Asynchronous Reconfiguration with Byzantine Failures

Petr Kuznetsov
LTCI, Télécom Paris, Institut Polytechnique Paris, France
petr.kuznetsov@telecom-paris.fr

Andrei Tonkikh
National Research University Higher School of Economics, Saint-Petersburg, Russia
andrei.tonkikh@gmail.com

Abstract

Replicated services are inherently vulnerable to failures and security breaches. In a long-running system, it is, therefore, indispensable to maintain a *reconfiguration* mechanism that would replace faulty replicas with correct ones. An important challenge is to enable reconfiguration without affecting the availability and consistency of the replicated data: the clients should be able to get correct service even when the set of service replicas is being updated.

In this paper, we address the problem of reconfiguration in the presence of Byzantine failures: faulty replicas or clients may arbitrarily deviate from their expected behavior. We describe a generic technique for building *asynchronous* and *Byzantine fault-tolerant* reconfigurable objects: clients can manipulate the object data and issue reconfiguration calls without reaching consensus on the current configuration. With the help of forward-secure digital signatures, our solution makes sure that superseded and possibly compromised configurations are harmless, that slow clients cannot be fooled into reading stale data, and that Byzantine clients cannot cause a denial of service by flooding the system with reconfiguration requests. Our approach is modular and based on *dynamic lattice agreement* abstraction, and we discuss how to extend it to enable Byzantine fault-tolerant implementations of a large class of reconfigurable replicated services.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Reconfiguration, Asynchronous Models, Byzantine Faults

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.27

Related Version A full version of the paper is available at [27], `https://arxiv.org/abs/2005.13499`.

1 Introduction

Replication and quorums. Replication is a natural way to ensure availability of shared data in the presence of failures. A collection of *replicas*, each holding a version of the data, ensure that the *clients* get a desired service, even when some replicas become unavailable or hacked by a malicious adversary. Consistency of the provided service requires the replicas to *synchronize*: intuitively, every client should be able to operate on the most "up-to-date" data, regardless of the set of replicas it can reach.

It always makes sense to assume as little as possible about the environment in which a system we design is expected to run. For example, *asynchronous* distributed systems do not rely on timing assumptions, which makes them extremely robust with respect to communication disruptions and computational delays. It is, however, notoriously difficult and sometimes even impossible to make such systems *fault-tolerant*. The folklore CAP theorem [12, 21] states that no replicated service can combine consistency, availability, and partition-tolerance. In particular, no consistent and available read-write storage can be implemented in the presence of partitions: clients in one partition are unable to keep track of the updates taking place in another one.

© Petr Kuznetsov and Andrei Tonkikh;
licensed under Creative Commons License CC-BY

Therefore, fault-tolerant storage systems tend to assume that partitions are excluded, e.g., by requiring a majority of replicas to be *correct* [6]. More generally, one can assume a *quorum system*, e.g., a set of subsets of replicas satisfying the intersection and availability properties [20]. Every (read or write) request from a client should be *accepted* by a quorum of replicas. As every two quorums have at least one replica in common, intuitively, no client can miss previously written data.

Of course, failures of replicas may jeopardize the underlying quorum system. In particular, we may find ourselves in a system in which no quorum is available and, thus, no operation may be able to terminate. Even worse, if the replicas are subject to Byzantine failures, we may not be able to guarantee the very correctness of read values.

Asynchronous reconfiguration. To anticipate such scenarios in a long run, we must maintain a *reconfiguration* mechanism that enables replacing compromised replicas with correct ones and update the corresponding quorum assumptions. A challenge here is to find an asynchronous implementation of reconfiguration in a system where both clients and replicas are subject to Byzantine failures that can be manifested by arbitrary and even malicious behavior. In the world of selfishly driven blockchain users, a reconfiguration mechanism must be prepared to this.

Recently, a number of reconfigurable systems were proposed for asynchronous *crash*-fault environments [2, 19, 23, 3, 34, 26] that were originally applied to (read-write) storage systems [2, 19, 3], and then extended to max-registers [23, 34] and more general *lattice* data type [26].

These proposals tend to ensure that the clients reach a form of "loose" agreement on the currently active configurations, which can be naturally expressed via the *lattice agreement* abstraction [8, 16]. We allow clients to (temporarily) live in different worlds, as long as these worlds are properly ordered. For example, we may represent a configuration as a set of *updates* (additions and removals of replicas) and require that all installed configurations should be related by containment. A configuration becomes *stale* as soon as it is subsumed by a new one representing a proper superset of updates.

Challenges of Byzantine fault-tolerant reconfiguration. In this paper, we focus on *Byzantine fault-tolerant* reconfiguration mechanism. We had to address here several challenges, specific to dynamic systems with Byzantine faults, which make it hard to benefit from existing crash fault-tolerant reconfigurable solutions.

First, when we build a system out of lower-level components, we need to make sure that the outputs provided by these components are "authentic". Whenever a (potentially Byzantine) process claims to have obtained a *value* v (e.g., a new configuration estimate) from an underlying object (e.g., Lattice Agreement), it should also provide a *proof* σ that can be independently verified by every correct process. The proof typically consists of digital signatures provided by a quorum of replicas of some configuration. We abstract this requirement out by equipping the object with a function VerifyOutputValue that returns a boolean value, provided v and σ. When invoked by a correct process, the function returns *true* if and only if v has indeed been produced by the object. When "chaining" the objects, i.e., adopting the output v provided by an object A as an input for another object B, which is the typical scenario in our system, a correct process invokes $A.\text{VerifyOutputValue}(v, \sigma)$, where σ is the proof associated with v by the implementation of A. This way, only values actually produced by A can be used as inputs to B.

Second, we face the **"I still work here" attack** [1]. It is possible that a client that did not log into the system for a long time tries to access a *stale*, outdated configuration in which some quorum is entirely compromised by the Byzantine adversary. The client can therefore be provided with an inconsistent view on the shared data. Thus, before accepting a new configuration, we need to make sure that the stale ones are no longer capable of processing data requests from the clients. We address this issue via the use of a forward-secure signature scheme [10]. Intuitively, every replica is provided with a distinct private key associated to each configuration. Before a configuration is replaced with a newer one, at least a quorum of its replicas are asked to destroy their private keys. Therefore, even if the replicas are to become Byzantine in the future, they will not be able to provide slow clients with inconsistent values. The stale 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* stale configurations remove their private keys as it would require solving consensus [17]. However, as we show in this paper, 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 "proofs" for output values.

Finally, there is a subtle, and quite interesting **"slow reader" attack**. Suppose that a client accesses *almost all* replicas in a quorum of the current configuration each holding a stale state, as the only correct replica in the quorum that has the up-to-date state has not yet responded. The client then falls asleep, meanwhile, the configuration is superseded by a new one. As we do not make any assumptions about the correctness of replicas in stale configurations, the replica that has not yet responded can be compromised. Moreover, due to asynchrony, this replica can still retain its original private key. The replica can then pretend to be unaware of the current state. Therefore, the slow client might still be able to complete its request in the superseded configuration and return a stale state, which would violate the safety properties of the system. We show that this issue can be addressed by an additional "confirming" round-trip executed by the client.

Our contribution: Byzantine fault-tolerant reconfigurable services. We provide a systematic solution to each of the challenges described above and present a set of techniques for building reconfigurable services in asynchronous model with Byzantine faults of both clients and replicas. We consider a very strong model of the adversary: any number of clients can be Byzantine and, as soon as some configuration is installed, no assumptions are made about the correctness of replicas in any of the prior configurations.

Moreover, in our quest for a simple solution for the Byzantine model, we devised a new approach to building asynchronous reconfigurable services by further exploring the connection between reconfiguration and the lattice agreement abstraction [23, 26]. We believe that this approach can be usefully applied to crash fault-tolerant systems as well.

Instead of trying to build a complex graph of configurations "on the fly" while transferring the state between those configurations, we start by simply assuming that we are already given a *linear history* (i.e., a sequence of configurations). We introduce the notion of a *dynamic object* – an object that can transfer its own state between the configurations of a given finite linear history and serve meaningful user requests. We then provide dynamic implementations of several important object types such as Lattice Agreement and Max-Register and expect that other asynchronous static algorithms can be translated to the dynamic model using a similar set of techniques.

Finally, we present a *general transformation* that allows us to combine *any* dynamic object with two *Dynamic Byzantine Lattice Agreement* objects in such a way that together they constitute a single *reconfigurable* object, which exports a general-purpose reconfiguration interface and supports all the operations of the original dynamic object.

Roadmap. The rest of the paper is organized as follows. We overview the model assumptions in Section 2 and define our principal abstractions in Section 3. In Section 4, we describe our implementation of Dynamic Byzantine Lattice Agreement and in Section 5, we show how to use it to implement a reconfigurable object. We discuss related work in Section 6 and conclude in Section 7. We refer to [27] for the full version of the paper.

2 System Model

Processes and channels. We consider a system of *processes*. A process can be a *replica* or a *client*. 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 not taking any 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. We assume that a process that remains correct forever (we call it *forever-correct*) does not prematurely stop taking steps of its algorithm.

We assume asynchronous *reliable authenticated* point-to-point links between each pair of processes [13]. 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.

We assume that the adversary is computationally bounded so that it is unable to break the cryptographic techniques, such as digital signatures, forward security schemes [10] and one-way hash functions.

Configuration lattice. A *join semi-lattice* (or simply a *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 $x \sqcup y$. $\sqcup$ is called the *join operator*. It is an associative, commutative, and idempotent binary operator on $\mathcal{L}$. We write $x \sqsubset y$ whenever $x \sqsubseteq y$ and $x \neq y$. We say that $x, y \in \mathcal{L}$ are *comparable* iff either $x \sqsubseteq y$ or $y \sqsubset x$.

For any (potentially infinite) set A, $(2^A, \sqsubseteq)$ is a join semi-lattice, called *the powerset lattice of* A. For all $Z_1, Z_2 \in 2^A$, $Z_1 \sqsubseteq Z_2 \triangleq Z_1 \subseteq Z_2$ and $Z_1 \sqcup Z_2 \triangleq Z_1 \cup Z_2$.

A configuration is an element of a join semi-lattice $(\mathcal{C}, \sqsubseteq)$. We assume that every configuration is associated with a finite set of replicas via a map $\mathit{replicas} : \mathcal{C} \to 2^{\Phi}$, and a *quorum system* via a map $\mathit{quorums} : \mathcal{C} \to 2^{2^{\Phi}}$, such that $\forall C \in \mathcal{C} : \mathit{quorums}(C) \subseteq 2^{\mathit{replicas}(C)}$. Additionally we assume that there is a map $\mathit{height} : \mathcal{C} \to \mathbb{Z}$, such that $\forall C \in \mathcal{C} : \mathit{height}(C) \geq 0$ and $\forall C_1, C_2 \in \mathcal{C}$: if $C_1 \sqsubset C_2$, then $\mathit{height}(C_1) < \mathit{height}(C_2)$. We say that a configuration C is *higher* (resp., *lower*) than a configuration D iff $D \sqsubset C$ (resp, $C \sqsubset D$).[1]

[1] Notice that "C is higher than D" implies "$\mathit{height}(C) > \mathit{height}(D)$", but not vice versa.

We say that *quorums*(C) is 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 at least one quorum is *available* (all its replicas are correct) at time t.

A natural (but not the only possible) way to define the lattice $\mathcal{C}$ is as follows: 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 $(2^{Updates}, \sqsubseteq)$. The mappings *replicas*, *quorums*, and *height* are defined as follows: $replicas(C) \triangleq \{s \in \Phi \mid (+,s) \in C \wedge (-,s) \notin C\}$, $quorums(C) \triangleq \{Q \subseteq replicas(C) \mid |Q| > \frac{2}{3}\,|replicas(C)|\}$, 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. Notice that, when this lattice is used for configurations, once a replica is removed from the system, it cannot be added again with the same identifier. In order to add such a replica back to the system, a new identifier must be used.

Forward-secure digital signatures. In a *forward-secure digital signature scheme* [10, 30, 11, 15] 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 $t_1 < t_2 \leq T$.[2] However "downgrading" the key to a lower timestamp, from sk_{t_2} to sk_{t_1}, is computationally infeasible. This way, if the signer updates their secret key to some timestamp t and then removes the original secret key, it will not be able to sign new messages with a timestamp lower than t, even if it later turns Byzantine.

For simplicity, we model a forward-secure signature scheme as an oracle which associates every process p with a timestamp st_p (initially, $st_p = 0$). The oracle provides p with three operations: (1) $\mathrm{UpdateFSKey}(t)$ sets st_p to $t \geq st_p$; (2) $\mathrm{FSSign}(m,t)$ returns a signature for message m and timestamp t if $t \geq st_p$, otherwise it returns $\perp$; and (3) $\mathrm{FSVerify}(m,p,s,t)$ returns *true* iff s was generated by invoking $\mathrm{FSSign}(m,t)$ by process p.[3]

In our protocols, we use the height of the configuration as the timestamp. When a replica answers requests in configuration C, it signs messages with timestamp *height*(C). When a higher configuration D is installed, the replica invokes $\mathrm{UpdateFSKey}(height(D))$. This prevents the "I still work here" attack described in Section 1.

3 Abstractions and Definitions

In this section, we introduce principal abstractions of this paper (access control-interface, Byzantine Lattice Agreement, Reconfigurable and Dynamic objects), state our quorum assumptions, and recall the definitions of broadcast primitives used in our algorithms.

3.1 Access control

In our implementations and definitions, we parametrize some abstractions by boolean functions $\mathrm{VerifyInputValue}(v,\sigma)$ and $\mathrm{VerifyInputConfig}(C,\sigma)$, where σ is called a *certificate*. Moreover, some objects also export a boolean function $\mathrm{VerifyOutputValue}(v,\sigma)$, which lets anyone to verify that the value v was indeed produced by the object. This helps us to deal with Byzantine clients. In particular, it achieves three important goals.

---

[2] T is a parameter of the scheme and can be set arbitrarily large (with some modest overhead). We believe that $T = 2^{32}$ or $T = 2^{64}$ should be sufficient for most applications.

[3] We assume 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 identifier.

First, the parameter VerifyInputConfig allows us to prevent Byzantine clients from reconfiguring the system in an undesirable way or simply flooding the system with excessively frequent reconfiguration requests. In the full version of this paper [27], we propose two simple approaches: each reconfiguration request must be signed by a quorum of replicas of some configuration[4] or by a quorum of preconfigured administrators.

Second, the parameter VerifyInputValue(v, σ) allows us 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 can 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 that client.

Finally, the exported function VerifyOutputValue allows us to *chain* several distributed objects in such a way that the output of one object is passed as input for another one.

3.2 Byzantine Lattice Agreement abstraction

In this section we formally define *Byzantine Lattice Agreement* abstraction (BLA for short), which serves as one of the main building blocks for constructing reconfigurable objects. Byzantine Lattice Agreement is an adaptation of Lattice Agreement [16] 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. We say that σ is a *valid certificate for input value v* iff VerifyInputValue$(v, \sigma) = true$.

We say that $v \in \mathcal{L}$ is 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, i.e., it maintains v and a valid certificate σ in its local memory. We require that the adversary is 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.[5]

- Operation Propose(v, σ) returns a response of the form $\langle w, \tau \rangle$, 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, we say that τ is a *valid certificate for output value w* iff VerifyOutputValue$(w, \tau) = true$. We say that w is 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:

- *BLA-Validity*: Every verifiable output value w is a join of some set of verifiable input values;
- *BLA-Verifiability*: If Propose(...) returns $\langle w, \tau \rangle$ to a correct process, then VerifyOutputValue$(w, \tau) = true$;
- *BLA-Inclusion*: If Propose(v, σ) returns $\langle w, \tau \rangle$ 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.

[4] Additional care is needed to prevent the "slow reader" attack. See [27] for more details.

[5] The main difference between an operation and a function is that a function can be computed without communicating with other processes and it always returns the same result given the same input.

For the sake of simplicity, we only guarantee liveness when there are finitely many verifiable input values. This is sufficient for the purposes of reconfiguration. The abstraction that provides unconditional liveness is called *Generalized Lattice Agreement* [16].

3.3 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, we define the notion of a reconfigurable object in a very abstract way. By combining this definition with the definition of a Byzantine Lattice Agreement from Section 3.2, we 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 [27]).

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. As for verifiable input values, we say that $C \in \mathcal{C}$ is a *verifiable input configuration* in a given run iff at some point in that run, some process knows σ such that VerifyInputConfig$(C, \sigma) = true$.

We require the total number of verifiable input configurations 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. This requirement is imposed by all asynchronous reconfigurable storage systems [1, 34, 26, 3] we are aware of, and, in fact, can be shown to be necessary [33].

When a correct replica r is ready to serve user requests in configuration C, it triggers upcall InstalledConfig(C). We then say that r *installs* configuration C. A configuration is called *installed* if some correct replica installed it. Finally, a configuration 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 C or a higher configuration will eventually be installed by all correct replicas.

3.4 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 (we call this *state transfer*). We decouple these two problems by introducing the notion of a *dynamic* object. Dynamic objects solve the second problem while "outsourcing" the first one.

Before we formally define dynamic objects, let us first define the notion of a *history*. In Section 2, we introduced the configuration lattice $\mathcal{C}$. A finite set $h \subseteq \mathcal{C}$ is called a *history* iff all elements of h are comparable (in other words, if they form a sequence). 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. We say that h is a *verifiable*

history in a given run iff at some point in that run, some process knows σ such that VerifyHistory(h, σ) = *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, we require all verifiable histories to be related by containment (i.e., comparable w.r.t. $\subseteq$). More formally, if VerifyHistory(h_1, σ_1) = *true* and VerifyHistory(h_2, σ_2) = *true*, then $h_1 \subseteq h_2$ or $h_2 \subseteq h_1$. We discuss how to build such histories in Section 5.

Similarly to reconfigurable objects, a dynamic object must have the InstalledConfig(C) upcall. 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. As the main contribution of this paper, we show how to build *dynamic* objects in a Byzantine environment and how to create *reconfigurable* objects using dynamic objects as building blocks. We argue that this technique is applicable to a large class of objects.

3.5 Quorum system assumptions

Most fault-tolerant implementations of distributed objects impose some requirements on the subsets of processes that can be faulty. We say that a configuration C is *available at time t* iff *replicas*(C) is a dissemination quorum system at time t (as defined in Section 2). Correctness of our implementations of *dynamic* objects relies on the assumption that active candidate configurations are available. Once a configuration is superseded by a higher configuration, we make no further assumptions about it.

For *reconfigurable* objects we impose a slightly more conservative requirement: every combination of verifiable input configurations that is not yet superseded must be available. More formally, let $C_1, \dots, C_k$ be verifiable input configurations such that $C = C_1 \sqcup \dots \sqcup C_k$ is not superseded at time t. Then we require *quorums*(C) to be a dissemination quorum system at time t.

Correctness of our *reconfigurable* objects relies solely on correctness of the dynamic building blocks. Formally, when k configurations are concurrently proposed, we require all possible combinations, i.e., $2^k - 1$ configurations, to be available. However, in practice, at most k of them will be chosen to be put in verifiable histories, and only those configurations actually need to be available. We impose a more conservative requirement because we do not know these configurations *a priori*.

3.6 Broadcast primitives

To make sure that no slow process is "left behind", we assume that a variant of *reliable broadcast primitive* [13] is available. 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. In practice such primitive can be implemented by some sort of a gossip protocol [24]. This primitive is "global" in a sense that it is not bound to any particular configuration. In pseudocode we use "**RB-Broadcast** $\langle ... \rangle$" to denote a call to the "global" reliable broadcast.

Additionally, we assume a "local" *uniform reliable broadcast* primitive [13]. It has a stronger totality property: if some *correct* process p delivered some message m, then every forever-correct process will eventually deliver m, even if p later turns Byzantine. This primitive can be implemented in a *static* system, provided a quorum system. As we deal with *dynamic* systems, we associate every broadcast message with a fixed configuration and only guarantee these properties if the configuration is *never superseded*. Notice that any static implementation of uniform reliable broadcast trivially guarantees this property. In pseudocode we use "**URB-Broadcast** $\langle ... \rangle$ **in** C" to denote a call to the "local" uniform reliable broadcast in configuration C.

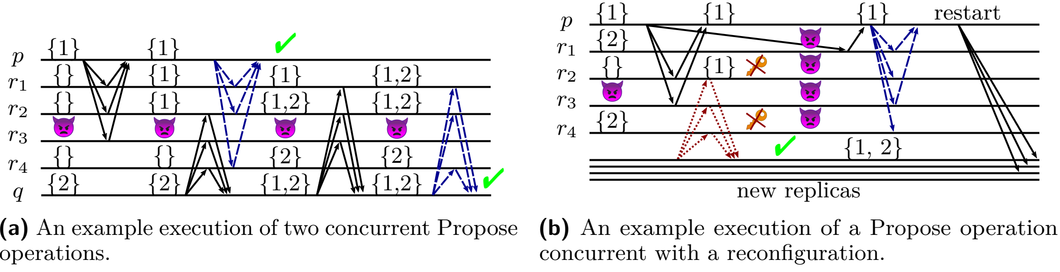
4 Dynamic Byzantine Lattice Agreement

Dynamic Byzantine Lattice Agreement abstraction (DBLA for short) is the main building block in our construction of reconfigurable objects. Its specification is a combination of the specification of Byzantine Lattice Agreement (Section 3.2) and the specification of a dynamic object (Section 3.4). The interface of a DBLA object is depicted in Figure 2a. To respect space limits, the complete pseudocode, the proof of correctness, and further discussion are presented in [27].

As we mentioned earlier, we use forward-secure digital signatures 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), we would like to make sure that the replicas of all candidate configurations lower than C invoke UpdateFSKey($height(C)$). 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 an asynchronous system [17].

Instead, we classify all candidate configurations in two categories: *pivotal* and *tentative*. A candidate configuration is called *pivotal* if it is the last 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 us 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.

In order to reconfigure a DBLA object, a correct client must use reliable 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 $history_p$. We use $Chighest_p$ to denote the most recent configuration in p's local history (i.e., $Chighest_p = \text{HighestConf}(history_p)$). Whenever a replica r updates $history_r$, it invokes UpdateFSKey($height(Chighest_r)$). Recall



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

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

Figure 1 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.

that if at least one forever-correct process delivers something via reliable broadcast, every other correct process will eventually deliver it as well.

Similarly, each process p keeps track of all verifiable input values it has seen $curVals_p \subseteq \mathcal{L} \times \Sigma$, where 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 with certificates to $curVals_p$.

4.1 Client implementation

The client's protocol is simple. As we mentioned earlier, the operation UpdateHistory(h, σ) is implemented as **RB-Broadcast** $\langle$**NewHistory**$, h, \sigma\rangle$. The rest of the reconfiguration process is handled by the replicas. The protocol for the operation Propose(v, σ) 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. [16]. Client p repeatedly sends message $\langle$**Propose**, $curVals_p$, $seqNum_r$, $C\rangle$ to all replicas in $replicas(C)$, where **Propose** is the message descriptor, $C = Chighest_p$, and $seqNum_r$ 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 combines the responses to form a certificate (called $acks_1$) and proceeds to the second stage. In the first two cases, the client simply restarts the operation. Because the number of verifiable input values, as well as the number of verifiable histories, are assumed to be finite, the number of restarts will also be finite.

The example in Figure 1a 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 $\{2\}$, respectively, from the lattice $\mathcal{L} = 2^{\mathbb{N}}$. 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 $\{2\}$).

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 than waits for a quorum of replicas to reply with a forward-secure signature with timestamp $height(C)$.

The example in Figure 1b illustrates how reconfiguration can interfere with an ongoing Propose operation in what we call *the "slow reader" attack*, and how the second stage of the protocol prevents safety violations. In this example, client p should not be able to return the proposed value $\{1\}$ because all correct replicas in quorum $\{r_1, r_3, r_4\}$ store value $\{2\}$, which means that previously some other client could have returned value $\{2\}$. 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. If r_1 lies to p, the client may finish the first stage of the protocol with value $\{1\}$. However, because 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. The client then waits until it receives the new verifiable history via reliable broadcast and restarts the operation in the new 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; and (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.

4.2 Replica implementation

Each replica r maintains, locally, its *current configuration* (denoted by $Ccurr_r$) and *the last configuration installed by this replica* (denoted by $Cinst_r$). $Cinst_r \sqsubseteq Ccurr_r \sqsubseteq Chighest_r$. Intuitively, $Ccurr_r = C$ 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. $Cinst_r = C$ 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.

As we saw earlier, each client message is associated with some configuration C. The replica only answers the message when $C = Cinst_r = Ccurr_r = Chighest_r$. If $C \sqsubset Chighest_r$, the replica simply ignores the message. Due to the properties of reliable broadcast, the client will eventually learn about $Chighest_r$ and will repeat its request there (or in an even higher configuration). If $Cinst_r \sqsubset C$ and $Chighest_r \sqsubseteq C$, the replica waits until C is installed before processing the message. Finally, if C is incomparable with $Cinst_r$ or $Chighest_r$, then 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 $curVals_r$ 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 Ccurr ≠ HighestConf({C ∈ history | r ∈ replicas(C)})
2:      let Cnext = HighestConf({C ∈ history | r ∈ replicas(C)})
3:      let S = {C ∈ history | Ccurr ⊑ C ⊏ Cnext}
4:      seqNum ← seqNum + 1
5:      for each C ∈ S do
6:          send ⟨UpdateRead, seqNum, C⟩ to replicas(C)
7:          wait for (C ⊏ Ccurr) ∨ (responses from any Q ∈ quorums(C) with s.n. seqNum)
8:      if Ccurr ⊏ Cnext then
9:          Ccurr ← Cnext
10:         URB-Broadcast ⟨UpdateComplete⟩ in Cnext
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 Cinst ⊏ C then
19:         if Ccurr ⊏ C then Ccurr ← C
20:         Cinst ← C
21:         trigger upcall InstalledConfig(C)
22:         if r ∉ replicas(C) then halt
```

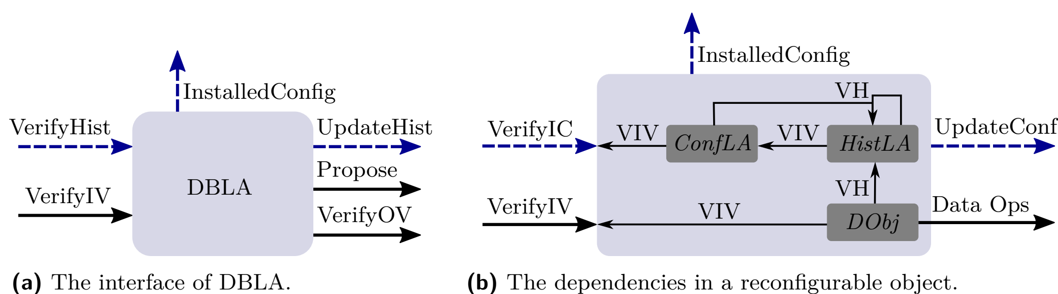
A very important part of the replica's implementation is *the state transfer protocol.* The pseudocode for it is presented in Algorithm 1. Note that we omit the subscript r in the pseudocode because each process can only access its own variables directly. Let $Cnext_r$ be the highest configuration in $history_r$ such that $r \in replicas(Cnext_r)$. Whenever $Ccurr_r \neq Cnext_r$, the replica tries to "move" to $Cnext_r$ by reading the current state from all configurations between $Ccurr_r$ and $Cnext_r$ one by one in ascending order (line 5). In order to read the current state from configuration $C \sqsubset Cnext_r$, replica r sends message $\langle$**UpdateRead**, $seqNum_r$, $C\rangle$ 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). We maintain the invariant that for any correct replica r': $st_{r'} = height(\text{HighestConf}(history_{r'}))$, where $st_{r'}$ is the timestamp of the private key of r'.

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, $v \sqsubseteq \text{JoinAll}(curVals_r)$, where v is the value returned by the Propose operation and $\text{JoinAll}(curVals_r)$ is the join of all verifiable input values in the set $curVals_r$; 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 1b).
- 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 $Ccurr_r$ (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



(a) The interface of DBLA. **(b)** The dependencies in a reconfigurable object.

Figure 2 (a): The interface of a DBLA object. Parameters are depicted on the left side, operations and functions are on the right side, and upcalls are at the top. The parts of the interface that are inherited from BLA are depicted as black arrows, while the parts of the interface that are inherited from the specification of a dynamic object are depicted as dashed blue arrows. "IV", "OV", and "Hist" are abbreviations for "InputValue", "OutputValue", and "History", respectively. **(b):** The structure of dependencies in our implementation of a reconfigurable object. An arrow from an object A to another object B marked with VIV (resp., VH) indicates that A.VerifyInputValue (resp., A.VerifyHistory) is implemented using B.VerifyOutputValue.

quorum of replicas in C (line 16). Because we want our protocol to 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 uniform reliable broadcast primitive that we introduced in Section 3.6.

4.3 Implementing other dynamic objects

While we do not provide any general approach for building dynamic objects, we expect that most asynchronous Byzantine fault-tolerant static algorithms can be adopted to the dynamic case by applying the same set of techniques. These techniques include our state transfer protocol (relying on forward-secure signatures), the use of an additional round-trip to prevent the "slow reader" attack, and the structure of our cryptographic proofs ensuring that tentative configurations cannot create valid certificates for output values. To illustrate this, in the full version of this paper [27], we present the dynamic version of Max-Register [5] and discuss the dynamic version of the Access Control abstraction.

5 Implementing reconfigurable objects

While dynamic objects are important building blocks, they are not particularly useful by themselves because they require an external source of comparable verifiable histories. In this section, we show how to combine several dynamic objects to obtain a single *reconfigurable* object. Similar 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 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.

We build a reconfigurable object by combining three *dynamic* ones. The first one is the dynamic object that we want to make reconfigurable (let us call it *DObj*). For example, it can be an instance of DBLA if we wanted to make a reconfigurable version of Byzantine

Lattice Agreement. 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 2b.

Algorithm 2 Reconfigurable object (short version).

```
    ▷ Code for client p
23: Data operations are performed directly on DObj.

24: operation UpdateConfig(C, σ)
25:     let ⟨C′, σ_C′⟩ = ConfLA.Propose(C, σ)
26:     let ⟨h, σ_h⟩ = HistLA.Propose({C′}, σ_C′)
27:     DObj.UpdateHistory(h, σ_h)
28:     ConfLA.UpdateHistory(h, σ_h)
29:     HistLA.UpdateHistory(h, σ_h)

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

The short version of 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 3.4. Moreover, the verifiable output values of *HistLA* are related by containment, and, therefore, can be used as verifiable histories in dynamic objects. We use them to reconfigure all three dynamic objects (lines 27–29).

The full pseudocode of the transformation with formal specification of all parameters, as well as the proof of correctness and the discussion of possible optimizations, are presented in the full version of this paper [27]. Additionally, in [27] we discuss several ways to prevent the Byzantine clients from reconfiguring the system in an undesirable way.

6 Related work

Dynamic replicated systems with *passive reconfiguration* [9, 7, 25] 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 [25], 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 [22] and reconfigurable Paxos [28], used consensus (and, thus, assumed certain level of synchrony) to ensure that the clients agree on the evolution of configurations. DynaStore [2] was the first *asynchronous* reconfigurable storage: clients propose incremental additions or removals to the system configuration. As the proposals commute, the processes can resolve their disagreements without involving consensus.

The *parsimonious speculative snapshot* task [19] allows to resolve conflicts between concurrent configuration updates in a storage system using instances of commit-adopt [18]. The worst-case time complexity, in the number of message delays, of reconfiguration was later reduced from $O(n^2)$ to $O(n)$ [34], where n is the number of concurrently proposed configuration updates.

SmartMerge [23] made an important step forward by treating reconfiguration as an instance of abstract *lattice agreement* [16]. However, the algorithm assumes an external (reliable) lattice agreement service which makes the system not fully reconfigurable. The recently proposed *reconfigurable lattice-agreement* abstraction [26] enables truly reconfigurable versions of a large class of objects and constructions, including state-based CRDTs [32], atomic-snapshot, max-register, conflict detector and commit-adopt. We believe that the reconfiguration service we introduced in this paper can be used to derive Byzantine fault-tolerant reconfigurable implementations of objects in the class.

Byzantine quorum systems [29] 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 [4] 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 [31] 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 [1]. In this paper, we propose an implementation of a Byzantine fault-tolerant reconfiguration service that does not rely on this assumption.

Forward-secure signature schemes [10, 11, 14, 15, 30] enable a decentralized way to construct a sequence of distinct private keys for each process. We use the scheme to provide each process with a unique private key for each configuration. To counter the "I still work here" attack, we ensure that sufficiently many correct processes destroy their configuration keys before a new configuration is installed, without relying on a global agreement of the configuration sequence [31].

7 Discussion

Communication cost. In this paper, we do not intend to provide the optimal implementations of each object or to implement the most general abstractions (such as generalized lattice agreement [16, 26]). Instead, we focused 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, our implementations leave plenty of space for optimizations. We discuss a few possible directions in the full version of this paper [27].

Open questions. We would like to mention two relevant directions for further research.

First, with regard to active reconfiguration, it would be interesting to devise algorithms that efficiently adapt to "small" configuration changes, while still supporting the option of completely changing the set of replicas in a single reconfiguration request. In this paper, we allow the sets of replicas of proposed configurations to be completely disjoint, which incurred an expensive quorum-to-quorum communication pattern. This might seem unnecessary for reconfigurations requests involving only slight changes of the set of replicas.

Second, with regard to Byzantine faults, it would be interesting to consider models with a "weaker" adversary. In this paper, we assumed a very strong model of the adversary: no assumptions are made about correctness of replicas in superseded configurations. This "pessimistic" approach leads to more complicated and expensive protocols.

References

1. 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.
2. Marcos Kawazoe Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, 2011.
3. 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.
4. 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.
5. James Aspnes, Hagit Attiya, and Keren Censor. Max registers, counters, and monotone circuits. In *PODC*, pages 36–45, 2009.
6. 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.
7. 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.
8. Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Comput.*, 8(3):121–132, 1995.
9. Roberto Baldoni, Silvia Bonomi, Anne-Marie Kermarrec, and Michel Raynal. Implementing a register in a dynamic distributed system. In *ICDCS*, pages 639–647, 2009.
10. Mihir Bellare and Sara K Miner. A forward-secure digital signature scheme. In *Annual International Cryptology Conference*, pages 431–448. Springer, 1999.
11. 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.
12. Eric A. Brewer. Towards robust distributed systems (abstract). In *PODC*, pages 7–, 2000.
13. Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011.
14. Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. *Journal of Cryptology*, 20(3):265–294, 2007.
15. 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`.
16. Jose Faleiro, Sriram Rajamani, Kaushik Rajan, Ganesan Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In *PODC*, pages 125–134, 2012.
17. 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.
18. Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *PODC*, pages 143–152, 1998.
19. Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *DISC*, pages 140–153, 2015.
20. David K. Gifford. Weighted voting for replicated data. In *SOSP*, pages 150–162, 1979.
21. Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.
22. 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.
23. Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *DISC*, pages 154–169, 2015.

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

25 Saptaparni Kumar and Jennifer L. Welch. Byzantine-tolerant register in a system with continuous churn. *CoRR*, abs/1910.06716, 2019. `arXiv:1910.06716`.

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

27 Petr Kuznetsov and Andrei Tonkikh. Asynchronous reconfiguration with byzantine failures. *CoRR*, abs/2005.13499, 2020. URL: `https://arxiv.org/abs/2005.13499`.

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

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

30 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.

31 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.

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

33 Alexander Spiegelman and Idit Keidar. On liveness of dynamic storage. In *Structural Information and Communication Complexity - 24th International Colloquium, SIROCCO 2017, Porquerolles, France, June 19-22, 2017, Revised Selected Papers*, pages 356–376, 2017.

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