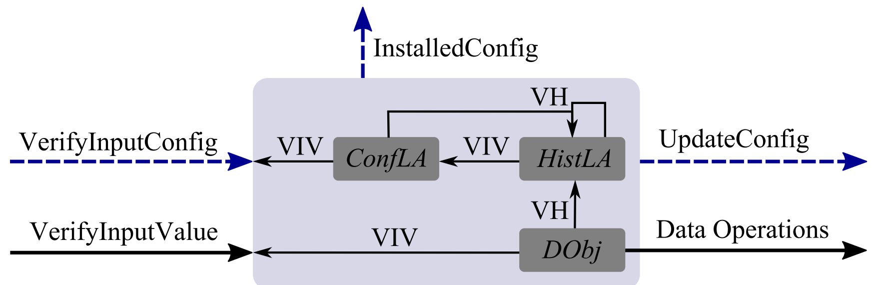


Figure 1.6: The structure of dependencies in the implementation of a reconfigurable object. An arrow from an object A to an object B marked with VIV (resp., VH) indicates that A.VerifyInputValue (resp., A.VerifyHistory) is implemented using B.VerifyOutputValue.

2. the use of an additional round-trip to prevent the "slow reader" attack;

3. the structure of cryptographic proofs ensuring that tentative configurations cannot create valid certificates for output values.

To illustrate this, in Appendix A.4, the dynamic version of Max-Register [13] is presented. Additionally, the dynamic version of the Access Control abstraction is discussed in Section 1.6.

1.5 Reconfigurable objects

While dynamic objects are important building blocks, they are not very useful by themselves as they require an external source of comparable verifiable histories. In this section, it is shown how to combine several dynamic objects to obtain a single *reconfigurable* object. Similarly to dynamic objects, the specification of a reconfigurable object can be obtained as a combination of the specification of a static object with the specification of an abstract reconfigurable object from Section 1.3.3. In particular, compared to static objects, reconfigurable objects have one more operation – UpdateConfig(C, σ), must be parameterized by a boolean function VerifyInputConfig(C, σ), and must satisfy Reconfiguration Validity, Reconfiguration Liveness, and Installation Liveness.

1.5.1 Implementation

A reconfigurable object is created by combining three *dynamic* ones. The first one is the dynamic object that executes clients' operations (let us call it *DObj*).

For example, in order to implement a reconfigurable version of Byzantine Lattice Agreement, one needs to take a dynamic version of Byzantine Lattice Agreement as *DObj*. Similarly, in order to implement a reconfigurable version of Max-Register [13], one needs to take a dynamic version of Max-Register as *DObj* (see Appendix A.4). The two remaining objects are used to build verifiable histories: *ConfLA* is a DBLA operating on the configuration lattice $\mathcal{C}$, and *HistLA* is a DBLA operating on the powerset lattice $2^{\mathcal{C}}$. The relationships between the three dynamic objects are depicted in Figure 1.6.

Algorithm 2 Reconfigurable object

```
    ▷ Common code
    Parameters:
23:     Lattice of configurations 𝒞 and the initial configuration C^init
24:     Boolean function VerifyInputConfig(C, σ)
25:     Dynamic object DObj, which we want to make reconfigurable
    Shared objects:
26:     DObj                               ▷ the dynamic object being transformed
27:     ConfLA                             ▷ DBLA on lattice 𝒞
28:     HistLA                             ▷ DBLA on lattice 2^𝒞

    ▷ Code for client p
29: Data operations are performed directly on DObj.
30: operation UpdateConfig(C, σ)
31:     let (D, σ_D) = ConfLA.Propose(C, σ)
32:     let (h, σ_h) = HistLA.Propose({D}, σ_D)
33:     DObj.UpdateHistory(h, σ_h)
34:     ConfLA.UpdateHistory(h, σ_h)
35:     HistLA.UpdateHistory(h, σ_h)

    ▷ Code for replica r
36: upon receive upcall InstalledConfig(C) from all DObj, ConfLA, and HistLA
37:     trigger upcall InstalledConfig(C)

    ▷ Parameters specification
38: function ConfLA.VerifyInputValue(v, σ) = VerifyInputConfig(v, σ)
39: function HistLA.VerifyInputValue(v, σ)
40:     if v is not a set of 1 element then return false
41:     let {C} = v
42:     return ConfLA.VerifyOutputValue(C, σ)
43: function VerifyHistory(h, σ) = HistLA.VerifyOutputValue(h, σ)    ▷ used by all dynamic
    objects
```

The pseudocode is presented in Algorithm 2. All data operations are performed directly on *DObj*. To update a configuration, the client first submits

its proposal to *ConfLA* and then submits the result as a singleton set to *HistLA*. Due to the BLA-Comparability property, all verifiable output values produced by *ConfLA* are comparable, and any combination of them would create a well-formed history as defined in Section 1.3.4. Moreover, the verifiable output values of *HistLA* are related by containment, and, therefore, can be used as verifiable histories in dynamic objects. They are used to reconfigure all three dynamic objects (lines 33–35).

Cryptographic keys. In Algorithm 2, several dynamic objects are used. Correct replicas are assumed to have separate public/private key pairs for each dynamic object. This prevents replay attacks across objects and allows each dynamic object to manage its keys separately. Later in this section, it is discussed how to avoid this assumption.

1.5.2 Proof of correctness

In the following two lemmas it is shown that the dynamic objects (*ConfLA*, *HistLA*, and *DObj*) are used correctly, i.e., all requirements imposed on verifiable histories are satisfied.

Lemma 1.1. *All histories passed to the dynamic objects by correct processes (Algorithm 2, lines 33–35) are verifiable with* VerifyHistory *(Algorithm 2, line 43).*

Proof. Follows from the BLA-Verifiability property of *HistLA*. □

Lemma 1.2. *All histories verifiable with* VerifyHistory *(Algorithm 2, line 43) are (1) well-formed (that is, consist of comparable configurations) and (2) related by containment. Moreover, (3) in any given infinite execution, there is only a finite number of histories verifiable with* VerifyHistory.

Proof. (1) follows from the BLA-Comparability property of *ConfLA*, the BLA-Validity property of *HistLA*, and the definition of *HistLA*.VerifyInputValue (Algorithm 2, line 39).

(2) follows directly from the BLA-Comparability property of *HistLA*.

(3) follows from the requirement of finite number of verifiable input configurations and the BLA-Validity property of *ConfLA* and *HistLA*. Only a finite number of configurations can be formed by *ConfLA* out of a finite number of verifiable input configurations, and only a finite number of histories can be formed by *HistLA* out of the configurations produced by *ConfLA*. □

Theorem 1.3 (Transformation safety)**.** *Our implementation satisfies the Reconfiguration Validity property of a reconfigurable object. That is, (1) every installed configuration C is a join of some set of verifiable input configurations; and (2) all installed configurations are comparable.*

Proof. (1) follows from the BLA-Validity property of *ConfLA* and *HistLA* and the Dynamic Validity property of the underlying dynamic objects. (2) follows directly from the Dynamic Validity property of the underlying dynamic objects. □

Theorem 1.4 (Transformation liveness)**.** *Our implementation satisfies the liveness properties of a reconfigurable object: Reconfiguration Liveness and Installation Liveness.*

Proof. Reconfiguration Liveness follows from the BLA-Liveness property of *ConfLA* and *HistLA* and the Dynamic Liveness property of the underlying dynamic objects. Installation Liveness follows from line 36 of the implementation and the Installation Liveness of the underlying dynamic objects. □

1.5.3 Discussion

Performance. Due to the BLA-Validity property of *ConfLA* and *HistLA*, when k reconfiguration requests are executed concurrently, at most k new verifiable histories will be created and the total number of candidate configurations will not exceed $k + 1$ (including the initial configuration). Hence, only $O(k)$ configurations are accessed for state transfer, and not an exponential number as in some earlier work on reconfiguration [8, 37]. This is known to be optimal [70].

Bootstrapping. The relationship between lattice agreement and reconfiguration has been studied before [42, 49]. In particular, as shown in [42], lattice

agreement can be used to build comparable configurations. In this thesis, a step further is taken in this direction. Two separate instances of lattice agreement are used: one to build comparable configurations (*ConfLA*) and one to build histories out of them (*HistLA*). These two LA objects can then be used to reconfigure a single dynamic object (*DObj*).

However, this raises a question: how to reconfigure the lattice agreement objects themselves? The intuition behind the proposed solution is the idea that is sometimes referred to as "bootstrapping". The lattice agreement objects are used to reconfigure themselves and at least one other object (*DObj*). This implies that the lattice agreement objects share the configurations with *DObj*. The most natural implementation is that the code for all three dynamic objects (*ConfLA*, *HistLA*, and *DObj*) will be executed by the same set of replicas.

Bootstrapping is a dangerous technique because, if applied badly, it can lead to infinite recursion. However, the solution is structured in such a way that there is no recursion at all: the client first makes normal requests to *ConfLA* and *HistLA* and then uses the resulting history to reconfigure all dynamic objects, as if this history was obtained by the client from the outside of the system. It is important to note that the liveness of the call *HistLA*.VerifyOutputValue(h, σ) is not affected by reconfiguration: the function simply checks some digital signatures and is guaranteed to always terminate given enough processing time.

Shared parts. All implementations of dynamic objects presented in this thesis have a similar structure. For example, they all share the same state transfer implementation (see Algorithm 1). However, implementations of other dynamic objects might have very different implementations. Therefore, in the transformation, *DObj* is used as a "black box" and no assumptions are made about its implementation. Moreover, for simplicity, the two DBLA objects are used as "black boxes" as well. In fact, *ConfLA* and *HistLA* may have different implementations and the transformation will still work as long as they satisfy the specification from Section 1.3. However, this generalization comes at a cost.

In particular, if implemented naively, a single reconfigurable object will run several independent state transfer protocols, and a single correct replica will have several private/public key pairs (as mentioned earlier in this section). How-

ever, if all dynamic objects have similar implementations of their state transfer protocols (as in this thesis), this can be done more efficiently by combining all state transfer protocols into one, which would need to transfer the states of all dynamic objects and make sure that the superseded configurations are harmless.

1.6 Access control

The implementation of reconfigurable objects relies on the parameter function VerifyInputConfig. Moreover, if the transformation from Section 1.5 is applied to the implementation of DBLA from Section 1.4, the resulting reconfigurable object will rely on the parameter function VerifyInputValue. The implementation of these parameters is highly application-specific. For example, in a storage system, it is reasonable to only allow requests that modify some data if they are accompanied by a digital signature produced by the owner of the data. For the sake of completeness, in this section, three generic implementations for these parameter functions are presented. Each of these implementations is suitable for some applications.

In order to do this, the *Access Control* object is introduced. It exports one operation and one function:

- Operation RequestCert(v) returns a *certificate* σ, which can be verified with VerifyCert(v, σ), or the special value $\perp$, indicating that the permission was denied;
- Function VerifyCert(v, σ) returns a boolean value.

The implementation of *Access Control* must satisfy the following property:

- *Certificate Verifiability*: If RequestCert(v) returned σ to a correct process, then VerifyCert$(v, \sigma) = $ *true*.

In Sections 1.6.1–1.6.3, three different implementations of the Dynamic Access Control object are presented. In Section 1.6.4, two methods to use the Dynamic Access Control abstraction in order to implement the parameter functions VerifyInputValue and VerifyInputConfig are demonstrated.

1.6.1 Sanity-check approach

Algorithm 3 Vote-Based Dynamic Access Control

```
    ▷ Code for client p
44: operation RequestCert(v)
45:     let C = HighestConf(history)
46:     seqNum ← seqNum + 1                    ▷ used to match requests with responses
    ▷ Phase one: request
47:     send ⟨REQUEST, v, seqNum, C⟩ to replicas(C)
48:     wait for (HighestConf(history) ≠ C) ∨ (received enough Yes-votes with valid signa-
    tures)
49:             ∨ (received a quorum of votes in total)
50:     if HighestConf(history) ≠ C then retry (goto line 45)
51:     if received not enough valid Yes-votes then return ⊥          ▷ access denied
52:     let acks_1 = {Yes-votes received on line 49}
    ▷ Phase two: confirm
53:     send ⟨CONFIRM, acks_1, seqNum, C⟩ to replicas(C)
54:     wait for (HighestConf(history) ≠ C) ∨ (replies from a quorum of replicas with valid
    signatures)
55:     if HighestConf(history) ≠ C then retry (goto line 45)
56:     let acks_2 = {acknowledgments received on line 54}
    ▷ Return certificate
57:     return (history, σ_history, acks_1, acks_2)

    ▷ Code for replica r
58: upon receive ⟨REQUEST, v, sn, C⟩ from client c
59:     wait for C = C^inst ∨ C ⊏ HighestConf(history)
60:     if C = HighestConf(history) then
61:         if VoteYes(v) then send ⟨YES, FSSign((YES, v, c), height(C)), sn⟩ to c
62:         else send ⟨NO, sn⟩ to c

63: upon receive ⟨CONFIRM, acks, sn, C⟩ from client c
64:     wait for C ∈ history
65:     if C = HighestConf(history) then
66:         let sig = FSSign((CONFIRMRESP, acks), height(C))
67:         send ⟨CONFIRMRESP, sig, sn⟩ to c
```

One of the simplest implementations of access control in a *static* system is to require at least $b + 1$ replicas to sign each certificate, where b is the maximal possible number of Byzantine replicas, sometimes called the *resilience threshold*. The correct replicas can perform some application-specific sanity checks before approving requests.

The key property of this approach is that it guarantees that each valid certifi-

cate is signed by *at least one correct replica*. In many cases, this is sufficient to guarantee resilience both against the Sybil attacks [30] and against attempts to flood the system with reconfiguration requests. The correct replicas can check the identities of the new participants and refuse to sign excessively frequent requests.

In *dynamic* asynchronous systems, just providing $b+1$ signatures is not sufficient. Despite the use of forward-secure signatures, in a superseded pivotal configuration there might be significantly more than b Byzantine replicas with their private keys not removed (in fact, at least $2b$). The straightforward way to implement this policy in a *dynamic* system is to add the *confirming* phase, as in the implementation of Dynamic Byzantine Lattice Agreement (see Section 1.4), after collecting $b+1$ signed approvals. The confirming phase guarantees that, during the execution of the first phase, the configuration was active. The state transfer protocol should be the same as for DBLA with the exception that no actual state is being transferred. The only goal of the state protocol in this case is it to make sure that the replicas update their private keys before a new configuration is installed.

This and the following approach can be generally described as "vote-based" access control policies. The pseudocode for their dynamic implementation is presented in Algorithm 3.

1.6.2 Quorum-based approach ("on-chain governance")

A more powerful strategy in a *static* system is to require a *quorum* of replicas to sign each certificate. An important property of this implementation is that it can detect and prevent conflicting requests. More formally, suppose that there are values v_1 and v_2, for which the following two properties should hold:

- Both are acceptable: RequestCert(v_i) should not return $\perp$ unless RequestCert(v_j) was invoked in the same execution, where $j \neq i$.
- At most one may be accepted: if some process knows σ_i such that VerifyCert(v_i, σ_i) then no process should know σ_j such that VerifyCert(v_j, σ_j).

Note that it is possible that neither v_1 nor v_2 is accepted by the Access Control if the requests are made concurrently. To guarantee that exactly one certificate is issued, one would need to implement consensus, which is impossible in asynchronous model [35]. If a correct replica has signed a certificate for value v_i, it should store this fact in persistent memory and refuse signing v_j if requested. Due to the quorum intersection property, this guarantees the "at most one" semantic in a static system.

This approach can be implemented in a *dynamic* system using the pseudocode from Algorithm 3 and the state transfer protocol from the DBLA implementation (see Algorithm 1).

Using the dynamic version of this approach to certifying reconfiguration requests allows capturing the notion of what is sometimes called "on-chain governance". The idea is that the participants of the system (in this case, the owners of the replicas) decide which actions or updates to allow by the means of voting. Every decision needs a quorum of signed votes to be considered valid and no two conflicting decisions can be made.

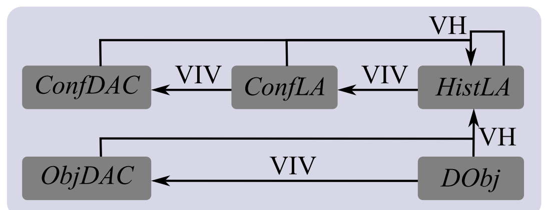
1.6.3 Trusted administrators

A naive yet common approach to dynamic systems is to have a trusted administrator, who signs the reconfiguration requests. However, if the administrator's private key is lost, the system might lose liveness, and if it is compromised, the system might lose even safety. A more viable approach is to have n administrators and to require $b + 1$ of them to sign every certificate, for some n and b such that $0 \leq b < n$. In this case, the system will "survive" up to b keys being compromised and up to $n - (b + 1)$ keys being lost. An interesting problem that is not considered in this thesis is an algorithm to change the set of administrators.

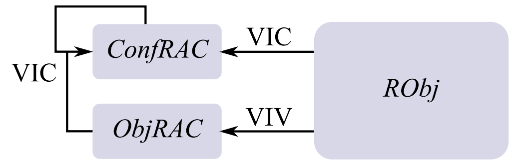
1.6.4 Combining Access Control with other objects

There are at least two possible ways to combine the Access Control abstraction with a reconfigurable object in a practical system.

The simplest, and, perhaps, the most practical approach is to embed two instances of Dynamic Access Control directly into the structure of a reconfig-



(a) Dynamic Access Control inside a reconfigurable object.



(b) Reconfigurable Access Control in combination with another reconfigurable object.

Figure 1.7: Two possible ways to integrate the Access Control abstraction with other types of objects. An arrow from an object A to another object B marked with VIV, (resp., VIC or VH) indicates that A.VerifyInputValue (resp., A.VerifyInputConfig or A.VerifyHistory) is implemented using B.VerifyOutputValue or B.VerifyCert.

urable object, as shown in Figure 1.7a. In this case, the replicas that execute the code for the Access Control are the same replicas as the replicas that execute the code of other dynamic objects in the implementation of this reconfigurable object.

Alternatively, one can apply the transformation from Section 1.5 to the dynamic Access Control implementation described in this section to obtain a *reconfigurable* version of the Access Control abstraction. It then can be combined with any other reconfigurable object in a structure depicted in Figure 1.7b. In this case, the replicas of *ConfRAC* produce verifiable input configurations for themselves and for two other objects.

1.7 Related work

Dynamic replicated systems with *passive reconfiguration* [17, 15, 46] do not explicitly regulate arrivals and departures of replicas. Their consistency properties are ensured under strong assumptions on the churn rate. Except for the recent work [46], churn-tolerant storage systems do not tolerate Byzantine failures. In contrast, *active reconfiguration* allows the clients to explicitly propose configuration updates, e.g., sets of new replica arrivals and departures.

Early proposals of (actively) reconfigurable storage systems tolerating process crashes, such as RAMBO [38] and reconfigurable Paxos [57], used consensus (and, thus, assumed certain level of synchrony) to ensure that the clients agree on the evolution of configurations. DynaStore [8] was the first *asynchronous* reconfigurable storage: clients propose incremental additions or re-

movals to the system configuration. As the proposals commute, the processes can resolve their disagreements without involving consensus.

The *parsimonious speculative snapshot* task [37] allows to resolve conflicts between concurrent configuration updates in a storage system using instances of commit-adopt [36]. The worst-case time complexity, in the number of message delays, of reconfiguration was later reduced from $O(n^2)$ to $O(n)$ [70], where n is the number of concurrently proposed configuration updates.

SmartMerge [42] made an important step forward by treating reconfiguration as an instance of abstract *lattice agreement* [34]. However, the algorithm assumes an external (reliable) lattice agreement service which makes the system not fully reconfigurable. The recently proposed *reconfigurable lattice-agreement* abstraction [49] enables truly reconfigurable versions of a large class of objects and constructions, including state-based CRDTs [66], atomic-snapshot, max-register, conflict detector and commit-adopt. The reconfiguration service described in this thesis can be used to derive Byzantine fault-tolerant reconfigurable implementations of objects in the class.

Byzantine quorum systems [59] introduce abstractions for ensuring availability and consistency of shared data in asynchronous systems with Byzantine faults. In particular, a *dissemination* quorum system ensures that every two quorums have a correct process in common and that at least one quorum only contains correct processes.

Dynamic Byzantine quorum systems [12] appear to be the first attempt to implement a form of active reconfiguration in a Byzantine fault-tolerant data service running on a *static* set of replicas, where clients can raise or lower the resilience threshold. Dynamic Byzantine storage [61] allows a trusted *administrator* to issue ordered reconfiguration calls that might also change the set of replicas. The administrator is also responsible for generating new private keys for the replicas in each new configuration to anticipate the "I still work here" attack [7]. In this thesis, an implementation of a Byzantine fault-tolerant reconfiguration service that does not rely on this assumption is proposed.

Forward-secure signature schemes [18, 19, 23, 31, 60] were originally designed to mitigate the consequences of key exposure: if the private key of an agent is compromised, signatures made prior to the exposure (i.e., with smaller

timestamps) can still be trusted. In this thesis, a novel application of forward-secure digital signatures is proposed: timestamps are associated with configurations. Before a new configuration is installed, the protocol ensures that sufficiently many correct processes update their private keys in prior configurations. This approach prevents the "I still work here" and "slow reader" attacks. Unlike previously proposed solutions [61], it does not rely on a global agreement on the configuration sequence or a trusted administrator.

1.8 Discussions

Communication cost. In this thesis, no intention was made to provide the optimal implementations of each object or to implement the most general abstractions (such as generalized lattice agreement [34, 49]). Instead, the focus was made on providing the minimal implementation for the minimal set of abstractions to demonstrate the ideas and the general techniques for defining and building reconfigurable services in the harsh world of asynchrony and Byzantine failures. Therefore, the implementations proposed in this thesis leave plenty of space for optimizations. Some directions for optimizations are discussed in Section 1.5.3 and Appendix A.3.

Further research. In this thesis, a very strong model of the adversary is assumed: no assumptions are made about correctness of replicas in superseded configurations. This pessimistic approach leads to more complicated and expensive protocols. Moreover, an attempt was made to generalize the discussion instead of focusing on creating a reconfigurable version of some particular object. In [39], we considered a simpler yet, arguably, more realistic model, where the adversary is not able to corrupt more than one third of participants even in superseded configurations. We also simplified the task by focusing on Byzantine reliable broadcast, which is a simpler problem compared to lattice agreement or register emulation. As expected, the resulting algorithm is simpler and much more efficient. As shown in [28], Byzantine reliable broadcast can be used as a foundation for an efficient implementation of a cryptocurrency.

While the discussion in this thesis is kept as general as possible, potential

practical applications of asynchronous Byzantine fault-tolerant reconfiguration is an interesting avenue for future research. As discussed in [28] and [39], a natural application is the implementation of permissioned cryptocurrencies. In [48], we applied and extended the techniques presented in this thesis to implement a *permissionless* proof-of-stake cryptocurrency, where anybody can participate in the system simply by obtaining and holding any amount of the corresponding asset.

Another application of Byzantine fault-tolerant reconfiguration is explored by De Souza et al. in [29]. They applied forward-secure digital signatures in a way similar to what is proposed in this thesis in order to implement *accountable* reconfigurable Byzantine fault-tolerant objects, which can detect Byzantine processes in case of a safety violation and automatically repair the system by excluding them. They considered a weaker model where clients are not subject to Byzantine failures, which leads to a significantly simpler algorithm.

Open problems. One of the main hurdles that prevent widespread adoption of asynchronous reconfiguration is that there is still no efficient "blackbox transformation" of static algorithms into reconfigurable ones, even for crash fault-tolerant systems. Although there are some repetitive patterns in the design of reconfigurable systems, each algorithm still has to be adapted to the dynamic setting individually. As a step in the direction of solving this issue, in this thesis, a generic framework for defining reconfigurable objects is proposed.

There is also no simple framework for proving correctness of dynamic and reconfigurable algorithms. For some objects, the correctness of a static algorithm might be proven within few sentences, while the rigorous proof of correctness of its dynamic counterpart might take several pages. Most notably, the simple properties of Byzantine quorum systems [59] have no trivial counterparts in dynamic systems.

Another issue that has not been addressed so far is scalability. In particular, it would be interesting to devise algorithms that would efficiently adapt to "small" configuration changes, while still supporting the option of completely changing the set of replicas in a single reconfiguration request. In the model considered in most papers on asynchronous reconfiguration, each reconfiguration is treated as

if it completely changes the set of replicas, which leads to an expensive quorum-to-quorum communication pattern. This seems unnecessary for reconfiguration requests involving only slight changes to the set of replicas.

2 Efficient Byzantine Fault-Tolerant Consensus

In this chapter, a new fast Byzantine consensus algorithm is proposed, accompanied by a lower bound that stipulates that the proposed algorithm has optimal resilience. The main results presented in this chapter are accepted for publication in the proceedings of PODC 2021 [1]. The latest review of the paper is available on arXiv [51].

The chapter is organized as follows. First, in Section 2.1, the model assumptions and the consensus problem are formalized. In Section 2.2, the proposed fast Byzantine consensus protocol is described and is proven correct. Section 2.3 contains the new lower bound and the discussion on the applicability of the lower bound from [62]. Finally, related work is discussed in Section 2.4.

2.1 Preliminaries

2.1.1 Model assumptions

A set $\Pi = \{p_1, \ldots, p_n\}$ of n *processes* is considered. Every process is assigned with an *algorithm* (deterministic state machine) that it is expected to follow. A process that *deviates* from its algorithm, by performing a step that is not prescribed by the algorithm or prematurely stopping taking steps, is called *Byzantine*.

It is assumed that the number of Byzantine processes in an execution of the algorithm does not exceed parameter f. Sometimes, a subset of executions in which up to $t \leq f$ processes are Byzantine is considered. Non-Byzantine processes are called *correct*.

The processes communicate by sending messages across *reliable* (no loss, no duplication, no creation) point-to-point communication channels. More precisely, if a correct process sends a message to a correct process, the message is eventually received. The adversary may not create messages or modify messages in transit. The channels are authenticated: the sender of each received message can be unambiguously identified.

Every process is assigned with a public/private key pair. Every process knows the identifiers and public keys of every other processes. The adversary

is computationally bounded to be unable to break to compute private keys of correct processes.

A *partially synchronous* system is assumed: there exists a known *a priori* bound on message delays Δ that holds *eventually*: there exists a moment in time after which every message sent by a correct process to a correct process is received within Δ time units. This (unknown to the processes) time when the bound starts to hold is called *global stabilization time* (GST). For brevity, the time spent on local computations is neglected.

2.1.2 The consensus problem

Each process $p \in \Pi$ is assigned with an input value x_p^{in}. At most once in any execution, a correct process can *decide* on a value x by triggering the callback $\textsc{Decide}(x)$.

Any infinite execution of a consensus protocol must satisfy the following conditions:

Liveness: Each correct process must eventually decide on some value;

Consistency: No two correct processes can decide on different values;

Validity: Two flavors of this property are considered:

- **Weak validity:** If all processes are correct *and propose the same value*, then only this value can be decided on;
- **Extended validity:** If all processes are correct, then only a value proposed by some process can be decided on.

Note that extended validity implies weak validity, but not vice versa (i.e., extended validity is *strictly stronger* than weak validity). Our algorithm solves consensus with extended validity, while our matching lower bound holds even for consensus with weak validity.

2.2 Algorithm

In this section, the new fast Byzantine consensus algorithm for the system of $n = 5f - 1$ processes is presented and is proven correct. Also, its generalization for $n = 3f + 2t - 1$ processes is discussed.

The algorithm proceeds in numbered views. Each process maintains its current *view number*, and each view is associated with a single *leader* process by an agreed upon mapping *leader* : $\mathbb{Z}_{>0} \to \Pi$. For simplicity, let us assume that *leader*$(v) = p_{(v \bmod n)+1}$. When all correct processes have the same current view number v, process *leader*(v) is said to be *elected*.

The processes execute a *view synchronization protocol* in the background. No explicit implementation for it is provided in this thesis since any implementation from the literature is sufficient [24, 20, 63].

The view synchronization protocol must satisfy the following three properties:

- The view number of a correct process is never decreased;
- In any infinite execution, a correct leader is elected an infinite number of times. In other words, for any point in the execution, there is a moment in the future when a correct leader is elected;
- If a correct leader is elected after GST, no correct process will change its view number for the time period of at least 5Δ.

Initially, the view number of each process is 1. Hence, process *leader*(1) is elected at the beginning of the execution. If *leader*(1) is correct and the network is synchronous from the beginning of the execution (GST $= 0$), our protocol guarantees that all correct processes decide some value before any process changes its view number.

The first leader begins with sending a PROPOSE message with its current decision estimate to all processes. If a process accepts the proposal, it sends an ACK message to every other process. A process *decides* on the proposed value once it receives ACK messages from a quorum ($n - f$) of processes. Therefore, as long as the leader is correct and the correct processes do not change their views prematurely, every correct process decides after just two communication steps.

When correct processes change their views, they engage in the *view change protocol*, helping the newly elected leader to obtain a *safe* value to propose equipped with a *certificate* that confirms that the value is safe. (A value is safe in a view if no other value was or will ever be decided in a smaller view).

The view change protocol consists of two phases: first, the leader collects *votes* from processes and makes a decision about which value is safe, and, second, the leader asks $2f + 1$ other processes to confirm with a digital signature that they agree with the leader's decision. This second phase, not typical for other consensus protocols, is used to ensure that the certificate sizes do not grow indefinitely in case of a long period of asynchrony.

Once the view change protocol is completed, the new leader runs the normal case protocol: it sends a PROPOSE message to every process and waits for $n - f$ acknowledgments.

Below, the normal case execution and the view change protocol are described in more detail.

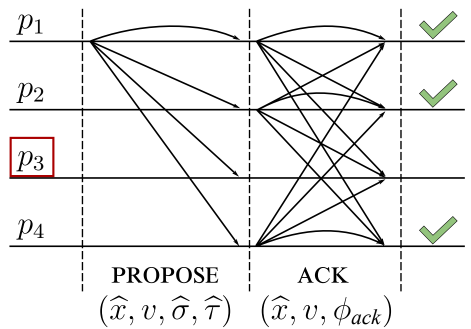
2.2.1 Normal case

A value x is said to be *safe in a view* v if no value other that x can be decided in a view $v' < v$.

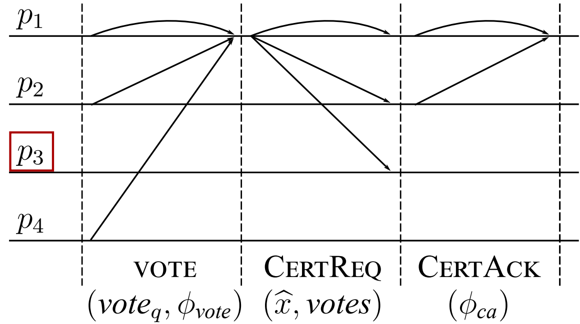
A view change protocol (described in more detail below) provides the new leader with a value $\widehat{x}$ and a *certificate* $\widehat{\sigma}$ ensuring that $\widehat{x}$ is safe in the current view v. The certificate can be independently verified by any process. In the first view ($v = 1$), any value is safe ($\widehat{x} = x^{in}_{leader(1)}$) and there is no need for such a certificate ($\widehat{\sigma} = \bot$).

To propose a value in the normal case (illustrated in Figure 2.1a), the leader p sends the message $\text{PROPOSE}(\widehat{x}, v, \widehat{\sigma}, \widehat{\tau})$ to all processes, where $\widehat{\tau} = sign_p((\text{PROPOSE}, \widehat{x}, v))$.

When a process receives the proposal for the first time in a given view and ensures that $\widehat{\sigma}$ and $\widehat{\tau}$ are valid, it sends an ACK message containing the proposed value and a digital signature to every process. Once a process receives $n - f$ signed acknowledgments for the same pair $(\widehat{x}, v)$, it decides on the proposed value $\widehat{x}$.



(a) Normal case execution example. $\widehat{\tau} = \mathit{sign}_p((\text{PROPOSE}, \widehat{x}, v))$, where p is the identifier of the leader process that sends the PROPOSE message. $\phi_{ack} = \mathit{sign}_q((\text{ACK}, \widehat{x}, v))$, where q is the identifier of the process that sends the ACK message.



(b) View change execution example. $\phi_{vote} = \mathit{sign}_q((\text{VOTE}, vote_q, v))$ and $\phi_{ca} = \mathit{sign}_q((\text{CERTACK}, \widehat{x}, v))$, where q is the identifier of the process that sends the message.

Figure 2.1: Execution examples.

2.2.2 View change

Every process q locally maintains a variable $vote_q$, an estimate of the value to be decided, in the form (x, u, σ, τ), where x is a value, u is a view number, σ is the certificate ensuring that x is safe in view u, and τ is a signature for the tuple $(\text{PROPOSE}, x, u)$ produced by the leader of view u. If $vote_q = (x, u, \sigma, \tau)$, process q is said to vote for "value x in view u". Initially, the variable $vote_q$ has special value *nil*. When a correct process receives a PROPOSE message from the leader of its current view for the first time, the process updates its vote by adopting the values from the PROPOSE message (before sending the ACK message back to the leader). Note that once a correct replica changes its vote from *nil* to something else, it never changes the vote back to *nil*. A vote is said to be *valid* if either it is equal to *nil* or both σ and τ are valid with respect to x and u.

Whenever a correct process q changes its current view (let v be the new view number), it sends the message $\text{VOTE}(vote_q, \phi_{vote})$ to the leader of view v, where $\phi_{vote} = \mathit{sign}_q((\text{VOTE}, vote_q, v))$. When a correct replica finds itself to be the leader of its current view v, unless $v = 1$, it executes the view change protocol (illustrated in Figure 2.1b). First, it waits for $n - f$ valid votes and runs the selection algorithm to determine a safe value to propose ($\widehat{x}$). Then it communicates with other processes to create the certificate $\widehat{\sigma}$.

Selection algorithm. Let *votes* be the set of all valid votes received by the leader (with the ids and the signatures of the processes that sent these votes).

$|votes| \geq n - f$. If all votes in *votes* are equal to *nil*, then the leader simply selects its own input value ($x^{in}_{leader(v)}$).

Otherwise, let w be the highest view number contained in a valid vote. If there is only one value x such that there is a valid vote $(x, w, *, *)$ in *votes*, then x is selected.

Let us now consider the case when there are two or more values with valid votes in view w. As a correct leader issues at most one proposal in its view, the only reason for two different valid votes $m_1 = (x_1, w, \sigma_1, \tau_1)$ and $m_2 = (x_2, w, \sigma_2, \tau_2)$ to exist is that the leader q of view w is Byzantine (process q is said to have *equivocated*). Tuple $\gamma = (m_1, m_2)$ ca be treated as an undeniable evidence of q's misbehavior. As there are at most f faulty processes, the leader can then wait for $n - f$ votes *not including q's vote* (i.e., the leader may need to wait for exactly one more vote if $|votes| = n - f$ and *votes* contains a vote from q). After receiving this additional vote, it may happen that w is no longer the highest view number contained in a valid vote. In this case, the selection algorithm needs to be restarted.

Otherwise, if w remains the highest view number contained in a valid vote, let $votes'$ denote the $n - f$ valid votes from processes other than q. There are two cases to be considered:

(1) If there is a set $V \subset votes'$ of $2f$ valid votes for a value x, then x is selected;

(2) If no such value x is found, then any value is safe in view v. In this case, the leader simply selects its own input value ($x^{in}_{leader(v)}$).

Certificate creation. Let $\widehat{x}$ be the value selected by the selection algorithm. As is proven in Section 2.2.3, if the leader honestly follows the selection algorithm as described above, the selected value $\widehat{x}$ will be safe in the current view v. However, the leader also needs to create a certificate $\widehat{\sigma}$ that will prove to all other processes that $\widehat{x}$ is safe.

The naive way to do so is to simply let $\widehat{\sigma}$ be the set of all valid votes received by the leader. Any process will be able to verify the authenticity of the votes (by checking the digital signatures) and that the leader followed the selection

algorithm correctly (by simulating the selection process locally on the given set of votes).

However, the major problem with this solution is that the certificate sizes will grow without a limit in case of a long period of asynchrony. Recall that each vote contains a certificate. If each certificate $\widehat{\sigma}$ consisted of $n-f$ votes, then each vote would contain a certificate of its own, which, in turn, would consist of $n-f$ votes cast in an earlier view, and so on. If this naive approach is implemented carefully, the certificate size (and, hence, the certificate verification time) will be linear with respect to the current view number. While it may be sufficient for some applications (e.g., if long periods of asynchrony are assumed to never happen), a solution with bounded certificate size would be much more appealing.

In order to compress the certificate, an additional round-trip is added to the view change protocol. The leader sends the votes alongside the selected value $\widehat{x}$ to at least $2f+1$ different processes and waits for $f+1$ signed confirmations. The certificate $\widehat{\sigma}$ is the set of $f+1$ signatures from different replicas for the tuple $(\text{CERTACK}, \widehat{x}, v)$. Intuitively, since there are at most f Byzantine processes in total, it is sufficient to present signatures from $f+1$ replicas to prove that at least one correct replica verified that the leader performed the selection algorithm correctly and, hence, that $\widehat{x}$ is safe in view v. As a result, the size of a protocol message does not depend on the view number.

2.2.3 Proof of consistency

It is easy to see that the protocol satisfies the liveness property of consensus: once a correct leader is elected after GST, there is nothing to stop it from driving the protocol to termination. The extended validity property is immediate. Hence, in this section, only consistency is discussed in detail. It is shown that a correct leader always chooses a safe value in the view change protocol.

Our proofs are based on the following three quorum intersection properties (recall that $n = 5f - 1$):

(QI1) **Simple quorum intersection:** any two sets of $n-f$ processes intersect in at least one correct process. This follows from the pigeonhole principle. It is sufficient to verify that $2(n-f) - n \geq f + 1$, which is equivalent to

$n \geq 3f + 1$ and holds for $n = 5f - 1$ assuming that $f \geq 1$;

(QI2) **Equivocation quorum intersection #1:** if $Q_1 \subset \Pi$ such that $|Q_1| = n - f$ and $Q_2 \subset \Pi$ such that $|Q_2| = n - f$ and there are at most $f - 1$ Byzantine processes in Q_2, then $Q_1 \cap Q_2$ contains at least $2f$ correct processes. Again, by the pigeonhole principle, it is sufficient to verify that $2(n - f) - n \geq (f - 1) + 2f$, which is equivalent to $n \geq 5f - 1$;

(QI3) **Equivocation quorum intersection #2:** if $Q_1 \subset \Pi$ such that $|Q_1| = n - f$ and $Q_2 \subset \Pi$ such that $|Q_2| = 2f$ and there are at most $f - 1$ Byzantine processes in Q_2, then $Q_1 \cap Q_2$ contains at least one correct process. It is sufficient to verify that $(n - f) + 2f - n \geq (f - 1) + 1$, which holds for any values of n and f, $n \geq 2f$.

First, let us address the corner case when the leader receives no valid votes other than *nil*.

Lemma 2.1. *If the leader of view v receives nil from $n - f$ different processes during the view change, then any value is safe in v.*

Proof. Suppose, for contradiction, that at some point of the execution some value y is decided in a view w' smaller than v. Consider the set $Q_1 \subset \Pi$ of $n - f$ processes that acknowledged value y in w'. Consider also the set $Q_2 \subset \Pi$ of $n - f$ processes that sent *nil* to the leader of view v. By property (QI1), $Q_1 \cap Q_2$ contains at least one correct process.

A correct process only sends messages associated with its current view and it never decreases its current view number. Hence, it cannot send the vote in view v before sending the acknowledgment in view w'. If the correct process acknowledged value y in w' before sending the vote to the leader of view v, the vote would have not been *nil*. A contradiction. □

For the rest of this section, let v denote a view number and let w denote the highest view number contained in a valid vote received by the leader of view v during the view change protocol.

Lemma 2.2. *No value was or will ever be decided in any view w' such that $w < w' < v$.*

Proof. Suppose, for contradiction, that at some point of the execution some other value y is decided in $w'(w < w' < v)$. Let $Q_1 \subset \Pi$ be the set of $n - f$ processes that acknowledged value y in w' and let $Q_2 \subset \Pi$ be the set of $n - f$ processes that sent their votes to the leader of view v. By property (QI1), $Q_1 \cap Q_2$ contains at least one correct process.

A correct process only sends messages associated with its current view and it never decreases its current view number. Hence, it cannot send the vote in view v before sending the acknowledgment in view w'. If the correct process acknowledged value y in w' before sending the vote to the leader of view v, the vote would have contained a view number at least as large as w'. This contradicts the maximality of w. □

Lemma 2.3. *If there is only one value x such that there is a valid vote (x, w, σ, τ), then x is safe in view v.*

Proof. Suppose, for contradiction, that at some point of the execution some other value y is decided in a view w' smaller than v. By the validity of certificate σ, w' cannot be smaller than w. By Lemma 2.2, w' cannot be larger than w. Let us consider the remaining case $(w' = w)$.

The proof is mostly identical to the proofs of Lemmas 2.1 and 2.2. Nevertheless, it is repeated here for completeness.

Let $Q_1 \subset \Pi$ be the set of $n - f$ processes that acknowledged value y in w. Let $Q_2 \subset \Pi$ be the set of $n - f$ processes that sent their votes to the leader of view v. By (QI1), $Q_1 \cap Q_2$ contains at least one correct process.

A correct process only sends messages associated with its current view and it never decreases its current view number. Hence, it cannot send the vote in view v before sending the acknowledgment in view w. If the correct process acknowledged value y in w before sending the vote to the leader of view v, the vote would have contained either a view number larger than w (which contradicts the maximality of w) or the value y (which contradicts the uniqueness of x). □

Lemma 2.4. *If the leader detects an equivocation by process q and receives at least $2f$ valid votes for a value x in view w from processes other than q, then x is safe in view v.*

Proof. Suppose, for contradiction, that at some point of the execution some other value (y) is decided in a view w' smaller than v. By the validity of the certificates attached to the votes cast for value x, w' cannot be smaller than w. By Lemma 2.2, w' cannot be larger than w. Let us consider the remaining case ($w' = w$).

Let $Q_1 \subset \Pi$ be the set of $n - f$ processes that acknowledged value y in w. Let $Q_2 \subset \Pi$ be the set of $2f$ processes that cast votes for value x in view w. Since $q \notin Q_2$ and q is provably Byzantine, there are at most $f - 1$ Byzantine processes in Q_2. By (QI3), there is at least one correct process in $Q_1 \cap Q_2$. A correct process only adopts a vote before acknowledging the value from the vote and it never acknowledges 2 different values in the same view. Hence, $y = x$. A contradiction. □

Lemma 2.5. *If the leader detects an equivocation by process q and do not receive $2f$ or more valid votes for any value x in view w from processes other than q, then any value is safe in v.*

Proof. Suppose, for contradiction, that at some point of the execution some value y is decided in a view w' smaller than v. Let $m_1 = (y_1, w, \sigma_1, \tau_1)$ and $m_2 = (y_2, w, \sigma_2, \tau_2)$ be the two valid votes such that $y_1 \neq y_2$. By the validity of σ_1, no value other than y_1 was or will ever be decided in a view smaller than w. The same applies for value y_2. Hence, no value was or will ever be decided in a view smaller than w (i.e., w' is not smaller than w). By Lemma 2.2, w' is not larger than w.

Let us consider the remaining case ($w' = w$). Recall that the leader collects $n - f$ votes from processes other than q. By (QI2), the leader would have received at least $2f$ votes for the value y in view w or at least one vote for a value in a view larger than w. □

2.2.4 Generalized version

By applying the techniques from prior studies on fast Byzantine consensus [47, 62, 4], one can obtain a generalized version of our algorithm. The protocol will tolerate f Byzantine failures and will be able to decide a value in the common case after just two communication steps as long as the actual number of faults

does not exceed threshold t ($t \leq f$). The required number of processes will be $\max\{3f+2t-1, 3f+1\}$ (i.e., $n = 3f+2t-1$ if $t \geq 1$ and $n = 3f+1$ if $t = 0$). Note that, when $t = 1$, one obtains a Byzantine consensus protocol with optimal resilience ($n = 3f+2t-1 \underset{[t=1]}{=} 3f+1$) that is able to decide a value with optimal latency in the common case in presence of a single Byzantine fault. To the best of our knowledge, in all prior algorithms with optimal resilience ($n = 3f + 1$), the optimistic fast path could make progress only when all processes were correct.

2.3 Lower bound

In this section, it is shown that any f-resilient Byzantine consensus protocol that terminates within two message delays when the number of actual failures does not exceed t (such a protocol is called *two-step*) requires at least $3f + 2t - 1$ processes.

It is then shown in Section 2.3.3 that the lower bound of $n = 3f + 2t + 1$ processes (claimed by Martin and Alvisi [62]) holds for a special class of protocols assuming that the processes that propose values (called *proposers*) are disjoint from the processes responsible for replicating the proposed values (called *acceptors*).

2.3.1 Preliminaries

Let $\mathcal{V}$ be the domain of the consensus protocol (i.e., the set of possible input values). An *initial configuration* is a function $I : \Pi \to \mathcal{V}$ that maps processes to their input values. Note that although I maps all processes to some input values, Byzantine processes can pretend as if they have different inputs.

An *execution* of the protocol is the tuple $(I, \mathcal{B}, \mathcal{S})$, where I is an initial configuration, $\mathcal{B}$ is the set of Byzantine processes ($|\mathcal{B}| \leq f$), and $\mathcal{S}$ is a totally ordered sequence of steps taken by every process with timestamps (consisting of “send message”, “receive message”, and “timer elapsed” events). Multiple events may have the same timestamp, but they, nevertheless, must be arranged in a total order. If $\rho = (I, \mathcal{B}, \mathcal{S})$, execution ρ is said to *start from* initial configuration I.

In the proof of this lower bound, all processes are assumed to have access to perfectly synchronized local clocks that show exact time elapsed since the beginning of the execution. Note that this only strengthens the lower bound. If there is no algorithm implementing fast Byzantine consensus with $3f + 2t - 2$ or fewer processes in the model *with* synchronized clocks, then clearly there is no such algorithm without synchronized clocks.

In all executions that are considered in the proof, the delivery of a message takes at least Δ time units. Events that happen during the half-open time interval $[0, \Delta)$ are referred to as *the first round*. Events that happen during the half-open time interval $[\Delta, 2\Delta)$ are referred to as *the second round*, and so on. Hence, a message sent in round i may only be delivered in round $i + 1$ or later. State of a process "after the i-th round" is its state after all events with timestamp $i\Delta$ or smaller and before any events with higher timestamps.

Lemma 2.6. *Actions taken by correct processes during the first round depend exclusively on their inputs (i.e., on the initial configuration).*

Proof. Indeed, in the executions that are considered, during the first round, no messages can be delivered. Messages that are sent at the time 0 are delivered not earlier than at the time Δ, which belongs to the second round. As only deterministic algorithms are considered, all actions taken by the processes in the first round are based on their input values. □

Thanks to the liveness property of consensus, it is sufficient to consider only finite executions in which every correct process decides on some value at some point. Moreover, by the consistency property of consensus, all correct processes have to decide the same value. Let us call this value the *consensus value* of an execution and denote it with $c(\rho)$, where ρ is an execution.

Given an execution ρ and a process p, the *decision view* of p in ρ is the view of p at the moment when it triggers the DECIDE callback. The view consists of the messages p received (ordered and with the precise time of delivery) together with the state of p in the initial configuration of ρ. Note that the messages received by p *after* it triggers the callback are not reflected in the decision view.

Let ρ_1 and ρ_2 be two executions, and let p be a process which is correct in ρ_1 and ρ_2. Execution ρ_1 is *similar* to execution ρ_2 with respect to p, denoted as

$\rho_1 \overset{p}{\sim} \rho_2$, if the decision view of p in ρ_1 is the same as the decision view of p in ρ_2. If P is a set of processes, $\rho_1 \overset{P}{\sim} \rho_2$ is used as a shorthand for $\forall p \in P : \rho_1 \overset{p}{\sim} \rho_2$.

Lemma 2.7. *If there is a correct process $p \in \Pi$ such that $\rho_1 \overset{p}{\sim} \rho_2$, then $c(\rho_1) = c(\rho_2)$.*

Proof. Since only executions where all correct processes decide some value are considered, in executions ρ_1 and ρ_2, process p had to decide values $c(\rho_1)$ and $c(\rho_2)$ respectively. However, since, at the moment of the decision, process p is in the same state in both executions and only deterministic processes are considered, p has to make identical decisions in the two executions. Hence, $c(\rho_1) = c(\rho_2)$. □

Execution $\rho = (I, \mathcal{B}, \mathcal{S})$ is called *$\mathcal{T}$-faulty two-step execution*, where $\mathcal{T} \subset \Pi$ and $|\mathcal{T}| = t$, iff:

1. All processes in $\Pi \setminus \mathcal{T}$ are correct and all processes in $\mathcal{T}$ are Byzantine (i.e., $\mathcal{B} = \mathcal{T}$);

2. Local computation is instantaneous. In particular, if one process receives a messages from another process at the time t and sends a reply without waiting, the reply will be sent also at time t and will arrive at the time $t + \Delta$;

3. Processes in $\mathcal{T}$ honestly follow the protocol during the first round and do not take any actions in later rounds. In particular, they do not send any messages at the time 2Δ or later;

4. Delivery of each message between each pair of processes takes precisely Δ time units;

5. Every correct process makes a decision not later than at the time 2Δ.

The following lemma explains how the weak validity property of consensus dictates the output values of $\mathcal{T}$-faulty two-step executions.

Lemma 2.8. *For any consensus protocol with weak validity, if all processes have the same input value x ($\forall p : I(p) = x$), for any $\mathcal{T}$-faulty two-step execution ρ starting from I, the consensus value $c(\rho) = x$.*

Proof. Let T be the moment in time such that, in execution ρ, by that moment, all correct processes have invoked the DECIDE callback. Let ρ' be an execution identical to ρ with the exception that processes in $\mathcal{T}$ are not Byzantine, but just slow. The messages they send after the first round do not reach other processes until after the moment T. The processes in $\Pi \setminus \mathcal{T}$ have no way to distinguish ρ' from ρ until they receive the delayed messages, which happens already after they invoke the DECIDE callback. Hence, $\rho' \overset{\Pi \setminus \mathcal{T}}{\sim} \rho$ and, by Lemma 2.7, $c(\rho') = c(\rho)$. By the weak validity property of the consensus protocol, if all processes have x as their input value, then $c(\rho') = x$. $\square$

Protocol $\mathcal{P}$ is a *two-step consensus protocol* if it satisfies the following conditions:

1. $\mathcal{P}$ is a consensus protocol with weak validity, as defined in Section 1.1;
2. $\exists \mathcal{M} \subset \Pi$ such that $|\mathcal{M}| \geq 2t+1$ and $\forall I$ – initial configuration: $\forall \mathcal{T} \subset \mathcal{M}$ such that $|\mathcal{T}| = t$: there is a $\mathcal{T}$-faulty two-step execution starting from I.

In other words, if all Byzantine processes belong to a known set of *"suspects"* $\mathcal{M}$ and fail by simply crashing at the moment Δ, local computation is immediate, and the network is synchronous, the algorithm must be able to make sure that all processes decide some value after just 2 steps. Otherwise, when the environment is not so gracious (e.g., the network is not synchronous from the beginning or some processes in $\Pi \setminus \mathcal{M}$ are Byzantine), the protocol is allowed to terminate after more than 2 steps. Intuitively, the optimistic fast path of the protocol can rely on the correctness of up to $n - (2t + 1)$ "leaders".

As an example, let us see that the protocol proposed in this thesis is a two-step consensus protocol according to this definition. Suppose that there are at least $3f+2t-1$ processes and $f \geq 1$. Recall that *leader*(1) is the leader for view 1. Let $p =$ *leader*(1). Let $\mathcal{M} = \Pi \setminus \{p\}$. Then, for any initial configuration and any $\mathcal{T}$ with $|\mathcal{T}| = t$, the following $\mathcal{T}$-faulty two-step execution exists:

1. p proposes its input value $x = x_p^{in}$ at time 0 with the message PROPOSE$(x, 1, \bot, \widehat{\tau})$;
2. The other processes, including those in $\mathcal{T}$, honestly follow the protocol and do nothing during the first round;

3. At time Δ, all processes receive the propose message. Among them, $3f + t - 1$ processes are correct (including the leader p), and they respond with an acknowledgment $\textsc{ack}(x, 1, \phi_{ack})$;

4. At time 2Δ, all the correct processes receive all the ACK messages and decide via $\textsc{Decide}(x)$.

2.3.2 Optimality of the proposed algorithm

Process $p \in \Pi$ is said to be *influential* if there are two initial configurations (I and I') such that $\forall q \neq p : I(q) = I'(q)$ and two non-intersecting sets of suspects not including p of size t ($\mathcal{T}, \mathcal{T}' \subset \mathcal{M} \setminus \{p\}$, $|\mathcal{T}| = |\mathcal{T}'| = t$, and $\mathcal{T} \cap \mathcal{T}' = \emptyset$) such that there are a $\mathcal{T}$-faulty execution ρ, and a $\mathcal{T}'$-faulty execution ρ' with different consensus values ($c(\rho) \neq c(\rho')$).

Intuitively, a process is influential if its input value under certain circumstances can affect the outcome of the fast path of the protocol. In Theorem 2.11, it is proven that, if the number of processes is smaller than $3f + 2t - 1$, an influential process can use its power to force disagreement.

Lemma 2.9. *For any two-step consensus protocol, there is at least one influential process.*

Proof. $\forall i \in \{0, \dots, n\}$: let I_i be the initial configuration in which the first i processes have the input value 1 and the remaining processes have the input value 0. In particular, in I_0, all processes have the input value 0, and, in I_n, all processes have the input value 1. By the definition of a two-step consensus protocol, $\forall i : \forall \mathcal{T} \subset \mathcal{M}$ such that $|\mathcal{T}| = t$: there must be a $\mathcal{T}$-faulty two-step execution starting from I_i. Moreover, by Lemma 2.8, $\forall \mathcal{T}$: all $\mathcal{T}$-faulty two-step executions starting from I_0 (resp., I_n) have the consensus value 0 (resp., 1). Let *pred*(i) be the predicate "there is a set $\mathcal{T}_1 \subset (\mathcal{M} \setminus \{p_i\})$ such that there is a $\mathcal{T}_1$-faulty two-step execution with consensus value 1 starting from I_i". We know that *pred*(0) $=$ *false* and *pred*(n) $=$ *true*. Hence, as we consider all number from 0 to n, *pred* must change its value from *false* to *true* at least once. Let j be such a number that *pred*$(j-1)$ $=$ *false* and *pred*(j) $=$ *true*, $j \geq 1$. Let $\mathcal{T}_1$ be the set of suspects defined in the predicate. By the definition of $\mathcal{T}_1$, $p \notin T_1$.

It follows that there is a minimum number $j \geq 1$ such that there is a set $\mathcal{T}_1$ such that there is a $\mathcal{T}_1$-faulty two-step execution with consensus value 1 starting from I_j.

Note that $p_j \notin \mathcal{T}_1$. Indeed, if $p_j \in \mathcal{T}_1$, then the input of p_j would not be able to affect the consensus value of any $\mathcal{T}_1$-faulty two-step execution (p_j simply would not participate in that execution). In that case, there would be a $\mathcal{T}_1$-faulty two-step execution with consensus value 1 starting from I_{j-1}, which contradicts the choice of j.

Let $\mathcal{T}_0 \subseteq \mathcal{M}$ be a set of suspect such that $|\mathcal{T}_0| = t$, $p_j \notin \mathcal{T}_0$, and $\mathcal{T}_0 \cap \mathcal{T}_1 = \emptyset$. Such a set exists because $|\mathcal{M}| = 2t+1$. By the definition of j, all $\mathcal{T}_0$-faulty two-step executions starting from initial configuration I_{j-1} have consensus value 0, and, by the definition of a two-step consensus protocol, there is at least one such execution.

It can be seen that p_j is an influential process. Indeed, I_{j-1} and I_j differ only in the input of process p_j, ρ_0 and ρ_1 are $\mathcal{T}_0$- and $\mathcal{T}_1$-faulty executions starting from I_{j-1} and I_j respectively, $\mathcal{T}_0 \cap \mathcal{T}_1 = \emptyset$, $p_j \notin (\mathcal{T}_0 \cup T_1)$, and $c(\rho_0) \neq c(\rho_1)$. □

Let us prove that $3f + 2t - 1$ is optimal in the special case when $t = 1$.

Theorem 2.10. *There is no two-step consensus protocol (assuming weak validity) with $f \geq 1$ and $t = 1$ that can be executed on $3f + 2t - 2 = 3f$ processes.*

Proof. This is a special case of the more general lower bound [65] that states that any Byzantine consensus protocol in partially synchronous model requires at least $3f + 1$ processes. □

Theorem 2.11. *There is no two-step consensus protocol (assuming weak validity) with $f \geq t \geq 2$ that can be executed on $3f + 2t - 2$ processes.*

Proof. Suppose, for contradiction, that there is such a two-step consensus protocol that can be executed on a set Π of $3f + 2t - 2$ processes ($f \geq t \geq 2$). By Lemma 2.9, there is an influential process p, two initial configurations (I' and I'') that differ only in the input of process p, two sets of suspects ($\mathcal{T}', \mathcal{T}'' \subset \Pi \setminus \{p\}$, $|\mathcal{T}'| = |\mathcal{T}''| = t$, and $\mathcal{T}' \cap \mathcal{T}'' = \emptyset$), and two executions: a $\mathcal{T}'$-faulty execution ρ' starting from I' and a $\mathcal{T}''$-faulty execution ρ'' starting from I'', such that

	$\{p\}$	P_1	P_2	P_3	P_4	P_5
ρ_5		s_1	s_2	s_3	s_4	🥷
ρ_4	🥷	s_1	s_2	s_3	🥷	t_5
ρ_3	🥷	s_1	s_2	🥷	t_4	t_5
ρ_2	🥷	s_1	🥷	t_3	t_4	t_5
ρ_1		🥷	t_2	t_3	t_4	t_5
size	1	t	$f-1$	$f-1$	$f-1$	t

ρ_5–ρ_4: similar for P_3
ρ_4–ρ_3: similar for P_1, P_2, and P_5
ρ_3–ρ_2: similar for P_1, P_4, and P_5
ρ_2–ρ_1: similar for P_3

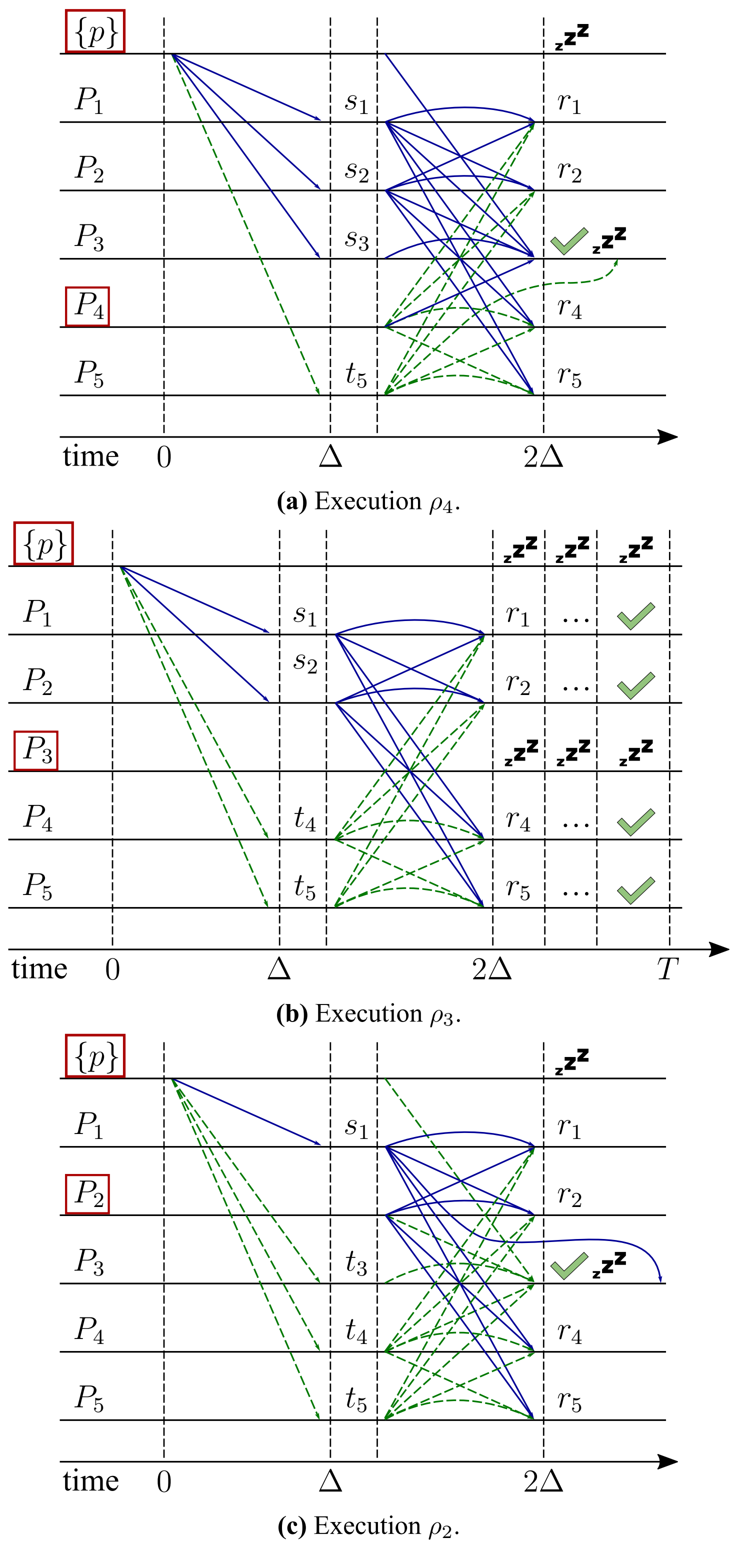
Figure 2.2: A visualization for the proof setup of the lower bound when $f \geq t \geq 2$. The rows are executions and the columns are groups (subsets) of the processes. A group that is byzantine will be denoted with 🥷. The s_i and t_i correspond to the state of the process after the first round.

$c(\rho') \neq c(\rho'')$. Without loss of generality, let us assume that $c(\rho') = 0$ and $c(\rho'') = 1$.

The set $\Pi \setminus \{p\}$ is partitioned into five groups: $P_1, \ldots, P_5$, where $P_1 = \mathcal{T}''$, $P_5 = \mathcal{T}'$, and $|P_2| = |P_3| = |P_4| = f-1$. The partition is depicted in Figure 2.2. Note that $|P_1| + \ldots + |P_5| + |\{p\}| = 2t + 3(f-1) + 1 = 3f + 2t - 2 = |\Pi|$.

Five executions are constructed, ρ_1 through ρ_5 ($\rho_1 = \rho''$ and $\rho_5 = \rho'$), such that for all $i \in \{1, \ldots, 5\}$, group P_i is Byzantine in ρ_i, and for all $j \neq i$, group P_j is correct in ρ_i. The influential process p is Byzantine for ρ_2, ρ_3, ρ_4 and is correct for ρ_1 and ρ_5; hence there are exactly f Byzantine actors for each execution as $|P_2| = |P_3| = |P_4| = f-1$. The goal is to show that each pair of adjacent executions will be similar for at least one correct process set, who will then decide the same value. This would naturally lead to a contradiction, since Lemma 2.7 implies that $0 = c(\rho_5) = \cdots = c(\rho_1) = 1$.

We can assume that p can send one of two types of messages in the first round: 0 or 1, where sending 0 causes execution ρ_5 to happen and sending 1 causes execution ρ_1 to happen. Recall that the initial configurations of ρ_1 and ρ_5 differ only in the input value of p. By Lemma 2.6, all actions taken by correct processes other than p **during** the first round will be the same in all executions. Additionally, for all i, in execution ρ_i, the processes in the Byzantine group P_i will act as if they are correct. Hence, the only process that acts differently in different executions during the first round is p. For each execution ρ_i, let p send 0 to the processes in P_j for $j < i$ and send 1 to the processes in P_j for $j > i$ (processes in P_i are Byzantine in ρ_i, so it does not matter what is sent to them).



(a) Execution ρ_4.

(b) Execution ρ_3.

(c) Execution ρ_2.

Figure 2.3: Executions ρ_2, ρ_3, and ρ_4. Solid blue lines and dashed green arrows represent messages identical to messages sent in ρ_5 and ρ_1 respectively. Green tick symbol means that all processes in the group decide a value. Messages from all processes other than p in the first round are identical in all executions and omitted on the picture for clarity. Messages sent in the second round to process p in both executions and to group P_3 in ρ_3 are also omitted as these processes are Byzantine and do not take any further steps in these executions after the second round. s_i and t_i represent states of correct processes after the first round in ρ_5 and ρ_1 respectively. r_i represent states of correct processes after the second round in ρ_3.

Note that for $i = 1$ or 5, this is equivalent to p honestly following the protocol for some initial configuration (i.e. sending the same message to all processes). For the other i, p is a Byzantine process that is role-playing a different initial configuration for each of the two nontrivial partitions of the other processes. Thus, after the first round, each group P_i can be assumed to take just one of two states s_i or t_i, where s_i is consistent with ρ_5 and t_i is consistent with ρ_1, solely dependent on which of the two messages p sends to them. See Figures 2.2 and 2.3. Let us see how ρ_2, ρ_3, and ρ_4 are constructed.

Execution ρ_4, second round (the actions of subsequent rounds in ρ_4 will be specified at a later time):

- Recall that $\{p\} \cup P_4$ are Byzantine. They ($\{p\} \cup P_4$) will send messages to group P_3 in exactly the same fashion as in ρ_5. For other processes, P_4 will act in exactly the same fashion as in ρ_1 (i.e., as if they are correct and were in the state t_5 after the first round). Process p will simply remain silent for the rest of the execution;

- P_5 is now honest, unlike it was in ρ_5. However, the messages from P_5 to P_3 are delayed and do not reach the recipients until after the time 2Δ. Other messages sent by P_5 in the second round are delivered in a timely fashion;

- P_3 is still honest but slow. Its messages will not be received by any other process until a finite moment in time T that will be specified later;

- All messages sent to P_3 by processes other than P_5 in the second round will be delivered at the exact same times and in the exact same order as in ρ_5.

Let us now look from the P_3's perspective. During the time interval $[0, 2\Delta]$, P_3 will not be able to distinguish this execution from ρ_5, since it receives the exact same messages from $\{p\}, P_1, P_2, P_4$, which all have the same state as in ρ_5 after the first round (or in P_4's case can fake the same state), and hears nothing from P_5 in both executions. Thus, by the time 2Δ, processes in P_3 will achieve

exactly the same state as in ρ_5 and will decide 0 as well (a reminder that this decision is done in silence, as P_3's messages will not be received by anyone else until the time T). Therefore, $\rho_5 \overset{P_3}{\sim} \rho_4$.

Execution ρ_2, second round: Recall that $\{p\} \cup P_2$ are Byzantine. ρ_2 is constructed similarly to ρ_4, but with the roles switched by both the symmetries $\rho_i \Leftrightarrow \rho_{6-i}$ and $P_i \Leftrightarrow P_{6-i}$. In particular, $\{p\} \cup P_2$ will send messages to group P_3 in exactly the same fashion as in ρ_1. By a symmetric argument as above, it can be concluded that by the time 2Δ, processes in P_3 will achieve exactly the same state as in ρ_1 and will decide 1 as well. Therefore, $\rho_1 \overset{P_3}{\sim} \rho_2$.

Execution ρ_3: First, let us look at what happens during the second round:

- In ρ_2, the Byzantine P_2 sends the same messages to P_1, P_4, and P_5 as its honest version does in ρ_4;
- In ρ_4, the Byzantine P_4 sends the same messages to P_1, P_2, and P_5 as its honest version does in ρ_2;
- P_3 is slow in both, so everybody has the same interaction with P_3;
- p is Byzantine, so it is possible for us to assume that p sends the same messages to each P_i during the two executions.

The important point is this: **after the second round, it can be assumed that every non-P_3 process has the same state (or if Byzantine in one of them, can act as if it were in the same state) in the two executions ρ_2 and ρ_4.** More precisely, both executions be continued (as long as P_3's messages do not reach anyone) as if the processes in $P_1 \cup P_2 \cup P_4 \cup P_5$ were the only correct processes starting from some state r after completing their second step.

Let us now construct ρ_3. Recall that $\{p\} \cup P_3$ are Byzantine. Let all processes in P_3 crash permanently at the end of the first round. The key idea with ρ_3 is that by the discussion we just had, it can be assumed that after the second round its (honest) processes $P_1 \cup P_2 \cup P_4 \cup P_5$ are in the same group state c as their counterparts in ρ_2 and ρ_4. Since there are only f Byzantine processes total, by

the liveness property of consensus, there must exist some execution that obtains consensus at some moment T. Let this execution be ρ_3.

Executions ρ_2 and ρ_4, later rounds: We are now finally ready to complete executions ρ_2 and ρ_4. Since they are symmetric, let us start by looking at ρ_4. It is specified what happens to ρ_4 up through the second round. Now, for the future rounds, emulate ρ_3 until the moment T, keeping P_3 silent. This execution is identical to ρ_3, so all the correct processes (in particular, all processes in P_1) will decide. This means $\rho_4 \overset{P_1}{\sim} \rho_3$. By a symmetric argument, $\rho_2 \overset{P_5}{\sim} \rho_3$. Hence, each adjacent pair of executions is similar to some processes, which leads to a contradiction. □

2.3.3 Optimality of FaB Paxos

While $n = 3f + 2t + 1$ is not optimal for fast Byzantine consensus algorithms in general, it is optimal for a special class of Paxos-like algorithms that separate proposers from acceptors. In Paxos [56], one of the first crash fault-tolerant solutions for the consensus problem, Leslie Lamport suggested a model with three distinct types of processes: *proposers*, *acceptors*, and *learners*. Proposers are "leaders" and they are responsible for choosing a safe value and sending it to acceptors. Acceptors store the proposed values and help the new leader to choose a safe value in case previous leader crashes. Finally, learners are the processes that trigger the DECIDE callback and use the decided value (e.g., they can execute replicated state machine commands). In this model, the consensus problem requires all learners to decide the same value. The Byzantine version of Paxos [53] requires presence of at least one correct proposer and $n = 3f + 1$ acceptors, where f is the possible number of Byzantine faults among acceptors.

In the proposed algorithm, when a correct leader (proposer) sees that some prior leader equivocated, it uses this fact to exclude one acceptor from consideration as it is provably Byzantine. This trick only works when the set of proposers is a subset of the set of acceptors. Moreover, this trick seems to be crucial for achieving the optimal resilience ($n = \max\{3f + 2t - 1, 3f + 1\}$). When the set of proposers is disjoint from the set of acceptors, or even if there is just one proposer that is not an acceptor, it can be shown that $n = 3f + 2t + 1$ is optimal.

In order to obtain the $n = 3f + 2t + 1$ lower bound for the model where proposers are separated from acceptors, we need to make just two minor modifications to our proof of theorem 2.11. First of all, the influential process p is no longer an acceptor. Hence, we are left with only 5 groups of acceptors ($P_1, \dots, P_5$) instead of 6 ($\{p\}, P_1, \dots, P_5$). Second, the groups of acceptors P_2, P_3, and P_4 can now be of size f instead of $f - 1$ (since p is no longer counted towards the quota of f Byzantine acceptors). After these two modifications, the proof shows that there is no two-step consensus protocol with $n = |P_1| + \cdots + |P_5| = 3f + 2t$ or fewer acceptors.

2.4 Related work

Kursawe [47] was the first to implement a fast (two-step) Byzantine consensus protocol. The protocol is able to run with $n = 3f + 1$ processes, but it is able to commit in two steps only when all n processes follow the protocol and the network is synchronous. Otherwise, it falls back to a randomized asynchronous consensus protocol.

Martin and Alvisi [62] present FaB Paxos–a fast Byzantine consensus protocol with $n = 5f + 1$. Moreover, they present a parameterized version of the protocol: it runs on $n = 3f + 2t + 1$ processes ($t \leq f$), tolerates f Byzantine failures, and is able to commit after just two steps in the common case when the leader is correct, the network is synchronous, and at most t processes are Byzantine. In the same paper, the authors claim that $n = 3f + 2t + 1$ is the optimal resilience for a fast Byzantine consensus protocol. In this thesis, it is shown that this lower bound only applies to the class of protocols that separate processes that execute the protocol (acceptors) from the process that propose values (proposers).

Bosco [68] is a Byzantine agreement that is able to commit values after just one communication step when there is no contention (i.e., when all processes propose the same value). In order to tolerate f failures, the algorithm needs $5f + 1$ or $7f + 1$ processes, depending on the desired validity property.

Zyzzyva [45], UpRight [27], and SBFT [40] are practical systems that build upon the ideas from FaB Paxos to provide optimistic fast path. Zyzzyva [45]

and UpRight [27] aim to replace crash fault-tolerant solutions in datacenters. The evaluations in these papers demonstrate that practical systems based on fast Byzantine consensus protocols can achieve performance comparable with crash fault-tolerant solutions while providing additional robustness of Byzantine fault-tolerance. In [40], Gueta et al. explore the applications of fast Byzantine consensus to permissioned blockchain. Due to the high number of processes usually involved in such protocols, the results of this thesis are less relevant for this setting.

In [3] and [4], Abraham et al. demonstrate and fix some mistakes in FaB Paxos [62] and Zyzzyva [45]. Moreover, they combine the ideas from the two algorithm into a new one, called Zelma. This algorithm lies at the core of the SBFT protocol [40].

In [32], the authors claim that their protocol, called hBFT, achieves the two-step latency despite f Byzantine failures with only $3f+1$ processes (as opposed to $5f-1$ required by the proposed lower bound). However, in a later paper [67], it was shown that hBFT fails to provide the consistency property of consensus.

In a concurrent work [6], Abraham et al. present a number of upper and lower bounds on the latency and resilience of the so-called "Byzantine broadcast" problem. In particular, they show that the problem of partially-synchronous validated Byzantine broadcast is solvable within 2 message delays iff $n \geq 5f-1$. The problem that they consider is similar to leader-based consensus with external validity. The results for the partially-synchronous systems are very similar to the results presented in this thesis and are based on the same ideas. In a complementary note [5], they present a practical Byzantine fault-tolerant state machine replication protocol based on these ideas. Despite the similarities, the lower bound provided in Section 2.3 of this thesis is more general than the one presented in [6] as it is not limited to leader-based algorithms and encompasses double-threshold algorithms.

Conclusion

In this work, two important problems in Byzantine fault-tolerant distributed computing were addressed: asynchronous reconfiguration and resilience of fast partially-synchronous consensus. For the first problem, there were no known solutions prior to this work. For the second problem, a new upper bound was proposed and an oversight in the lower bound was fixed.

References

[1] List of accepted papers, 2021. URL: https://www.podc.org/podc2021/list-of-accepted-papers/.

[2] List of ethereum client implementations, 2021. URL: https://ethereum.org/en/developers/docs/nodes-and-clients/#clients.

[3] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance. *arXiv preprint arXiv:1712.01367*, 2017.

[4] Ittai Abraham, Guy Gueta, Dahlia Malkhi, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance: Thelma, velma, and zelma. *arXiv preprint arXiv:1801.10022*, 2018.

[5] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Fast validated byzantine broadcast. *arXiv preprint arXiv:2102.07932*, 2021.

[6] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: a complete categorization. *arXiv preprint arXiv:2102.07240*, 2021.

[7] Marcos K Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, Alexander Shraer, et al. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS*, (102):84–108, 2010.

[8] Marcos Kawazoe Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, 2011.

[9] Marcos Kawazoe Aguilera and Sam Toueg. A simple bivalency proof that t-resilient consensus requires t+ 1 rounds. *Information Processing Letters*, 71(3-4):155–158, 1999.

[10] Eduardo Alchieri, Alysson Bessani, Fabíola Greve, and Joni da Silva Fraga. Efficient and modular consensus-free reconfiguration for fault-tolerant storage. In *OPODIS*, pages 26:1–26:17, 2017.

[11] Bowen Alpern and Fred B Schneider. Recognizing safety and liveness. *Distributed computing*, 2(3):117–126, 1987.

[12] Lorenzo Alvisi, Dahlia Malkhi, Evelyn Pierce, Michael K Reiter, and Rebecca N Wright. Dynamic byzantine quorum systems. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pages 283–292. IEEE, 2000.

[13] James Aspnes, Hagit Attiya, and Keren Censor. Max registers, counters, and monotone circuits. In *PODC*, pages 36–45, 2009.

[14] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[15] Hagit Attiya, Hyun Chul Chung, Faith Ellen, Saptaparni Kumar, and Jennifer L. Welch. Emulating a shared register in a system that never stops changing. *IEEE Trans. Parallel Distrib. Syst.*, 30(3):544–559, 2019.

[16] Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Comput.*, 8(3):121–132, 1995.

[17] Roberto Baldoni, Silvia Bonomi, Anne-Marie Kermarrec, and Michel Raynal. Implementing a register in a dynamic distributed system. In *ICDCS*, pages 639–647, 2009.

[18] Mihir Bellare and Sara K Miner. A forward-secure digital signature scheme. In *Annual International Cryptology Conference*, pages 431–448. Springer, 1999.

[19] Xavier Boyen, Hovav Shacham, Emily Shen, and Brent Waters. Forward-secure signatures with untrusted update. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 191–200, 2006.

[20] Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making byzantine consensus live. In *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[21] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, 2016.

[22] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[23] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. *Journal of Cryptology*, 20(3):265–294, 2007.

[24] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[25] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. Base: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 21(3):236–269, 2003.

[26] Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, volume 1, pages 3–9, 1978.

[27] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 277–290, 2009.

[28] Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments

by merely broadcasting messages. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 26–38. IEEE, 2020.

[29] Luciano Freitas de Souza, Petr Kuznetsov, Thibault Rieutord, and Sara Tucci-Piergiovanni. Accountability and reconfiguration: Self-healing lattice agreement. *arXiv preprint arXiv:2105.04909*, 2021.

[30] John R Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002.

[31] Manu Drijvers, Sergey Gorbunov, Gregory Neven, and Hoeteck Wee. Pixel: Multi-signatures for consensus. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, August 2020. USENIX Association. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/drijvers`.

[32] Sisi Duan, Sean Peisert, and Karl N Levitt. hbft: speculative byzantine fault tolerance with minimum cost. *IEEE Transactions on Dependable and Secure Computing*, 12(1):58–70, 2014.

[33] Partha Dutta and Rachid Guerraoui. The inherent price of indulgence. *Distributed Computing*, 18(1):85–98, 2005.

[34] Jose Faleiro, Sriram Rajamani, Kaushik Rajan, Ganesan Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In *PODC*, pages 125–134, 2012.

[35] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[36] Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *PODC*, pages 143–152, 1998.

[37] Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *DISC*, pages 140–153, 2015.

[38] Seth Gilbert, Nancy A Lynch, and Alexander A Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, 2010.

[39] Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, and Andrei Tonkikh. Dynamic byzantine reliable broadcast. In *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[40] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 568–580. IEEE, 2019.

[41] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[42] Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *DISC*, pages 154–169, 2015.

[43] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults–a tutorial. In *Latin-American Symposium on Dependable Computing*, pages 366–368. Springer, 2003.

[44] Anne-Marie Kermarrec and Maarten Van Steen. Gossiping in distributed systems. *ACM SIGOPS operating systems review*, 41(5):2–7, 2007.

[45] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, 2007.

[46] Saptaparni Kumar and Jennifer L. Welch. Byzantine-tolerant register in a system with continuous churn. *CoRR*, abs/1910.06716, 2019. URL: `http://arxiv.org/abs/1910.06716`, `arXiv:1910.06716`.

[47] Klaus Kursawe. Optimistic byzantine agreement. In *21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings.*, pages 262–267. IEEE, 2002.

[48] Petr Kuznetsov, Yvonne-Anne Pignolet, Pavel Ponomarev, and Andrei Tonkikh. Permissionless and asynchronous asset transfer [technical report]. *arXiv preprint arXiv:2105.04966*, 2021.

[49] Petr Kuznetsov, Thibault Rieutord, and Sara Tucci-Piergiovanni. Reconfigurable lattice agreement and applications. In *OPODIS*, 2019.

[50] Petr Kuznetsov and Andrei Tonkikh. Asynchronous reconfiguration with byzantine failures. In *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[51] Petr Kuznetsov, Andrei Tonkikh, and Yan X Zhang. Revisiting optimal resilience of fast byzantine consensus. *arXiv preprint arXiv:2102.12825*, 2021.

[52] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications*, 1978.

[53] Leslie Lamport. Byzantizing paxos by refinement. In *International Symposium on Distributed Computing*, pages 211–224. Springer, 2011.

[54] Leslie Lamport. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. 2019.

[55] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[56] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[57] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, 2010.

[58] Barbara Liskov and James Cowling. Viewstamped replication revisited. 2012.

[59] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed computing*, 11(4):203–213, 1998.

[60] Tal Malkin, Daniele Micciancio, and Sara Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 400–417. Springer, 2002.

[61] J-P Martin and Lorenzo Alvisi. A framework for dynamic byzantine storage. In *International Conference on Dependable Systems and Networks, 2004*, pages 325–334. IEEE, 2004.

[62] J-P Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

[63] Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine smr. *arXiv preprint arXiv:2002.07539*, 2020.

[64] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17, 1988.

[65] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[66] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *SSS*, pages 386–400, 2011.

[67] Nibesh Shrestha, Mohan Kumar, and SiSi Duan. Revisiting hbft: Speculative byzantine fault tolerance with minimum cost. *arXiv preprint arXiv:1902.08505*, 2019.

[68] Yee Jiun Song and Robbert van Renesse. Bosco: One-step byzantine asynchronous consensus. In *International Symposium on Distributed Computing*, pages 438–450. Springer, 2008.

[69] Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: A tutorial (tutorial). In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[70] Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: Abstraction and optimal asynchronous solution. In *DISC*, pages 40:1–40:15, 2017.

[71] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

[72] Xiong Zheng, Changyong Hu, and Vijay K. Garg. Lattice agreement in message passing systems. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 41:1–41:17, 2018.

A Reconfiguration Appendix

A.1 Pseudocode for the DBLA implementation

In this section, the complete pseudocode of the Dynamic Byzantine Lattice Agreement abstraction, defined in Section 1.4, is provided. The pseudocode is split into two parts: Algorithm 4 describes the behavior of a correct client and Algorithm 5 describes the behavior of a correct replica. In the beginning of each algorithm, all parameters, global variables, and auxiliary functions used in the pseudocode are defined.[7]

Additionally, "**RB-Broadcast** $\langle$DESCRIPTOR, *msgParams* …$\rangle$" is used to denote a call to the *global reliable broadcast* primitive introduced in Section 1.3.6, and "**URB-Broadcast** $\langle$DESCRIPTOR, *msgParams* …$\rangle$ **in** C" is used to denote a call to the *local uniform reliable broadcast* primitive in configuration C (see Section 1.3.6).

Execution environment. Single-threaded execution of the pseudocode is assumed. The lines of code are to be executed one by one in a sequential order. Some events (such as message delivery or an assignment to a global variable) may activate some handlers, but the execution of these handlers is delayed. However, some fairness is assumed, in a sense that if some handler remains "ready" indefinitely, it will eventually be executed. Sometimes waiting is explicitly mentioned in the code (e.g., Algorithm 4, line 85). In these places, the control flow may switch to other blocks. It may later return to the line after the "**wait for**" statement if the condition in the statement is satisfied.

Notation. The notation "let $var = expression$" is to denote an assignment to a local variable and "$var \leftarrow expression$" to denote an assignment to a global variable (they are usually defined in the "Global variables" section).

[7] A global variable is a variable that can be accessed from anywhere in the code executed *by the same process*. This is not to be confused with shared variables in the shared memory model.

Algorithm 4 DBLA: code for client p

Parameters:

68: Lattice of configurations $\mathcal{C}$ and the initial configuration C^{init}

69: The object lattice $\mathcal{L}$ and the initial value V^{init}u

70: Boolean functions VerifyHistory(h, σ) and VerifyInputValue(v, σ)

Global variables:

71: *history* $\subseteq \mathcal{C}$, initially $\{C^{init}\}$ ▷ local history of this process

72: $\sigma_{history} \in \Sigma$, initially $\perp$ ▷ proof for the local history

73: *curVals* $\subseteq \mathcal{L} \times \Sigma$, initially $\{(V^{init}, \perp)\}$ ▷ known verifiable input values with proofs

74: *status* $\in \{$*inactive*, *proposing*, *confirming*$\}$, initially *inactive*

75: *seqNum* $\in \mathbb{Z}$, initially 0 ▷ used to match requests with responses

76: $acks_1$, initially $\emptyset$ ▷ a set of pairs of form (*processId*, *sig*)

77: $acks_2$, initially $\emptyset$ ▷ a set of pairs of form (*processId*, *sig*)

Auxiliary functions:

78: HighestConf(h) ▷ returns the highest configuration in history h

79: ContainsQuorum$(acks, C)$ ▷ returns *true* iff $\exists Q \in quorums(C)$ such that $\forall r \in Q : (r, *) \in acks$

80: JoinAll(vs) ▷ returns the lattice join of all elements in *vs*

81: VerifyInputValues(vs) ▷ returns *true* iff $\forall (v, \sigma) \in vs$: VerifyInputValue(v, σ)

82: FSVerify(m, r, s, t) ▷ verifies forward-secure signature (see Section 1.1)

83: **operation** Propose(v, σ)

84: Refine$(\{(v, \sigma)\})$

85: **wait for** ContainsQuorum$(acks_2$, HighestConf$(history))$

86: *status* $\leftarrow$ *inactive*

87: let $\sigma = (curVals, history, \sigma_{history}, acks_1, acks_2)$

88: **return** (JoinAll$(curVals), \sigma)$

89: **operation** UpdateHistory(h, σ)

90: **RB-Broadcast** $\langle$NEWHISTORY, h, $\sigma\rangle$

91: **function** VerifyOutputValue(v, σ)

92: **if** $\sigma = \perp$ **then return** $v = V^{init}$

93: let $(vs, h, \sigma_h, proposeAcks, confirmAcks) = \sigma$

94: let $C =$ HighestConf(h)

95: **return** JoinAll$(vs) = v \wedge$ VerifyHistory(h, σ_h)

96: $\wedge$ ContainsQuorum$(proposeAcks, C) \wedge$ ContainsQuorum$(confirmAcks, C)$

97: $\wedge \; \forall \; (r, s) \in proposeAcks$: FSVerify$((\text{PROPOSERESP}, vs), r, s, height(C))$

98: $\wedge \; \forall \; (r, s) \in confirmAcks$: FSVerify$((\text{CONFIRMRESP}, proposeAcks), r, s, height(C))$

99: **procedure** Refine(vs)
100: $acks_1 \leftarrow \emptyset$; $acks_2 \leftarrow \emptyset$
101: $curVals \leftarrow curVals \cup vs$
102: $seqNum \leftarrow seqNum + 1$
103: $status \leftarrow proposing$
104: let C = HighestConf($history$)
105: **send** $\langle$PROPOSE, $curVals$, $seqNum$, $C\rangle$ **to** $replicas(C)$

106: **upon** ContainsQuorum($acks_1$, HighestConf($history$))
107: $status \leftarrow confirming$
108: let C = HighestConf($history$)
109: **send** $\langle$CONFIRM, $acks_1$, $seqNum$, $C\rangle$ **to** $replicas(C)$

110: **upon receive** $\langle$PROPOSERESP, vs, sig, $sn\rangle$ **from** replica r
111: let $sigValid$ = FSVerify((PROPOSERESP, vs), r, sig, $height$(HighestConf($history$)))
112: **if** $status = proposing \wedge sn = seqNum \wedge sigValid$ **then**
113: **if** $vs \nsubseteq curVals$ $\wedge$ VerifyInputValues($vs \setminus curVals$) **then** Refine(vs)
114: **else if** $vs = curVals$ **then** $acks_1 \leftarrow acks_1 \cup \{(r, sig)\}$

115: **upon receive** $\langle$CONFIRMRESP, sig, $sn\rangle$ **from** replica r
116: let $sigValid$ = FSVerify((CONFIRMRESP, $acks_1$), r, sig, $height$(HighestConf($history$)))
117: **if** $status = confirming \wedge sn = seqNum \wedge sigValid$ **then** $acks_2 \leftarrow acks_2 \cup \{(r, sig)\}$

118: **upon RB-deliver** $\langle$NEWHISTORY, h, $\sigma\rangle$ **from** any sender
119: **if** VerifyHistory(h, σ) $\wedge$ $history \subset h$ **then**
120: $history \leftarrow h$; $\sigma_{history} \leftarrow \sigma$
121: **if** $status \in \{proposing, confirming\}$ **then** Refine($\emptyset$)

Algorithm 5 DBLA: code for replica r

Parameters: $\mathcal{C}$, $\mathcal{L}$, C^{init}, V^{init}, VerifyHistory(h, σ), and VerifyInputValue(v, σ) (see Algorithm 4)

Global variables:

122: $\mathit{history} \subseteq \mathcal{C}$, initially $\{C^{init}\}$ ▷ local history of this process

123: $\mathit{curVals} \subseteq \mathcal{L} \times \Sigma$, initially $\{(V^{init}, \bot)\}$ ▷ known verifiable input values with proofs

124: $C^{curr} \in \mathcal{C}$, initially C^{init} ▷ current configuration

125: $C^{inst} \in \mathcal{C}$, initially C^{init} ▷ installed configuration

126: $\mathit{seqNum} \in \mathbb{Z}$, initially 0 ▷ used to match requests with responses

Auxiliary functions:

127: HighestConf, ContainsQuorum, JoinAll, VerifyInputValues (see Algorithm 4).

128: FSSign$(\mathit{message}, \mathit{timestamp})$ ▷ produces a forward-secure signature (see Section 1.1)

129: UpdateFSKey(t) ▷ updates the signing timestamp (see Section 1.1)

130: **upon receive** $\langle$PROPOSE, vs, sn, $C\rangle$ **from** client c

131: **wait for** $C = C^{inst} \lor$ HighestConf$(\mathit{history}) \not\sqsubseteq C$

132: **if** $C =$ HighestConf$(\mathit{history}) \land$ VerifyInputValues$(\mathit{vs} \setminus \mathit{curVals})$ **then**

133: $\mathit{curVals} \leftarrow \mathit{curVals} \cup \mathit{vs}$

134: let $\mathit{sig} =$ FSSign$((\text{PROPOSERESP}, \mathit{curVals}), \mathit{height}(C))$

135: **send** $\langle$PROPOSERESP, $\mathit{curVals}$, sig, $\mathit{sn}\rangle$ **to** c

136: **else** ignore the message

137: **upon receive** $\langle$CONFIRM, $\mathit{proposeAcks}$, sn, $C\rangle$ **from** client c

138: **wait for** $C = C^{inst} \lor$ HighestConf$(\mathit{history}) \not\sqsubseteq C$

139: **if** $C =$ HighestConf$(\mathit{history})$ **then**

140: let $\mathit{sig} =$ FSSign$((\text{CONFIRMRESP}, \mathit{proposeAcks}), \mathit{height}(C))$

141: **send** $\langle$CONFIRMRESP, sig, $\mathit{sn}\rangle$ **to** c

142: **else** ignore the message

▷ State transfer

143: **upon** $C^{curr} \neq$ HighestConf$(\{C \in \mathit{history} \mid r \in \mathit{replicas}(C)\})$

144: let $C^{next} =$ HighestConf$(\{C \in \mathit{history} \mid r \in \mathit{replicas}(C)\})$

145: let $S = \{C \in \mathit{history} \mid C^{curr} \sqsubseteq C \sqsubset C^{next}\}$

146: $\mathit{seqNum} \leftarrow \mathit{seqNum} + 1$

147: **for each** $C \in S$ **do**

148: **send** $\langle$UPDATEREAD, seqNum, $C\rangle$ **to** $\mathit{replicas}(C)$

149: **wait for** $(C \sqsubset C^{curr}) \lor$ (responses from any $Q \in \mathit{quorums}(C)$ with s.n. seqNum)

150: **if** $C^{curr} \sqsubset C^{next}$ **then**

151: $C^{curr} \leftarrow C^{next}$

152: **URB-Broadcast** $\langle$UPDATECOMPLETE$\rangle$ **in** C^{next}

153: **upon RB-deliver** $\langle$NEWHISTORY, h, $\sigma\rangle$ **from** any sender
154:     **if** VerifyHistory$(h, \sigma) \wedge$ *history* $\subset h$ **then**
155:         *history* $\leftarrow h$
156:         UpdateFSKey$($*height*(HighestConf(*history*)$))$

157: **upon receive** $\langle$UPDATEREAD, *sn*, $C\rangle$ **from** replica r'
158:     **wait for** $C \sqsubset$ HighestConf(*history*) ▷ only reply after UpdateFSKey
159:     **send** $\langle$UPDATEREADRESP, *curVals*, *sn*$\rangle$ **to** r'

160: **upon receive** $\langle$UPDATEREADRESP, *vs*, *sn*$\rangle$ **from** replica r'
161:     **if** VerifyInputValues(*vs* \ *curVals*) **then** *curVals* $\leftarrow$ *curVals* $\cup$ *vs*

162: **upon URB-deliver** $\langle$UPDATECOMPLETE$\rangle$ **in** C **from** quorum $Q \in$ *quorums*(C)
163:     **wait for** $C \in$ *history*
164:     **if** $C^{inst} \sqsubset C$ **then**
165:         **if** $C^{curr} \sqsubset C$ **then** $C^{curr} \leftarrow C$
166:         $C^{inst} \leftarrow C$
167:         **trigger** upcall InstalledConfig(C)
168:         **if** $r \notin$ *replicas*(C) **then** halt

A.2 Correctness proof of the DBLA implementation

A.2.1 Safety

Recall that a configuration is called *candidate* iff it appears in some verifiable history. The following lemma gathers some obvious yet very useful statements about candidate configurations.

Lemma A.1 (Candidate configurations)**.**

1. *There is a finite number of candidate configurations.*

2. *All candidate configurations are comparable with "$\sqsubseteq$".*

Proof. The total number of verifiable histories is required to be finite, and each history is finite, hence (1). All verifiable histories are required to be related by containment and all configurations within one history are required to be comparable, hence (2). □

Recall that a configuration is called *pivotal* if it is the last configuration in some verifiable history. Non-pivotal candidate configurations are called *tentative*. Intuitively, the next lemma states that in the rest of the proofs we can almost always consider only pivotal configurations. Tentative configurations are both harmless and useless.

Lemma A.2 (Tentative configurations)**.**

1. *No correct client will ever make a request to a tentative configuration.*

2. *Tentative configurations cannot be installed.*

3. *A correct process will never invoke* FSVerify *with timestamp height*(C) *for any tentative configuration* C*.*

4. *A correct replica will never broadcast any message via the uniform reliable broadcast primitive in a tentative configuration.*

Proof. Follows directly from the algorithm. Both clients and replicas only operate on configurations that were obtained by invoking the function HighestConf(h) on some verifiable configuration. □

The next lemma states that correct processes cannot "miss" any *pivotal* configurations in their local histories. This is crucial for the correctness of the state transfer protocol.

Lemma A.3. *If* $C \sqsubseteq \text{HighestConf}(h)$*, where* C *is a pivotal configuration and* h *is the local history of a correct process, then* $C \in h$*.*

Proof. Follows directly from the definition of a pivotal configuration and the requirement that all verifiable histories are related by containment (see Section 1.3.4). □

Recall that a configuration is called *superseded* iff some higher configuration is installed (see Section 1.3.3). A configuration is *installed* iff some *correct* replica has triggered the InstalledConfig upcall (Algorithm 5, line 167). For this, the correct replica must receive a quorum of UPDATECOMPLETE messages via the uniform reliable broadcast primitive (Algorithm 5, line 162).

Theorem A.4 (Dynamic Validity)**.** *Our implementation of DBLA satisfies Dynamic Validity. I.e., only a candidate configuration can be installed.*

Proof. Follows directly from the implementation. A correct replica will not install a configuration until it is in the replica's local history (Algorithm 5, line 163). □

In our algorithm, it is possible for a configuration to be installed *after* it was superseded. Imagine that a quorum of replicas broadcast UPDATECOMPLETE messages in some configuration C which is not yet installed. After that, before any replica delivers those messages, a higher configuration is installed, making C superseded. It is possible that some correct replica $r \in \mathit{replicas}(C)$ that does not yet know that a higher configuration is installed, will deliver the broadcast messages and trigger the upcall InstalledConfig(C) (Algorithm 5, line 167).

Let us call the configurations that were installed while being active (i.e., not superseded) "*properly installed*". We will use this definition to prove next few lemmas.

Lemma A.5. *The lowest properly installed configuration higher than configuration* C *is the first installed configuration higher than* C *in the real-time order.*

Proof. Let N be the lowest properly installed configuration higher than C. If some configuration higher than N were installed earlier, then N would not be properly installed (by the definition of a properly installed configuration). If some configuration between C and N were installed earlier, then N would not be the lowest. □

The following lemma stipulates that our state transfer protocol makes the superseded pivotal configurations "harmless" by leveraging a forward-secure signature scheme.

Lemma A.6 (Key update)**.** *If a pivotal configuration C is superseded, then no quorum of replicas in that configuration is capable of signing messages with timestamp $height(C)$, i.e., $\nexists Q \in quorums(C)$ s.t. $\forall r \in Q : st_r \leq height(C)$.*

Proof. Let N be the lowest properly installed configuration higher than C. Let us consider the moment when N was installed. By the algorithm, all correct replicas in some quorum $Q_N \in quorums(N)$ had to broadcast UPDATECOMPLETE messages before N was installed (Algorithm 5, line 162). Since N was not yet superseded at that moment, there was at least one correct replica $r_N \in Q_N$.

By Lemma A.3, C was in r_N's local history whenever it performed state transfer to any configuration higher than C. By the protocol, a correct replica only advances its C^{curr} variable after executing the state transfer protocol (Algorithm 5, line 151) or right before installing a configuration (Algorithm 5, line 165). Since no configurations between C and N were yet installed, r_N had to pass through C in its state transfer protocol and to receive UPDATEREADRESP messages from some quorum $Q_C \in quorums(C)$ (Algorithm 5, line 149).

Recall that correct replicas update their private keys whenever they learn about a higher configuration (Algorithm 5, line 156) and that they will only reply to message $(\text{UPDATEREAD}, sn, C)$ once C is *not* the highest configuration in their local histories (Algorithm 5, line 158). This means that all correct replicas in Q_C actually had to update their private keys before N was installed, and, hence, before C was superseded. By the quorum intersection property, this means that in each quorum in C at least one replica updated its private key to a timestamp higher than $height(C)$ and will not be capable of signing messages with timestamp $height(C)$ even if it becomes Byzantine. □

Note that in a tentative configuration there might be arbitrarily many Byzantine replicas that have not updated their private keys. This is inevitable in asynchronous system: forcing the replicas in tentative configurations to update their private keys would require solving consensus. This does not affect correct processes because, as shown in Lemma A.2, tentative configurations are harmless. However, it is important to remember this when designing new dynamic protocols.

The following lemma implies that the state is correctly transferred between configurations.

Lemma A.7 (State transfer correctness)**.**
If $\sigma = (\mathit{vs}, h, \sigma_h, \mathit{proposeAcks}, \mathit{confirmAcks})$ *is a valid proof for* v*, then for each active installed configuration* D *such that* $\text{HighestConf}(h) \sqsubset D$*, there is a quorum* $Q_D \in \mathit{quorums}(D)$ *such that for each correct replica* $r \in Q_D$*:* $\mathit{vs} \subseteq \mathit{curVals}_r$*.*

Proof. Let $C = \text{HighestConf}(h)$. We proceed by induction on the sequence of all properly installed configurations higher than C. Let us denote this sequence by $\widetilde{\mathcal{C}}$. By the definition of a properly installed configuration, these are precisely the configurations that we consider in the statement of the lemma.

Let N be the lowest configuration in $\widetilde{\mathcal{C}}$. Let $Q_C \in \mathit{quorums}(C)$ be a quorum of replicas whose signatures are in *proposeAcks*. Consider the moment of installation of N. There must be a quorum $Q_N \in \mathit{quorums}(N)$ in which all correct replicas broadcast $\langle \text{UpdateComplete}, N \rangle$ before the moment of installation. For each correct replica $r_N \in Q_N$, r_N passed with its state transfer protocol through configuration C and received UpdateReadResp messages from some quorum of replicas in C. Note that at that moment configuration C was not yet superseded. By the quorum intersection property, there is at least one correct replica $r_C \in Q_C$ that sent an UpdateReadResp message to r_N (Algorithm 5, line 159). Since r_C will send the UpdateReadResp message only after updating its private keys (Algorithm 5, line 158), it had to sign $(\text{ProposeResp}, \mathit{vs})$ (Algorithm 5, line 134) before sending reply to r_N, which means that the UpdateReadResp message from r_C to r_N must have contained a set of values that includes all values from *vs*. This proves the base case of the

induction.

Let us consider any configuration $D \in \widetilde{\mathcal{C}}$ such that $N \sqsubset D$. Let M be the highest configuration in $\widetilde{\mathcal{C}}$ such that $N \sqsubseteq M \sqsubset D$ (in other words, the closest to D in $\widetilde{\mathcal{C}}$). Assume that the statement holds for M, i.e., while M was active, there were a quorum $Q_M \in \mathit{quorums}(M)$ such that for each correct replica $r_M \in Q_M$: $\mathit{vs} \subseteq \mathit{curVals}_{r_M}$. Similarly to the base case, let us consider a quorum $Q_D \in \mathit{quorums}(D)$ such that every correct replica in Q_D reliably broadcast $\langle \text{UpdateComplete}, D \rangle$ before D was installed. For each correct replica $r_D \in Q_D$, by the quorum intersection property, there is at least one correct replica in Q_M that sent an UpdateReadResp message to r_D. This replica attached its *curVals* to the message, which contained *vs*. This proves the induction step and completes the proof. □

The next lemma states that if two output values were produced in the same configuration, they are comparable. In a static system it could be proven by simply referring to the quorum intersection property. In a dynamic Byzantine system, however, to use the quorum intersection, we need to prove that the configuration was active during the whole period when the clients were exchanging data with the replicas. In other words, we need to prove that the "slow reader" attack is impossible. Luckily, we have the second stage of our algorithm designed for this sole purpose.

Lemma A.8 (BLA-Comparability in one configuration)**.**
If $\sigma_1 = (\mathit{vs}_1, h_1, \sigma_{h1}, \mathit{proposeAcks}_1, \mathit{confirmAcks}_1)$ *is a valid proof for output value* v_1*,*
and $\sigma_2 = (\mathit{vs}_2, h_2, \sigma_{h2}, \mathit{proposeAcks}_2, \mathit{confirmAcks}_2)$ *is a valid proof for output value* v_2*,*
and $\text{HighestConf}(h_1) = \text{HighestConf}(h_2)$*, then* v_1 *and* v_2 *are comparable.*

Proof. Let $C = \text{HighestConf}(h_1) = \text{HighestConf}(h_2)$. By definition, the fact that σ is a valid proof for v implies that $\text{VerifyOutputValue}(v, \sigma) = \mathit{true}$ (Algorithm 4, line 91). By the implementation, h_1 and h_2 are verifiable histories (Algorithm 4, line 95). Therefore, C is a pivotal configuration.

The set $\mathit{confirmAcks}_1$ contains signatures from a quorum of replicas of configuration C, with timestamp $\mathit{height}(C)$. Each of these signatures had to be pro-

duced after each of the signatures in $proposeAcks_1$ because they sign the message $(\text{ConfirmResp}, proposeAcks_1)$ (Algorithm 5, line 140). Combining this with the statement of Lemma A.6 (Key Update), it follows that at the moment when the last signature in the set $proposeAcks_1$ was created, the configuration C was active (otherwise it would be impossible to gather $confirmAcks_1$). We can apply the same argument to the sets $proposeAcks_2$ and $confirmAcks_2$.

It follows that there are quorums $Q_1, Q_2 \in quorums(C)$ and a moment in time t such that: (1) C is not superseded at time t, (2) all *correct* replicas in Q_1 signed message $(\text{ProposeResp}, vs_1)$ before t, and (3) all *correct* replica in Q_2 signed message $(\text{ProposeResp}, vs_2)$ before t. Since C is not superseded at time t, there must be a correct replica in $Q_1 \cap Q_2$ (due to quorum intersection), which signed both $(\text{ProposeResp}, vs_1)$ and $(\text{ProposeResp}, vs_2)$ (Algorithm 5, line 134). Since correct replicas only sign ProposeResp messages with comparable sets of values[8], vs_1 and vs_2 are comparable, i.e., either $vs_1 \subseteq vs_2$ or $vs_2 \subset vs_1$. Hence, $v_1 = \text{JoinAll}(vs_1)$ and $v_2 = \text{JoinAll}(vs_2)$ are comparable. □

Finally, let us combine the two previous lemmas to prove the BLA-Comparability property of our DBLA implementation.

Theorem A.9 (BLA-Comparability)**.** *Our implementation of DBLA satisfies the BLA-Comparability property. That is, all verifiable output values are comparable.*

Proof. Let $\sigma_1 = (vs_1, h_1, \sigma_{h1}, proposeAcks_1, confirmAcks_1)$ be a valid proof for output value v_1, and $\sigma_2 = (vs_2, h_2, \sigma_{h2}, proposeAcks_2, confirmAcks_2)$ be a valid proof for output value v_2. Also, let $C_1 = \text{HighestConf}(h_1)$ and $C_2 = \text{HighestConf}(h_2)$. Since h_1 and h_2 are verifiable histories (Algorithm 4, line 95), both C_1 and C_2 are pivotal by definition.

If $C_1 = C_2$, v_1 and v_2 are comparable by Lemma A.8.

Consider the case when $C_1 \neq C_2$. Without loss of generality, assume that $C_1 \sqsubset C_2$. Let $Q_1 \in quorums(C_2)$ be a quorum of replicas whose signatures are in $proposeAcks_2$. Let t be the moment when first correct replica signed $\langle \text{ProposeResp}, vs_2 \rangle$. Correct replicas only start processing user requests

[8]Indeed, set *curVals* at each correct replica can only grow, and the replicas only sign messages with the same set of verifiable input values as *curVals* (see Algorithm 5, lines 133–134).

in a configuration when this configuration is installed (Algorithm 5, line 131). Therefore, by Lemma A.7, at time t there was a quorum of replicas $Q_2 \in$ *quorums*(C_2) such that for every correct replica in Q_2: $vs_1 \subseteq curVals$. By the quorum intersection property, there must be at least one correct replica in $Q_1 \cap Q_2$. Hence, $vs_1 \subseteq vs_2$ and $\text{JoinAll}(vs_1) \sqsubseteq \text{JoinAll}(vs_2)$. □

Theorem A.10 (DBLA safety)**.** *Our implementation satisfies the safety properties of DBLA: BLA-Validity, BLA-Verifiability, BLA-Inclusion, BLA-Comparability, and Dynamic Validity.*

Proof.

- BLA-Validity follows directly from the implementation: a correct client collects verifiable input values and joins them before returning from Propose (Algorithm 4, line 88);
- BLA-Verifiability follows directly from how correct replicas form and check the proofs for output values (Algorithm 4, lines 87 and 93–98);
- BLA-Inclusion follows from the fact that the set *curVals* of a correct client only grows (Algorithm 4, line 101);
- BLA-Comparability follows from Theorem A.9;
- Finally, Dynamic Validity follows from Theorem A.4.

□

A.2.2 Liveness

Lemma A.11 (History Convergence)**.** *Local histories of all correct processes will eventually become identical.*

Proof. Let p and q be any two forever-correct processes[9]. Suppose, for contradiction, that the local histories of p and q have diverged at some point and will never converge again. Recall that correct processes only adopt verifiable

[9] If either p or q eventually halts or becomes Byzantine, their local histories are not required to converge.

histories, and that the total number of verifiable histories is required to be finite. Therefore, there is some history h_p, which is the the largest history ever adopted by p, and some history h_q which is the the largest history ever adopted by q. Since all verifiable histories are required to be related by containment and we assume that $h_p \neq h_q$, one of them must be a subset of the other. Without loss of generality, suppose that $h_p \subset h_q$. Since q had to deliver h_q through reliable broadcast (unless h_q is the initial history) and q remains correct forever, p will eventually deliver h_q as well, and will adopt it. Hence, h_p is not the largest history ever adopted by p. A contradiction. □

Next, an important definition, which we will use throughout the rest of the proofs, is introduced.

Definition A.12 (Maximal installed configuration)**.** *In a given infinite execution, a* maximal installed configuration *is a configuration that eventually becomes installed and never becomes superseded.*

Lemma A.13 (C_{max} existence)**.** *In any infinite execution there is a unique maximal installed configuration.*

Proof. By Lemma A.1 (Candidate configurations) and Theorem A.4 (Dynamic Validity), the total number of installed configurations is finite and they are comparable. Hence, we can choose a unique maximum among them, which is never superseded by definition. □

Let us denote the (unique) maximal installed configuration by C_{max}.

Lemma A.14 (C_{max} installation)**.** *The maximal installed configuration will eventually be installed by all correct replicas.*

Proof. Since C_{max} is installed, by definition, at some point some correct replica has triggered upcall InstalledConfig(C_{max}) (Algorithm 5, line 167). This, in turn, means that this replica delivered a quorum of UpdateComplete messages via the *uniform reliable broadcast* in C_{max} when it was correct. Therefore, even if this replica later becomes Byzantine, by definition of the uniform reliable broadcast, either C_{max} will become superseded (which is impossible), or every correct replica will eventually deliver the same set of UpdateComplete messages and install C_{max}. □

Lemma A.15 (State transfer progress)**.** *State transfer (Algorithm 5, lines 143–152) executed by a forever-correct replica always terminates.*

Proof. Let r be a correct replica executing state transfer. By Lemma A.1, the total number of candidate configurations is finite. Therefore, it is enough to prove that there is no such configuration that r will wait for replies from a quorum of that configuration indefinitely (Algorithm 5, line 149). Suppose, for contradiction, that there is such configuration C.

If $C \sqsubset C_{max}$, then, by Lemma A.14, r will eventually install C_{max}, and C^{curr} will become not lower than C_{max} (Algorithm 5, line 165). Hence, r will terminate from waiting through the first condition ($C \sqsubset C^{curr}$). A contradiction.

Otherwise, if $C_{max} \sqsubseteq C$, then, by the definition of C_{max}, C will never be superseded. Since r remains correct forever, by Lemma A.11 (History Convergence), every correct replica will eventually have C in its local history. Since reliable links between processes are assumed (see Section 2.1), every correct replica in $replicas(C)$ will eventually receive r's UPDATEREAD message and will send a reply, which r will receive (Algorithm 5, line 159). Hence, the waiting on line 149 of Algorithm 5 will eventually terminate through the second condition (r will receive responses from some $Q \in quorums(C)$ with the correct sequence number). A contradiction. □

Intuitively, the following lemma states that C_{max} is, in some sense, the "final" configuration. After some point every correct process will operate exclusively on C_{max}. No correct process will know about any configuration higher than C_{max} or "care" about any configuration lower than C_{max}.

Lemma A.16. *C_{max} will eventually become the highest configuration in the local history of each correct process.*

Proof. By Lemma A.11 (History Convergence), the local histories of all correct processes will eventually converge to the same history h. Let $D =$ HighestConf(h). Since C_{max} is installed and never superseded, it cannot be higher than D (at least one correct replica will always have C_{max} in its local history).

Suppose, for contradiction, that $C_{max} \sqsubset D$. In this case, D is never superseded, which means that there is a quorum $Q_D \in quorums(D)$ that consists

entirely of forever-correct processes. By Lemma A.11 (History Convergence), all replicas in Q_D will eventually have D in their local histories and will try to perform state transfer to it. By Lemma A.15, they will eventually succeed and install D—a contradiction with the definition of C_{max}. □

Theorem A.17 (BLA-Liveness)**.** *Our implementation of DBLA satisfies the BLA-Liveness property: if the total number of verifiable input values is finite, every call to* Propose(v, σ) *by a forever-correct process eventually returns.*

Proof. Let p be a forever-correct client that invoked Propose(v, σ). By Lemma A.16, C_{max} will eventually become the highest configuration in the local history of p. If the client's request will not terminate by the time it learns about C_{max}, the client will call Refine$(\emptyset)$ after it (Algorithm 4, line 121). By Lemma A.14, C_{max} will eventually be installed by all correct replicas. Since it will never be superseded, there will be a quorum of forever-correct replicas. Thus, every round of messages from the client will eventually be responded to by a quorum of correct replicas.

Since the total number of verifiable input values is finite, the client will call Refine only a finite number of times (Algorithm 4, line 113). After the last call to Refine, the client will inevitably receive acknowledgments from a quorum of replicas, and will proceed to sending CONFIRM messages (Algorithm 4, line 109). Again, since there is an available quorum of correct replicas that installed C_{max}, the client will eventually receive enough acknowledgments and will complete the operation (Algorithm 4, line 86). □

Theorem A.18 (DBLA liveness)**.** *Our implementation satisfies the liveness properties of DBLA: BLA-Liveness, Dynamic Liveness, and Installation Liveness.*

Proof. BLA-Liveness follows from Theorem A.17. Dynamic Liveness and Installation Liveness follow directly from Lemmas A.16 and A.14 respectively. □

A.3 Possible optimizations for the DBLA implementation

Here, a few possible directions for optimization are discussed. Applying these optimizations can significantly reduce the communication cost of the protocol. These optimizations were not applied in the original pseudocode as it would significantly complicate the protocol and make it harder to understand.

First, the proofs in the protocol include the full local history of a process. Moreover, this history comes with its own proof, which also usually contains a history, and so on. If implemented naively, the size of one proof in bytes will be at least quadratic with respect to the number of distinct candidate configurations, which is completely unnecessary. The first observation is that these histories will be related by containment. So, in fact, they can be compressed just to the size of the largest one, which is linear. But it is possible to go further and say that, in fact, in a practical implementation, the processes almost never should actually send full histories to each other because every process maintains its local history and all histories with proofs are already disseminated via the reliable broadcast primitive. When one process wants to send some history to some other process, it can just send a cryptographic hash of this history. The other process can check if it already has this history and, if not, ask the sender to only send the missing parts, instead of the *whole* history.

Second, a naive implementation of the DBLA protocol would send ever-growing sets of verifiable input values around, which is, just as with histories, completely unnecessary. The processes should just limit themselves to sending diffs between what they know and what they think the recipient knows.

Third, almost every proof in the systems contains signatures from a quorum of replicas. This adds another linear factor to the communication cost. However, it can be significantly reduced by the use of forward-secure *multi-signatures*, such as Pixel [31], which was designed for similar purposes.

Finally, a suboptimal implementation of lattice agreement is used as the foundation for the DBLA protocol. Perhaps, adapting a more efficient crash fault-tolerant asynchronous solution [72] could be beneficial.

A.4 Max Register

The presented methodology of constructing dynamic and reconfigurable objects is not limited to lattice agreement. In this section, an implementation of a dynamic version of an atomic Byzantine fault-tolerant Max-Register is presented. One can then apply the technique presented in Section 1.5 to create a reconfigurable Max-Register.

An atomic (a.k.a. linearizable) multi-writer multi-reader Byzantine Max-Register is a distributed object that has two operations: Read() and Write(v, σ) and must be parametrized by a boolean function VerifyInputValue(v, σ). As before, a certificate σ is said to be a *valid certificate for input value* v iff VerifyInputValue(v, σ) = *true*, and a value v is said to be a *verifiable input value* iff some process knows σ such that VerifyInputValue(v, σ) = *true*. It is assumed that correct clients invoke Write(v, σ) only if VerifyInputValue(v, σ) = *true*. However, no assumptions are made on the number of verifiable input values for this abstraction (i.e., it can be infinite).

The Max-Register object satisfies the following three properties:

- *MR-Validity*: if Read() returns value v to a correct process, then v is verifiable input value;
- *MR-Atomicity*: if some correct process p completed Write(v, σ) or received v from Read() strictly before some correct process q invoked Read(), then the value returned to q must be greater than or equal to v;
- *MR-Liveness*: every call to Read() and Write(v, σ) by a forever-correct process eventually returns.

For simplicity, unlike Byzantine Lattice Agreement, Max-Register does not provide the VerifyOutputValue(v, σ) function.

A.4.1 Dynamic Max-Register implementation

In this section, the implementation of the *dynamic* version of the Max-Register abstraction (Dynamic Max-Register or DMR for short) is presented. Overall, the "application" part of the implementation is very similar to the classical ABD

Algorithm 6 Dynamic Max-Register: code for client p

Parameters:

169: Lattice of configurations $\mathcal{C}$ and the initial configuration C^{init}

170: Set of values $\mathbb{V}$ and the initial value V^{init}

171: Boolean functions VerifyHistory(h, σ) and VerifyInputValue(v, σ)

Global variables:

172: *history* $\subseteq \mathcal{C}$, initially $\{C^{init}\}$ ▷ local history of this process

173: *seqNum* $\in \mathbb{Z}$, initially 0 ▷ used to match requests with responses

Auxiliary functions: HighestConf(h), FSVerify (see Section 1.1)

174: **operation** Read()

175: **repeat**

176: let $(readOk, (v, \sigma)) =$ Get()

177: let $success =$ **if** $readOk$ **then** Set(v, σ) **else** *false*

178: **until** *success*

179: **return** v

180: **operation** Write(v, σ)

181: **repeat** let $success =$ Set(v, σ)

182: **until** *success*

183: **operation** UpdateHistory(h, σ)

184: **RB-Broadcast** $\langle$NewHistory, h, $\sigma\rangle$

185: **procedure** Set(v, σ)

186: $seqNum \leftarrow seqNum + 1$

187: let $C =$ HighestConf(*history*)

188: **send** $\langle$Set, v, *seqNum*, $C\rangle$ **to** *replicas*(C)

189: **wait for** (HighestConf(*history*) $\neq C$) $\vee$ (replies from $Q \in quorums(C)$ with valid signatures)

190: **return** HighestConf(*history*) $\neq C$

191: **procedure** Get()

192: $seqNum \leftarrow seqNum + 1$

193: let $C =$ HighestConf(*history*)

194: **send** $\langle$Get, *seqNum*, $C\rangle$ **to** *replicas*(C)

195: **wait for** (HighestConf(*history*) $\neq C$) $\vee$ (replies from $Q \in quorums(C)$)

196: **if** HighestConf(*history*) $\neq C$ **then return** (*false*, $\perp$)

197: **else return** (*true*, maximal verifiable input value among received)

198: **upon RB-deliver** $\langle$NewHistory, h, $\sigma\rangle$ **from** any sender

199: **if** VerifyHistory$(h, \sigma) \wedge history \subset h$ **then** $history \leftarrow h$

Algorithm 7 Dynamic Max-Register: code for replica r

Parameters: $\mathcal{C}$, C^{init}, $\mathbb{V}$, V^{init}, VerifyHistory(h, σ), and VerifyInputValue(v, σ) (See Algorithm 6)

Global variables:

200: $history \subseteq \mathcal{C}$, initially $\{C^{init}\}$ ▷ local history of this process
201: $v_{curr} \in \mathbb{V}$, initially V^{init}
202: $\sigma_{curr} \in \Sigma$, initially σ_{init}
203: $C^{curr} \in \mathcal{C}$, initially C^{init} ▷ current configuration
204: $C^{inst} \in \mathcal{C}$, initially C^{init} ▷ installed configuration

Auxiliary functions: HighestConf(h), FSSign(m, t), UpdateFSKey(t) (see Section 1.1)

205: **upon receive** $\langle$GET, *sn*, $C\rangle$ **from** client c
206: **wait for** $C = C^{inst} \lor$ HighestConf$(history) \not\sqsubseteq C$
207: **if** $C =$ HighestConf$(history)$ **then**
208: **send** $\langle$GETRESP, v_{curr}, σ_{curr}, *sn*$\rangle$ **to** c
209: **else** ignore the message

210: **upon receive** $\langle$SET, v, σ, *sn*, $C\rangle$ **from** client c
211: **wait for** $C = C^{inst} \lor$ HighestConf$(history) \not\sqsubseteq C$
212: **if** $C =$ HighestConf$(history) \land$ VerifyInputValue(v, σ) **then**
213: **if** $v > v_{curr}$ **then** $(v_{curr}, \sigma_{curr}) \leftarrow (v, \sigma)$
214: **send** $\langle$SETRESP, FSSign$((c, sn), height(C))$, *sn*$\rangle$ **to** c
215: **else** ignore the message

▷ State transfer
216: **upon** $C^{curr} \neq$ HighestConf$(\{C \in history \mid r \in replicas(C)\})$
217: Same as for DBLA (Algorithm 5, lines 143–152)

218: **upon receive** $\langle$UPDATEREAD, C, *sn*$\rangle$ **from** replica r'
219: **wait for** $C \sqsubset$ HighestConf$(history)$
220: **send** $\langle$UPDATEREADRESP, v_{curr}, σ_{curr}, *sn*$\rangle$ **to** r'

221: **upon receive** $\langle$UPDATEREADRESP, v, σ, *sn*$\rangle$ **from** replica r'
222: **if** VerifyInputValue$(v, \sigma) \land v > v_{curr}$ **then** $(v_{curr}, \sigma_{curr}) \leftarrow (v, \sigma)$

223: **upon RB-deliver** $\langle$NEWHISTORY, h, $\sigma\rangle$ **from** any sender
224: Same as for DBLA (Algorithm 5, lines 153–156)

225: **upon URB-deliver** $\langle$UPDATECOMPLETE$\rangle$ **in** C **from** quorum $Q \in quorums(C)$
226: Same as for DBLA (Algorithm 5, lines 162–168)

algorithm [14], and the "dynamic" part of the implementation is almost the same as in DBLA.

Client implementation. From the client's perspective, the two main procedures are Get() and Set(v, σ) (not to be confused with the Read and Write operations). Set(v, σ) (Algorithm 6, lines 185–190) is used to store the value on a quorum of replicas of the most recent configuration. It returns *true* iff it manages to receive signed acknowledgments from a quorum of some configuration. Forward-secure signatures are used to prevent the "I still work here" attack. Since Set does not try to read any information from the replicas, it is not susceptible to the "slow reader" attack. Get() (lines 191–197) is very similar to Set$(\ldots)$ and is used to request information from a quorum of replicas of the most recent configuration. Since the VerifyOutputValue$(\ldots)$ function is not provided, the replies from replicas are not signed (Algorithm 7, line 208). Therefore, Get() is susceptible to both the "I still work here" and the "slow reader" attack when used by itself. Later in this section, it is discussed how the invocation of Set$(\ldots)$ right after Get() (Algorithm 6, line 177) prevents these issues.

Operation Write(v, σ) (lines 180–182) is used by correct clients to store values in the register. It simply performs repeated calls to Set(v, σ) until some call succeeds to reach a quorum of replicas. Retries are safe because, as in lattice agreement, write requests to a max-register are idempotent. Since the total number of verifiable histories is assumed to be finite, only a finite number of retries is possible.

Operation Read() (lines 174–179) is used to request the current value from the register, and it consists of repeated calls to both Get() and Set$(\ldots)$. The call to Get() is simply used to query information from the replicas. The call to Set$(\ldots)$ is usually called "the write-back phase" and serves two purposes here:

- It is used instead of the "confirming" phase to prevent the "I still work here" and the "slow-reader" attacks. Indeed, if the configuration was superseded during the execution of Get(), Set$(\ldots)$ will not succeed because it will not be able to gather a quorum of signed replies in the same configuration;

- Also, it is used to order the calls to Read$()$ and to guarantee the MR-Atomicity property. Intuitively, if some correct process successfully completed Set(v, σ) strictly before some other correct process invoked Get$()$, the later process will receive a value that is not smaller than v (unless the "slow reader" attack happens).

Replica implementation. The replica implementation (Algorithm 7) essentially follows the implementation of DBLA (Algorithm 5), except that the replica handles client requests specific to Max-Register (Algorithm 7, lines 205–215). The only other difference is that in handling the UPDATEREAD and UPDATEREADRESP messages (Algorithm 7, lines 218–222), the replicas exchange v_{curr} and σ_{curr} instead of *curVals*, as in DBLA.

A.4.2 Proof of correctness

Since the Dynamic Max-Register implementation uses the same state transfer protocol as DBLA, most proofs from Section A.2 that apply to DBLA, also apply to DMR (with some minor adaptations). To avoid repetition, here, only the statements of such theorems are provided, without proofs. Then, several theorems specific to DMR are introduced.

DMR safety.

Lemma A.19 (Candidate configurations)**.**

1. *Each candidate configuration is present in some verifiable history.*

2. *There is a finite number of candidate configurations.*

3. *All candidate configurations are comparable with "$\sqsubseteq$".*

Lemma A.20 (Tentative configurations)**.**

1. *No correct client will ever make a request to a tentative configuration.*

2. *Tentative configurations cannot be installed.*

3. *A correct process will never invoke* FSVerify *with timestamp height*(C) *for any tentative configuration* C.

4. *A correct replica will never broadcast any message via the uniform reliable broadcast primitive in a tentative configuration.*

Lemma A.21. *If* $C \sqsubseteq \text{HighestConf}(h)$*, where* C *is a pivotal configuration and* h *is the local history of a correct process, then* $C \in h$.

Theorem A.22 (Dynamic Validity)**.** *The implementation of DMR satisfies Dynamic Validity. I.e., only a candidate configuration can be installed.*

Lemma A.23 (Key update)**.** *If a pivotal configuration* C *is superseded, then there is no quorum of replicas in that configuration capable of signing messages with timestamp height*(C)*, i.e.,* $\nexists Q \in$ *quorums*(C) *s.t.* $\forall r \in Q : st_r \leq$ *height*(C).

A correct client is said to *complete its operation in configuration* C iff at the moment when the client completes its operation, the highest configuration in its local history is C.

Lemma A.24 (State transfer correctness)**.**
If some correct process completed $\text{Write}(v, \sigma)$ *in* C *or received* v *from* $\text{Read}()$ *operation completed in* C*, then for each active installed configuration* D *such that* $C \sqsubset D$*, there is a quorum* $Q_D \in$ *quorums*(D) *such that for each correct replica in* Q_D*:* $v_{curr} \geq v$.

The following lemma is the first lemma specific to DMR.

Lemma A.25 (MR-Atomicity in one configuration)**.**
If some correct process p *completed* $\text{Write}(v, \sigma)$ *in* C *or received* v *from* $\text{Read}()$ *operation completed in* C *strictly before some correct process* q *invoked* $\text{Read}()$ *and* q *completed its operation in* C*, then the value returned to* q *is greater than or equal to* v.

Proof. Recall that $\text{Read}()$ operation consists of repeated calls to two procedures: $\text{Get}()$ and $\text{Set}(\dots)$. If process q successfully completed $\text{Set}(\dots)$ in configuration

C, then, by the use of forward-secure signatures, configuration C was active during the execution of Get() that preceded the call to Set. This also means that configuration C was active during the execution of Set(v, σ) by process p, since it was before process q started executing its request. By the quorum intersection property, process q must have received v or a greater value from at least one correct replica. □

Theorem A.26 (MR-Atomicity)**.** *The implementation of DMR satisfies the MR-Atomicity property. If some correct process* p *completed* Write(v, σ) *or received* v *from* Read() *strictly before some correct process* q *invoked* Read()*, then the value returned to* q *must be greater than or equal to* v

Proof. Let C (resp., D) be the highest configuration in p's (resp., q's) local history when it completed its request. Also, let v (resp., u) be the value that p (resp., q) passed to the last call to Set$(\ldots)$ (note that both Read() and Write$(\ldots)$ call Set$(\ldots)$).

If $C = D$, then $u \geq v$ by Lemma A.25.

Suppose, for contradiction, that $D \sqsubset C$. Since correct replicas do not reply to user requests in a configuration until this configuration is installed (Algorithm 7, line 206), configuration C had to be installed before p completed its request. By Lemma A.23 (Key Update), this would mean that q would not be able to complete Set$(\ldots)$ in D—a contradiction.

The remaining case is when $C \sqsubset D$. In this case, by Lemma A.24, the quorum intersection property, and the use of forward-secure signatures in Set$(\ldots)$, q received v or a greater value from at least one correct replica during the execution of Get(). Therefore, in this case u is also greater than or equal to v. □

Theorem A.27 (DMR safety)**.** *The implementation satisfies the safety properties of DMR: MR-Validity, MR-Atomicity, and Dynamic Validity.*

Proof. MR-Validity follows directly from the implementation: correct clients only return verifiable input values from Get() (Algorithm 6, line 197). MR-Atomicity follows directly from Theorem A.26. Dynamic Validity follows from Theorem A.22. □

DMR liveness.

Lemma A.28 (History Convergence)**.** *Local histories of all correct processes will eventually become identical.*

Recall that the *maximal installed configuration* is the highest installed configuration and is denoted by C_{max} (see Definition A.12 and Lemma A.13 in Section A.2).

Lemma A.29 (C_{max} installation)**.** *The maximal installed configuration will eventually be installed by all correct replicas.*

Lemma A.30 (State transfer progress)**.** *State transfer executed by a forever-correct replica always terminates.*

Lemma A.31. *C_{max} will eventually become the highest configuration in the local history of each correct process.*

Theorem A.32 (MR-Liveness)**.** *The implementation of DMR satisfies the MR-Liveness property: every call to* Read() *and* Write(v, σ) *by a forever-correct process eventually returns.*

Proof. Let p be a forever-correct client that invoked Read() or Write$(\ldots)$. By Lemma A.16, C_{max} will eventually become the highest configuration in the local history of p. If the client's request does not terminate by the time the client learns about C_{max}, the client will restart the request in C_{max}. Since C_{max} will eventually be installed by all correct replicas and will never be superseded, there will be a quorum of forever-correct replicas, and p will be able to complete its request there. □

Theorem A.33 (DMR liveness)**.** *The implementation satisfies the liveness properties of DMR: MR-Liveness, Dynamic Liveness, and Installation Liveness.*

Proof. MR-Liveness follows from Theorem A.32. Dynamic Liveness and Installation Liveness follow directly from Lemmas A.31 and A.29 respectively. □