View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Reconfigurable Lattice Agreement and Applications

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

Thibault Rieutord
CEA LIST, PC 174, Gif-sur-Yvette, 91191, France
thibault.rieutord@cea.fr

Sara Tucci-Piergiovanni
CEA LIST, PC 174, Gif-sur-Yvette, 91191, France
sara.tucci@cea.fr

Abstract

Reconfiguration is one of the central mechanisms in distributed systems. Due to failures and connectivity disruptions, the very set of service replicas (or *servers*) and their roles in the computation may have to be reconfigured over time. To provide the desired level of consistency and availability to applications running on top of these servers, the *clients* of the service should be able to reach some form of agreement on the system configuration. We observe that this agreement is naturally captured via a *lattice* partial order on the system states. We propose an asynchronous implementation of *reconfigurable* lattice agreement that implies elegant reconfigurable versions of a large class of *lattice* abstract data types, such as max-registers and conflict detectors, as well as popular distributed programming abstractions, such as atomic snapshot and commit-adopt.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Reconfigurable services, lattice agreement

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.31

Related Version A full version of the paper is available at `https://arxiv.org/abs/1910.09264`.

1 Introduction

A decentralized service [6,14,25,28] runs on a set of fault-prone *servers* that store replicas of the system state and run a synchronization protocol to ensure consistency of concurrent data accesses. In the context of a storage system exporting read and write operations, several proposals [2,3,18,20,23,31] came out with a reconfiguration interface that allows the servers to join and leave while ensuring consistency of the stored data. Early proposals of reconfigurable storage systems [20] were based on using *consensus* [16,21] to ensure that replicas *agree* on the evolution of the system membership. Consensus, however, is expensive and difficult to implement, and recent solutions [2,3,18,23,31] replace consensus with weaker abstractions capturing the minimal coordination required to safely modify the system configuration. These solutions, however, lack a uniform way of deriving reconfigurable versions of static objects.

Lattice objects. In this paper, we propose a universal reconfigurable construction for a large class of objects. Unlike a consensus-based reconfiguration proposed earlier for generic state-machine replication [26], our construction is purely asynchronous, at the expense of assuming a restricted object behavior. More precisely, we assume that the set $\mathcal{L}$ of the object's states can be represented as a (join semi-) *lattice* $(\mathcal{L}, \sqsubseteq)$, where $\mathcal{L}$ is partially ordered by the binary relation $\sqsubseteq$ such that for all elements of $x, y \in \mathcal{L}$, there exists the *least upper bound* in $\mathcal{L}$, denoted $x \sqcup y$, where $\sqcup$, called the *join* operator, is an associative, commutative,

© Petr Kuznetsov, Thibault Rieutord, and Sara Tucci-Piergiovanni;
licensed under Creative Commons License CC-BY

and idempotent binary operator on $\mathcal{L}$. Many important data types, such as sets and counters, as well as useful concurrent abstractions, such as conflict detector [5], can be expressed this way. Intuitively, $x \sqcup y$ can be seen as a *merge* of two alternatively proposed updated states x and y. Thus, an implementation ensuring that all "observable" states are ordered by $\sqsubseteq$ cannot be distinguished from an atomic object.

Consider, for example, the *max-register* [4] data type with two operations: *writeMax* writes a value and *readMax* returns the largest value written so far. Its state space can be represented as a lattice $(\sqsubseteq, \sqcup)$ of its values, where $\sqsubseteq = \leq$ and $x \sqcup y = \max(x, y)$. Intuitively, a linearizable implementation of max-register must ensure that every read value is a join of previously proposed values, and all read values are totally ordered (with respect to $\leq$).

Reconfigurable lattice agreement. In this paper, we introduce the *reconfigurable lattice agreement* [8,15]. It is natural to treat the *system configuration*, i.e., the set of servers available for data replication, as an element in a lattice. A lattice-defined join of configurations, possibly concurrently proposed by different clients, results in a new configuration. The lattice-agreement protocol ensures that configurations evaluated by concurrent processes are *ordered*. Despite processes possibly disagreeing about the precise configuration they belong to, they can use the configurations relative ordering to maintain the system data consistency.

A configuration is defined by a set of servers, a quorum system [19], i.e., a set system ensuring the intersection property[1] and, possibly, other parameters. For example, elements of a reconfiguration lattice can be defined as sets of *configuration updates*: each such update either adds a server to the configuration or removes a server from it. The *members* of such a configuration are the set of all servers that were added but not yet removed. A join of two configurations defined this way is simply a union of their updates (this approach is implicitly used in earlier asynchronous reconfigurable constructions [2, 18, 31]).

Reconfigurable L-ADTs and applications. We show that our reconfigurable lattice agreement, defined on a product of a *configuration lattice* and an *object lattice*, immediately implies reconfigurable versions of many sequential types, such as *max-register* and *conflict detector*. More generally, any *state-based commutative* abstract data (called *L-ADT*, for *lattice abstract data type*, in this paper) has a reconfigurable *interval-linearizable* [12] implementation. Intuitively, interval-linearizability [12], a generalization of the classical linearizability [22], allows specifying the behavior of an object when multiple concurrent operations "influence" each other. Their effects are then merged using a join operator, which turns out to be natural in the context of reconfigurable objects.

Our transformations are straightforward. To get an (interval-linearizable) reconfigurable implementation of an L-ADT, we simply use its state lattice, as a parameter, in our reconfigurable lattice agreement. The resulting implementations are naturally composable: we get a reconfigurable composition of two L-ADTs by using a *product* of their lattices. If operations on the object can be partitioned into *updates* (modifying the object state without providing informative responses) and *queries* (not modifying the object state), as in the case of max-registers, the reconfigurable implementation becomes linearizable[2].

[1] The most commonly used quorum system is majority-based: quorums are all majorities of servers. We can, however, use any other quorum system, as suggested in [20, 23].

[2] Such "update-query" L-ADTs are known as state-based convergent replicated data types (CvRDT) [29]. These include *max-register*, *set* and *abort flag* (a new type introduced in this paper).

We then use our reconfigurable implementations of *max-register*, *conflict detector*, *set* and *abort-flag* to devise reconfigurable versions of *atomic snapshot* [1], *commit-adopt* [17] and *safe agreement* [10]. Figure 1 shows how are constructions are related. Due to lack of space, the *safe agreement* and its associated objects are delegated to the technical report [24].

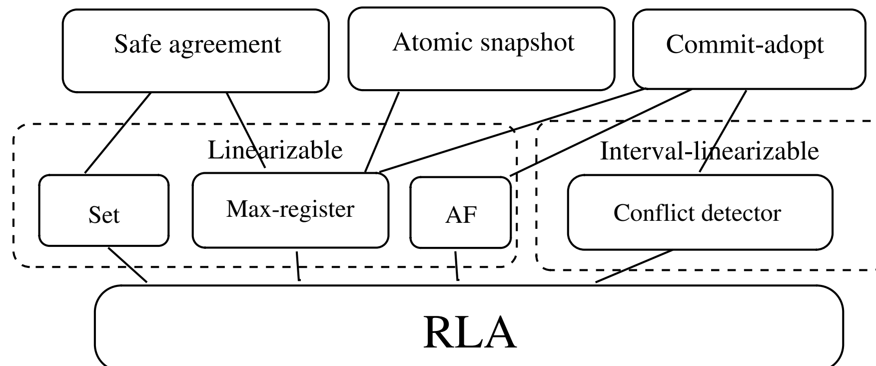

Figure 1 Our reconfigurable implementations: reconfigurable lattice agreement (RLA) is used to construct linearizable implementations of a set, a max-register, an abort flag, and an interval-linearizable implementation of a conflict detector. On top of max-registers we construct an atomic snapshot; on top of a max-register, an abort-flag, and a conflict detector, we construct a commit-adopt abstraction; and, on top of sets and a max register, we implement a safe agreement abstraction.

Summary. Our reconfigurable construction is the first to be, at the same time:

- Asynchronous, unlike consensus-based solutions [13, 20, 26], and not assuming an external lattice agreement service [23];
- Uniformly applicable to a large class of objects, unlike existing reconfigurable systems that either focus on read-write storage [2,18,20,23] or require data type-specific implementations of exported reconfiguration interfaces [31];
- Allowing for a straightforward composition of reconfigurable objects;
- Maintaining configurations with abstract *quorum systems* [19], not restricted to *majority-based* quorums [2, 18];
- Exhibiting optimal time complexity and message complexity comparable with the best known implementations [2, 23, 31];
- Logically separating *clients* (external entities that use the implemented service) from *servers* (entities that maintain the service and can be reconfigured).

We also believe our reconfigurable construction to be the simplest on the market, using only twenty one lines of pseudocode and provided with a concise proof.

Roadmap. The rest of the paper is organized as follows. We give basic model definitions in Section 2. In Section 3, we define our type of reconfigurable objects, followed by the related notion of reconfigurable lattice agreement in Section 4. In Section 5, we describe our implementation of reconfigurable lattice agreement, and, in Section 6, we show how to use it to implement a reconfigurable L-ADT object. In Section 7 we describe some possible applications. We conclude with, in Section 8, an overview of the related work, and, in Section 9, a discussion on algorithms complexity and possible trade-offs. The full version of this paper, with detailed proofs, is available at [24].

2 Definitions

Replicas and clients. Let Π be a (possibly infinite) set of potentially participating processes. A subset of the processes, called *replicas*, are used to maintain a replicated object. A process can also act as a *client*, invoking operations on the object and proposing system reconfigurations. Both replicas and clients are subject to crash failures: a process *fails* when it prematurely stops taking steps of its algorithm. A *failure model* stipulates when and where failures might occur. We present our failure model in Section 4, where we formally define reconfigurable lattice agreement.

Abstract data types. An abstract data type (*ADT*) is a tuple $T = (A, B, Z, z_0, \tau, \delta)$. Here A and B are countable sets called the *inputs* and *outputs*. Z is a countable set of abstract object *states*, $z_0 \in Z$ being the initial state of the object. The map $\tau : Z \times A \to Z$ is the *transition function*, specifying the effect of an input on the object state and the map $\delta : Z \times A \to B$ is the *output function*, specifying the output returned for a given input and object local state. The input represents an operation with its parameters, where (i) the operation can have a side-effect that changes the abstract state according to transition function τ and (ii) the operation can return values taken in the output B, which depends on the state in which it is called and the output function δ (for simplicity, we only consider deterministic types here, check, e.g., [27], for more details.)

Interval linearizability. We now briefly recall the notion of *interval-linearizability* [12], a recent generalization of linearizability [22].

Let us consider an abstract data type $T = (A, B, Z, z_0, \tau, \delta)$. A *history* of T is a sequence of inputs (elements of A) and outputs (elements of B), each labeled with a process identifier and an operation identifier. An *interval-sequential history* is a sequence:

$$z_0, I_1, R_1, z_1, I_2, R_2, z_2 \dots, I_m, R_m, z_m,$$

where each $z_i \in Z$ is a state, $I_i \subseteq A$ is a set of inputs, and $R_i \subseteq B$ is a set of outputs. An *interval-sequential specification* is a set of interval-sequential histories.

We only consider *well-formed* histories. Informally, in a well-formed history, a process only invokes an operation once its previous operation has returned and every response r is preceded by a "matching" operation i.

A history H is *interval-linearizable* respectively to an interval-sequential specification $\mathcal{S}$ if it can be *completed* (by adding matching responses to incomplete operations) so that the resulting history $\bar{H}$ can be associated with an interval-sequential history S such that: (1) $\bar{H}$ and S are *equivalent*, i.e., $\forall p \in \Pi$, $\bar{H}|p = S|p$, (2) $S \in \mathcal{S}$, and (3) $\to_H \subseteq \to_S$, i.e., S preserves the real-time precedence relation of H. (Check [12] for more details on the definition.)

Lattice agreement. An abstract (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 of $x, y \in \mathcal{L}$, there exists the least upper bound for the set $\{x, y\}$. The least upper bound is an associative, commutative, and idempotent binary operation on $\mathcal{L}$, denoted by $\sqcup$ and called the *join operator* on $\mathcal{L}$. We write $x \sqsubset y$ whenever $x \sqsubseteq y$ and $x \neq y$. With a slight abuse of notation, for a set $L \subseteq \mathcal{L}$, we also write $\bigsqcup L$ for $\bigsqcup_{x \in L} x$, i.e., $\bigsqcup L$ is the join of the elements of L.

Notice that two lattices $(\mathcal{L}_1, \sqsubseteq_1)$ and $(\mathcal{L}_2, \sqsubseteq_2)$ naturally imply a *product* lattice $(\mathcal{L}_1 \times \mathcal{L}_2, \sqsubseteq_1 \times \sqsubseteq_2)$ with a product join operator $\sqcup = \sqcup_1 \times \sqcup_2$. Here for all $(x_1, x_2), (y_1, y_2) \in \mathcal{L}_1 \times \mathcal{L}_2$, $(x_1, x_2)(\sqsubseteq_1 \times \sqsubseteq_2)(y_1, y_2)$ if and only if $x_1 \sqsubseteq_1 y_1$ and $x_2 \sqsubseteq_2 y_2$.

The (generalized) *lattice agreement* concurrent abstraction, defined on a lattice $(\mathcal{L}, \sqsubseteq)$, exports a single operation *propose* that takes an element of $\mathcal{L}$ as an argument and returns an element of $\mathcal{L}$ as a response. When the operation *propose*(x) is invoked by process p we say that p *proposes* v, and when the operation returns v' we say that p *learns* v'. Assuming that no process invokes a new operation before its previous operation returns, the abstraction satisfies the following properties:

- **Validity.** If a *propose*(v) operation returns a value v' then v' is a join of some proposed values including v and all values learnt before the invocation of the operation.
- **Consistency.** The learnt values are totally ordered by $\sqsubseteq$.
- **Liveness.** If a process invokes a *propose* operation and does not fail then the operation eventually returns.

A historical remark. The original definition of long-lived lattice agreement [15] separates "receive" events and "learn" events. Here we suggest a simpler definition that represents the two events as the invocation and the response of a *propose* operation. This also allows us to slightly strengthen the validity condition so that it accounts for the *precedence* relation between *propose* operations. As a result, we can directly relate lattice agreement to linearizable [22] and interval-linearizable [12] implementations, without introducing artificial "nop" operations [15].

3 Lattice Abstract Data Type

In this section, we introduce a class of types that we call *lattice abstract data types* or *L-ADT*. In an *L-ADT*, the set of states forms a join semi-lattice with a partial order $\sqsubseteq^Z$. A lattice object is therefore defined as a tuple $L = (A, B, (Z, \sqsubseteq^Z, \sqcup^Z), z_0, \tau, \delta)$.[3] Moreover, the transition function δ must comply with the partial order $\sqsubseteq^Z$, that is $\forall z, a \in Z \times A : z \sqsubseteq^Z \tau(z, a)$, and the composition of transitions must comply with the join operator, that is $\forall z \in Z, \forall a, a' \in A : \tau(\tau(z, a), a') = \tau(z, a) \sqcup^Z \tau(z, a') = \tau(\tau(z, a'), a)$. Hence, we can say that the transition function is "commutative".

Update-query L-ADT. We say an L-ADT $L = (A, B, (Z, \sqsubseteq^Z, \sqcup^Z), z_0, \tau, \delta)$ is *update-query* if A can be partitioned in *updates* U and *queries* Q such that:

- there exists a special "dummy" response $\perp$ (z_0 may also be used) such that $\forall u \in U,\ z \in Z$, $\delta(u, z) = \perp$, i.e., updates do not return informative responses;
- $\forall q \in Q,\ z \in Z,\ \tau(q, z) = z$, i.e., queries do not modify the states.

This class is also known as a state-based convergent replicated data types (CvRDT) [29]. Typical examples of update-query L-ADTs are *max-register* [4] (see Section 1) or *sets*. Note that any (L-)ADT can be transformed into an update-query (L-)ADT by "splitting its operations" into an update and a query (see [27]).

Composition of L-ADTs. The composition of two ADTs $T = (A, B, Z, z_0, \tau, \delta)$ and $T' = (A', B', Z', z'_0, \tau', \delta')$ is denoted $T \times T'$ and is equal to $(A + A', B \cup B', Z \times Z', (z_0, z'_0), \tau'', \delta'')$; where $A + A'$ denotes the disjoint union and where τ'' and δ'' apply, according to the domain A or A' of the input, either τ and δ or τ' and δ' on their respecting half of the state (see [27]).

Since the cartesian product of two lattices remains a lattice, the composition of L-ADTs is naturally defined and produces an L-ADT. The composition is also closed to update-query ADT, and thus to update-query L-ADT. Moreover, the composition is an associative and commutative operator, and hence, can easily be used to construct elaborate L-ADT.

[3] For convenience, we explicitly specify the join operator $\sqcup^Z$ here, i.e., the least upper bound of $\sqsubseteq^Z$.

Configurations as L-ADTs. Let us also use the formlism of L-ADT to define a *configuration L-ADT* as a tuple $(A^{\mathcal{C}}, B^{\mathcal{C}}, (\mathcal{C}, \sqsubseteq^{\mathcal{C}}, \sqcup^{\mathcal{C}}), C_0, \tau^{\mathcal{C}}, \delta^{\mathcal{C}})$ with $C_0 \in \mathcal{C}$ the *initial configuration*. For each element C of the *configuration lattice* $\mathcal{C}$, the input set A includes the query operations $members()$, such that $\delta^{\mathcal{C}}(C, members()) \subseteq \Pi$, and $quorums()$ where $\delta^{\mathcal{C}}(C, quorums()) \subseteq 2^{\delta^{\mathcal{C}}(C, members())}$ is a *quorum system*, that is, every two subsets in $\delta^{\mathcal{C}}(C, quorums())$ have a non-empty intersection. With a slight abuse of notation, we will write these operations as $members(C)$ and $quorums(C)$.

For example, $\mathcal{C}$ can be the set of tuples (In, Out), where $In \subseteq \Pi$ is a set of *activated* processes, and $Out \subseteq \Pi$ is a set of *removed* processes. Then $\sqsubseteq^{\mathcal{C}}$ can be defined as the piecewise set inclusion on (In, Out). The set of members of (In, Out) will simply be $In - Out$ and the set of quorums (pairwise-intersecting subsets of $In - Out$), e.g., all majorities of $In - Out$. Operations in $A^{\mathcal{C}}$ can be $add(s)$, $s \in \Pi$, that adds s to the set of activated processes and $remove(s)$, $s \in \Pi$, that adds s to the set of removed processes of a configuration. One can easily see that updates "commute" and that the type is indeed a configuration L-ADT. Let us note that L-ADTs allow for more expressive reconfiguration operations than simple *adds* and *removes*, e.g., maintaining a minimal number of members in a configuration or adapting the quorum system dynamically, as studied in detail by Jehl et al. in [23].

Interval-sequential specifications of L-ADTs. Let $L = (A, B, (Z, \sqsubseteq^Z, \sqcup^Z), z_0, \tau, \delta)$ be an L-ADT. As τ "commutes", the state reached after a sequence of transitions is order-independent. Hence, we can define a natural, deterministic, interval-sequential specification of L, $\mathcal{S}_L$, as the set of interval-sequential histories $z_0, I_1, R_1, z_1, \dots, I_m, R_m, z_m$ such that:

- $\forall i = 1, \dots, m$, $z_i = \bigsqcup^Z_{a \in I_{i-1}} \tau(a, z_{i-1})$, i.e., every state z_i is a join of operations in I_{i-1} applied to z_{i-1}.
- $\forall i = 1, \dots, m$, $\forall r \in R_i$, $r = \delta(a, z_i)$, where a is the matching invocation operation for r, i.e., every response in R_i is the result of the associated operation applied to state z_i.

4 Reconfigurable lattice agreement: definition

We define a reconfigurable lattice $(\mathcal{L}, \sqsubseteq)$ as the product of the state spaces of an *object* L-ADT $(A^{\mathcal{O}}, B^{\mathcal{O}}, (\mathcal{O}, \sqsubseteq^{\mathcal{O}}, \sqcup^{\mathcal{O}}), O_0, \tau^{\mathcal{O}}, \delta^{\mathcal{O}})$ and a *configuration* L-ADT $(A^{\mathcal{C}}, B^{\mathcal{C}}, (\mathcal{C}, \sqsubseteq^{\mathcal{C}}, \sqcup^{\mathcal{C}}), C_0, \tau^{\mathcal{C}}, \delta^{\mathcal{C}})$ (see Section 3). That is, $(\mathcal{L}, \sqsubseteq) = (\mathcal{O} \times \mathcal{C}, \sqsubseteq^{\mathcal{O}} \times \sqsubseteq^{\mathcal{C}})$ with the product join operator $\sqcup = \sqcup^{\mathcal{O}} \times \sqcup^{\mathcal{C}}$. Our main tool is the reconfigurable lattice agreement, a generalization of lattice agreement operating on $(\mathcal{L}, \sqsubseteq)$. We say that $\mathcal{L}$ is the set of *states*. For a state $u = (O, C) \in \mathcal{L}$, we use notations $u.O = O$ and $u.C = C$.

Failure model. When a client p invokes $propose((O, C))$, we say that p *proposes* object state O and configuration state C. We say that p *learns* an object state O' and a configuration C' if its *propose* invocation returns (O', C').

We say that a configuration C is *potential* if there is a set $\{C_1, \dots, C_k\}$ of proposed configurations such that $C = C_0 \sqcup^{\mathcal{C}} (\bigsqcup^{\mathcal{C}}_{i=1,\dots,k} C_i)$ (with C_0 the initial configuration). A configuration C is said to be *superseded* as soon as a process learns a state $(*, C')$ with $C \sqsubseteq^{\mathcal{C}} C'$ and $C \neq C'$. At any moment of time, a configuration is *active* if it is a potential but not yet superseded configuration. Intuitively, some quorum of a configuration should remain "reachable" as long as the configuration is active.

We say that a replica r is *active* when it is a member of an active configuration C, i.e., $r \in member(C)$. A replica is *correct* if, from some point on, it is forever active and not failed. A *client* is correct if it does not fail while executing a *propose* operation.

A configuration C is *available* if some set of replicas in $quorums(C)$ contains only correct processes. In arguing liveness in this paper, we assume the following:

- **Configuration availability.** Any potential configuration that is never superseded must be available.

Therefore, if a configuration is superseded by a strictly larger (w.r.t. $\sqsubseteq^{\mathcal{C}}$) one, then it does not have to be available, i.e., we can safely remove some replicas from it for maintenance.

Liveness properties. In a constantly reconfigured system, we may not be able to ensure liveness to all operations. A slow client can be always behind the active configurations: its set of estimated potential configurations can always be found to constitute a superseded configuration. Therefore, for liveness, we assume that only finitely many reconfigurations occur. Otherwise, only lock-freedom may be provided.

Therefore, to get a reconfigurable object, we replace the liveness property of lattice agreement with the following one:

- **Reconfigurable Liveness.** In executions with finitely many distinct proposed configurations, every *propose* operation invoked by a correct client eventually returns.

Thus, the desired liveness guarantees are ensured as long as only finitely many distinct configurations are proposed. However, the clients are free to perform infinitely many *object* updates without making any correct client starve.

Formally, *reconfigurable lattice agreement* defined on $(\mathcal{L}, \sqsubseteq) = (\mathcal{O} \times \mathcal{C}, \sqsubseteq^{\mathcal{O}} \times \sqsubseteq^{\mathcal{C}})$ satisfies the Validity and Consistency properties of lattice agreement (see Section 2) and the Reconfigurable Liveness property above.

Furthermore, we can only guarantee liveness to clients assuming that, eventually, every correct system participant (client or replica) is informed of the currently active configuration. It boils down to ensuring that an eventually consistent reconfigurable memory is available to store the greatest learnt configuration.

For simplicity, we assume that a reliable broadcast primitive [11] is available, ensuring that (i) every broadcast message was previously broadcast, (ii) if a correct process broadcasts a message m, then it eventually delivers m, and (iii) every message delivered by a correct process is eventually delivered by every correct process. Note that *Configuration availability* implies that an active configuration is either available or sufficiently responsive to be superseded.

5 Reconfigurable lattice agreement: implementation

We now present our main technical result, reconfigurable (generalized) lattice agreement. This algorithm will then be used to implement reconfigurable objects.

Overview. The algorithm for every process p (client or replica) is presented in Algorithm 1. We assume that all procedures (including sub-calls to the *updateState* procedure) are executed by p *sequentially* until they terminate or get interrupted by the wait condition in line 9.

Every process (client or server) p maintains a *state* variable $v_p \in \mathcal{L}$ storing its local estimate of the greatest committed object ($v_p.O$) and configuration ($v_p.C$) states, initialized to the initial element of the lattice (O_0, C_0). We say that a state is *committed* if a process broadcasts it in line 13. Note that all learnt states are previously committed (either directly by the learning process or indirectly by another process). Every process p also maintains T_p, the set of *pending input* configuration states, i.e., known input configuration states that are not superseded by the committed state estimate v_p. For the object lattice, processes stores in obj_p the join of all known proposed objects states.

Algorithm 1 Reconfigurable universal construction: code for process p.

```
Local variables:
seq_p, initially 0          {The number of issued requests }
v_p, initially (O_0, C_0)   {The last learnt state }
T_p, initially ∅            {The set of proposed configuration states }
obj_p, initially O_0        {The candidate object state }

operation propose(prop)     {Propose a new state prop }     {Executed by a client }
1    updateState(v_p, prop.O, {prop.C})
2    learnLB := ⊥
3    while true do
4        seq_p := seq_p + 1
5        oldCommit := v_p          {Archive commit estimate }
6        oldCandidates := (obj_p, T_p)     {Archive candidate states }
7        V := {⊔^C({v_p.C} ∪ S) | S ⊆ T_p}     {Queried configurations }
8        send ⟨(REQ, seq_p), (v_p, obj_p, T_p)⟩ to ⋃_{u∈V} members(u)
9        wait until oldCommit.C ≠ v_p.C or ∀u ∈ V, received responses of the type
                 ⟨(RESP, seq_p), _⟩ from some Q ∈ quorums(u)
10       if oldCommit.C = v_p.C ∧ oldCandidates = (_, T_p) then      {Stable configurations }
11           if learnLB = ⊥ then learnLB = (obj_p, ⊔^C({v_p.C} ∪ T_p))
12           if oldCandidates = (obj_p, _) then     {No greater object received }
13               broadcast ⟨COMMIT, (obj_p, ⊔^C({v_p.C} ∪ T_p))⟩
14               return (obj_p, ⊔^C({v_p.C} ∪ T_p))
15       if learnLB ≠ ⊥ ∧ learnLB ⊑ v_p then return v_p       {Adopt learnt state }

upon receive ⟨msgType, msgContent⟩ from process q
16   updateState(msgContent)       {Update tracked states }
17   if msgType = (REQ, seq) then send ⟨(RESP, seq), (v_p, obj_p, T_p)⟩ to q

upon deliver ⟨COMMIT, state⟩
18   updateState(state, state.O, ∅)      {Adopt the committed state }

procedure updateState(v, s_O, S_C)      {Merge tracked states }
19   v_p := v_p ⊔ v        {Update the commit estimate }
20   obj_p := obj_p ⊔^O s_O       {Update the object candidate }
21   T_p := {u ∈ (T_p ∪ S_C) | u ⋢^C v_p.C}      {Update and trim input candidates }
```

To propose a new state, *prop*, client p updates its local variables through the *updateState* procedure using its input object and configuration states, $prop.O$ and $prop.C$ (line 1). The client p then enters a while loop where it sends a *request* containing (v_p, obj_p, T_p) and equipped with its current sequence number seq_p to all replicas from *every possible join* of pending input configurations with the commit estimate configuration state (line 7). Then p waits until either (1) a greater committed configuration is discovered from the quorum responses or the underlying reliable broadcast (line 18), or (2) a quorum of members of every reached configuration responds with messages of the type $\langle(RESP, seq_p), (v, s_O, S_C)\rangle$, where (v, s_O, S_C) corresponds to the replica updated triple (v_p, obj_p, T_p) (lines 8–9).

When a process (client or replica) p receives a new request or response of the type $\langle msgType, (v, s_O, S_C)\rangle$ or delivers a new committed state (line 18), it updates its commit estimate and its object candidate by joining its current values with the one received or delivered. The process also merges its set of pending input configurations T_p with the

received pending input configurations, except for those superseded by the updated commit estimate (lines 19–21). Every replica also sends a response containing the updated triple (v_p, obj_p, T_p) to the sender of the request (line 17).

If responses from quorums of all accessed configurations are received and no response contains a *new* (not yet known) pending input configuration or a *greater* object state, then the couple formed by obj_p and the join of the commit estimate configuration with all pending input configurations, i.e., $\bigsqcup^{\mathcal{C}}(\{v.C\} \cup T_p)$, is broadcasted and then returned as the new learnt state (lines 12-14). Otherwise, the client proceeds to a new round of the while loop.

To ensure wait-freedom, we integrate a *helping mechanism* consisting of having the clients adopt their committed state estimate (line 15). But, to know when a committed state can be returned, the client must first complete a communication round without interference from reconfigurations (line 11). After such a round, a committed state greater than the join of all known states, stored in *learnLB*, can safely be returned.

Correctness. We give here a short intuition of why our algorithm indeed implements reconfigurable lattice agreement (we refer the reader to [24] for a complete proof). We show first that the *updateState* function is actually a *join* operator on a lattice defined on the triples (v_p, obj_p, T_p), and that the join operator is consistent with the lattice partial order on $\mathcal{L}$ for the projection $(obj_p, \bigsqcup^{\mathcal{C}}(\{v_p.C\} \cup T_p))$. This way we can see our algorithm as the classical (generalized) lattice agreement [15] where the proposed elements are joined until no incompatibles values are received. However, there are some additional subtleties here.

First, for the validity property, we require that every learnt state includes all states that were learnt before the corresponding *propose* operation started. Proving this property is similar to proving the consistency property, i.e., that all learnt states are totally ordered by $\sqsubseteq$. In both cases, we need to show that all committed states are transitively related to each other by $\sqsubseteq$ and the real-time partial order (precedence ordering). Intuitively, we get this by showing that two clients proposing operations either went through quorums of the same configuration or that one client was already aware of a committed state greater than the other client's committed state. This transitive property is the base of the safety proof.

Finally, to argue reconfigurable liveness, we show first that the algorithm is "lock-free", i.e., at least one correct client is making progress, and then show that this client helps other clients to complete their operations. Lock-freedom relies on the underlying broadcast mechanism and the property of *configuration availability* (Section 4): no correct client may be indefinitely blocked waiting for responses to a request. The broadcast mechanism ensures that correct clients are eventually aware of the greatest state committed so far. Hence, as finitely many distinct configurations are proposed, eventually all correct clients will query the same configurations, which must be available. Hence, all states proposed by a correct process will eventually be included in all learnt states and thus a non-committing correct process cannot starve as it will eventually adopt a committed state.

▶ **Theorem 1.** *Algorithm 1 implements reconfigurable lattice agreement.*

6 Reconfigurable objects

In this section, we use our reconfigurable lattice agreement (RLA) abstraction to construct an interval-linearizable reconfigurable implementation of any L-ADT L. The proofs of this section are relatively straightforward and can be found in the technical report [24].

6.1 Defining and implementing reconfigurable L-ADTs

Let us consider two L-ADTs, an *object* L-ADT $L^{\mathcal{O}} = (A^{\mathcal{O}}, B^{\mathcal{O}}, (\mathcal{O}, \sqsubseteq^{\mathcal{O}}, \sqcup^{\mathcal{O}}), O_0, \tau^{\mathcal{O}}, \delta^{\mathcal{O}})$ and a *configuration* L-ADT $L^{\mathcal{C}} = (A^{\mathcal{C}}, B^{\mathcal{C}}, (\mathcal{C}, \sqsubseteq^{\mathcal{C}}, \sqcup^{\mathcal{C}}), C_0, \tau^{\mathcal{C}}, \delta^{\mathcal{C}})$ (Section 2).

The corresponding *reconfigurable L-ADT* implementation, defined on the composition $L = L^{\mathcal{O}} \times L^{\mathcal{C}}$, exports operations in $A^{\mathcal{O}} \times A^{\mathcal{C}}$. It must be interval-linearizable (respectively to $\mathcal{S}_L$) and ensure Reconfigurable Liveness (under the configuration availability assumption).

In the reconfigurable implementation of L, presented in Algorithm 2, whenever a process invokes an operation $a \in A^{\mathcal{O}}$, it proposes a state, $\tau^{\mathcal{O}}(O_p, a)$ – the result from applying a to the last learnt state (initially, C_0) – to RLA, updates (O_p, C_p) and returns the response $\delta^{\mathcal{O}}(O_p, a)$ corresponding to the new learnt state. Similarly, to update the configuration, the process applies its operation to the last learnt configuration and proposes the resulting state to RLA.

Algorithm 2 Interval-linearizable implementation of L-ADT $L = L^{\mathcal{O}} \times L^{\mathcal{C}}$: code for process p.

Shared: *RLA*, reconfigurable lattice agreement
Local variables:
  O_p, initially O_0 { *The last learnt object state* }
  C_p, initially C_0 { *The last learnt configuration state* }
upon invocation of $a \in A^{\mathcal{O}}$ { *Object operation* }
1 $(O_p, C_p) := RLA.propose((\tau^{\mathcal{O}}(O_p, a), C_p))$
2 **return** $\delta^{\mathcal{O}}(O_p, a)$
upon invocation of $a \in A^{\mathcal{C}}$ { *Reconfiguration* }
3 $(O_p, C_p) := RLA.propose((O_p, \tau^{\mathcal{C}}(C_p, a)))$
4 **return** $\delta^{\mathcal{C}}(C_p, a)$

▶ **Theorem 2.** *Algorithm 2 is a reconfigurable implementation of an L-ADT.*

In the special case, when the L-ADT is *update-query*, the construction above produces a *linearizable* implementation:

▶ **Theorem 3.** *Algorithm 2 is a reconfigurable linearizable implementation of an update-query L-ADT.*

6.2 L-ADT examples

We provide four examples of L-ADTs that allow for interval-linearizable (Theorem 2) and linearizable (Theorem 3) reconfigurable implementations.

Max-register. The *max-register* sequential object defined on a totally ordered set $(V, \leq_V)$ provides operations $writeMax(v)$, $v \in V$, returning a default value $\bot$, and $readMax$ returning the largest value written so far (or $\bot$ if there are no preceding writes). We can define the type as an update-query L-ADT as follows:

$$MR_V = (writeMax(v)_{v \in V} \cup \{readMax\}, V \cup \{\bot\}, (V \cup \{\bot\}, \leq_V, max_V), \bot, \tau_{MR_V}, \delta_{MR_V}).$$

where $\leq_V$ is extended to $\bot$ with $\forall v \in V : \bot \leq_V v$, $\delta_{MR_V}(z, a) = z$ if $a = readMax$ and $\bot$ otherwise, and $\tau_{MR_V}(z, a) = max_V(z, v)$ if $a = writeMax(v)$ and z otherwise.

It is easy to see that $(V \cup \{\bot\}, \leq_V, max_V)$ is a join semi-lattice and the L-ADT MR_V satisfies the sequential *max-register* specification.

Set. The (add-only) *set* sequential object defined using a countable set V provides operations $addSet(v)$, $v \in V$, returning a default value $\bot$, and $readSet$ returning the set of all values added so far (or $\emptyset$ if there are no preceding add operation). We can define the type as an update-query L-ADT as follows:

$$Set_V = (addSet(v)_{v \in V} \cup \{readSet\}, 2^V \cup \{\bot\}, (2^V, \subseteq, \cup), \emptyset, \tau_{Set}, \delta_{Set}).$$

where $\subseteq$ and $\cup$ are the usual operators on sets, $\delta_{Set}(z, a) = z$ if $a = readSet$ and $\bot$ otherwise, and $\tau_{Set}(z, a) = z \cup \{v\}$ if $a = addSet(v)$ and z otherwise.

It is easy to see that $(2^V, \subseteq, \cup)$ is a join semi-lattice and the L-ADT Set_V satisfies the sequential (add-only) *set* specification.

Abort flag. An *abort-flag* object stores a boolean flag that can only be raised from $\bot$ to $\top$. Formally, the LADT AF is defined as follows:

$$AF = \big(\{abort, check\}, \{\bot, \top\}, (\{\bot, \top\}, \sqsubseteq^{AF}, \sqcup^{AF}), \bot, \tau_{AF}, \delta_{AF}\big)$$

where $\bot \sqsubseteq^{AF} \top$, $\tau_{AF}(z, abort) = \delta_{AF}(z, abort) = \top$, and $\tau_{AF}(z, check) = \delta_{AF}(z, check) = z$.

Conflict detector. The *conflict-detector* abstraction [5] exports operation $check(v)$, $v \in V$, that may return *true* ("conflict"), or *false* ("no conflict"). The abstraction respects the following properties:

- If no two *check* operations have different inputs, then no operation can return *true*.
- If two *check* operations have different inputs, then they cannot both return *false*.

A conflict detector can be specified as an L-ADT defined as follows:

$$CD = \big(check(v)_{v \in V}, \{true, false\}, (V \times \{\top, \bot\}, \sqsubseteq^{CD}, \sqcup^{CD}), \bot, \tau_{CD}, \delta_{CD}\big)$$

where

- $\bot \sqsubseteq^{CD} \top$; $\forall v \in V$, $\bot \sqsubseteq^{CD} v$ and $v \sqsubseteq^{CD} \top$; $\forall v, v' \in V$, $v \neq v' \Rightarrow v \not\sqsubseteq^{CD} v'$;
- $\tau_{CD}(z, check(v)) = v$ if $z = \bot$ or $z = v$, and $\tau_{CD}(z, check(v)) = \top$ otherwise;
- $\delta_{CD}(z, check(v)) = true$ if $z = \top$ and *false* otherwise.

Also, we can see that $v \sqcup^{CD} v' = v'$ if $v = v'$ or $v = \bot$, and $\top$ otherwise.

▶ **Theorem 4.** *Any interval-linearizable implementation of CD is a conflict detector.*

7 Applications

Many ADTs do not have commutative operations and, thus, do not belong to L-ADT. Moreover, many distributed programming abstractions do not have a sequential specification at all and, thus, cannot be defined as ADTs, needless to say as L-ADTs.

However, as we show, certain such objects can be implemented from L-ADT objects. As L-ADTs are naturally composable, the resulting implementations can be seen as using a single (composed) L-ADT object. By using a reconfigurable version of this L-ADT object, we obtain a reconfigurable implementation. In our constructions we omit talking about reconfigurations explicitly: to perform an operation on the configuration component of the system state, a process simply proposes it to the underlying RLA (see, e.g., Algorithm 2).

Our examples here are atomic snapshots [1] and commit-adopt [17]. The proofs of this section and a discussion of safe agreement [10] are delegated to the technical report [24].

Atomic snapshots

An m-position atomic-snapshot memory maintains an array of m positions and exports two operations, $update(i, v)$, where $i \in \{1, \dots, m\}$ is a location in the array and $v \in V$ – the value to be written, that returns a predefined value ok and $snapshot()$ that returns an m-vector of elements in V. Its sequential specification stipulates that every $snapshot()$ operation returns a vector that contains, in each index $i \in \{1, \dots, m\}$, the value of the last preceding *update* operation on the ith position (or a predefined initial value, if there is no such *update*).

Registers using $MR_{\mathbb{N}\times V}$. We first consider the special case of a single register (1-position atomic snapshot). We describe its implementation from a *max-register*, assuming that the set of values V is totally-ordered with relation $\leq^V$. Let $\leq^{reg}$ be a total order on $\mathbb{N} \times V$ (defined lexicographically, first on $\leq$ and then, in case of equality, on $\leq^V$). Let MR be a max-register defined on $(\mathbb{N} \times V, \leq^{reg})$.

The idea is to associate each written value *val* with a *sequence number seq* and to store them in MR as a tuple (seq, val). To execute an operation $update(v)$, the process first reads MR to get the "maximal" sequence number s written to MR so far. Then it writes $(s + 1, v)$ back to MR. Notice that multiple processes may use $s + 1$ in their *update* operations, but only for concurrent operations. Ties are then broken by choosing the maximal value in the second component in the tuple. A *snapshot* operation simply reads MR and returns the value in the tuple.

Using any reconfigurable linearizable implementation of MR (Theorem 3), we obtain a reconfigurable implementation of an atomic (linearizable) register. Intuitively, all values returned by *snapshot* (read) operations on MR can be totally ordered based on the corresponding sequence numbers (ties broken using $\leq^V$), which gives the order of *reads* in the corresponding sequential history S.

Atomic snapshots. Our implementation of an m-position atomic snapshot (depicted in Algorithm 3) is a straightforward generalization of the register implementation described above. Consider the L-ADT defined as the product of m max-register L-ADTs. In particular, the partial order of the L-ADT is the product of m (total) orders $\leq^{snap}$: $\leq^{reg_1} \times \dots \times \leq^{reg_m}$.

We also enrich the interface of the type with a new query operation *readAll* that returns the vector of m values found in the m max-register components. Note that the resulting type is still an update-query L-ADT, and thus, by Theorem 3, we can use a reconfigurable linearizable implementation of this type, let us denote it by *MRset*.

To execute $update(v, i)$ on the implemented atomic snapshot, a process performs a read on the ith component of *MRset* to get sequence number s of the returned tuple and perfroms $writeMax((s + 1, v))$ on the ith component. To execute a snapshot, the process performs *readAll* on *MRset* and returns the vector of the second element of each item of the array.

Similarly to the case of a single register, the results of all *snapshot* operations can be totally ordered using the $\leq^{snap}$ total order on the returned vectors. Placing the matching *update* operation accordingly, we get an equivalent sequential execution that respects the atomic snapshot specification.

▶ **Theorem 5.** *Algorithm 3 implements an m-component MWMR atomic snapshot.*

Algorithm 3 Simulation of an m-component atomic snapshot using an L-ADT.

operation $update(i, v)$ $\{update\ register\ i\ with\ v\ \}$
1 $(s, -) := MRset[i]$.readMax
2 $MRset[i]$.writeMax$((s + 1, v))$

operation $snapshot()$
3 $r := MRset.readAll$
4 **return** $snap$ **with** $\forall i \in \{1, \ldots, m\}, r[i] = (-, snap[i])$

The Commit-Adopt Abstraction

Let us take a more elaborated example, the commit-adopt abstraction [17]. It is defined through a single operation $propose(v)$, where v belongs to some input domain V. The operation returns a couple $(flag, v)$ with $v \in V$ and $flag \in \{commit, adopt\}$, so that the following conditions are satisfied:

- **Validity:** If a process returns $(_, v)$, then v is the input of some process.
- **Convergence:** If all inputs are v, then all outputs are $(commit, v)$.
- **Agreement:** If a process returns $(commit, v)$, then all outputs must be of type $(_, v)$.

We assume here that V, the set of values that can be proposed to the commit-adopt abstraction, is totally ordered. The assumption can be relaxed at the cost of a slightly more complicated algorithm (by replacing the max register with a set object for example).

Our implementation of (reconfigurable) commit-adopt uses a *conflict-detector* object CD (used to detect distinct proposals), a max-register MR_V (used to write non-conflicting proposals), and an *abort flag* object AF.

Our commit-adopt implementation is presented in Algorithm 4. In its *propose* operation, a process first accesses the *conflict-detector* object CD (line 1). Intuitively, the conflict detector makes sure that committing processes share a common proposal.

If the object returns *false* (no conflict detected), the process writes its proposal in the max-register MR_V (line 2) and then checks the abort flag AF. If the check operation returns $\bot$, then the proposed value is returned with the *commit* flag (line 4). Otherwise, the same value is returned with the *adopt* flag (line 3).

If a conflict is detected (CD returns *true*), then the process executes the *abort* operation on AF (line 6). Then the process reads the *max-register*. If a non-$\bot$ value is read (some value has been previously written to MR), the process adopts that value (line 9). Otherwise, the process adopts its own proposed value (line 8).

▶ **Theorem 6.** *Algorithm 4 implements commit-abort.*

8 Related Work

Lattice agreement. Attiya et al. [8] introduced the (one-shot) lattice agreement abstraction and, in the shared-memory context, described a wait-free reduction of lattice agreement to atomic snapshot. Falerio et al. [15] introduced the long-lived version of lattice agreement (adopted in this paper) and described an asynchronous message-passing implementation of lattice agreement assuming a majority of correct processes, with $\mathcal{O}(n)$ time complexity (in terms of message delays) in a system of n processes. Our RLA implementation in Section 5 builds upon this algorithm.

Algorithm 4 Commit-adopt implementation using L-ADTs.

operation *propose*(v)
1 **if** $CD.check(v) = false$ **then** {*check conflicts* }
2 $MR_V.writeMax(v)$
3 **if** $AF.check = \top$ **then return** ($adopt, v$) {*adopt the input* }
4 **else return** ($commit, v$) {*commit proposal* }
5 **else** {*Try to abort in case of conflict* }
6 $AF.abort$ {*raise abort flag* }
7 $val := MR_V$.readMax
8 **if** $val = \bot$ **then return** ($adopt, v$) {*adopt the input* }
9 **else return** ($adopt, val$) {*adopt the possibly committed value* }

CRDT. Conflict-free replicated data types (CRDT) were introduced by Shapiro et al. [29] for eventually synchronous replicated services. The types are defined using the language of join semi-lattices and assume that type operations are partitioned in updates and queries. Falerio et al. [15] describe a "universal" construction of a linearizable CRDT from lattice agreement. Skrzypczak et al. [30] argue that avoiding consensus in such constructions may bring performance gains. In this paper, we consider a more general class of types (L-ADT) that are "state-commutative" but not necessarily "update-query" and leverage the recently introduced criterion of interval-linearizability [12] for *reconfigurable* implementations of L-ADTs using RLA.

Reconfiguration. *Passive reconfiguration* [7, 9] assumes that replicas enter and leave the system under an explicit *churn model*: if the churn assumptions are violated, consistency is not guaranteed. In the *active reconfiguration* model, processes explicitly propose configuration updates, e.g., sets of new process members. Early proposals, such as RAMBO [20] focused on read-write storage services and used consensus to ensure that the clients agree on the evolution of configurations.

Recent solutions [2, 3, 18, 23, 31] propose an asynchronous reconfiguration by replacing consensus with weaker abstractions capturing the minimal coordination required to safely modify the system configuration. Moreover, Freestore [3] proposes a modular solution to derive interchangeable consensus-based and asynchronous reconfiguration.

Asynchronous reconfiguration. Dynastore [2] was the first solution emulating a reconfigurable atomic read/write register without consensus: clients can asynchronously propose incremental additions or removals to the system configuration. Since proposals commute, concurrent proposals are collected together without the need of deciding on a total order. Assuming n proposals, a Dynastore client might, in the worst case, go through 2^{n-1} candidate configurations before converging to a final one. Assuming a run with a total number of configurations m, complexity is $\mathcal{O}(min(mn, 2^n))$.

SmartMerge [23] allows for reconfiguring not only the system membership but also its quorum system, excluding possible undesirable configurations. SmartMerge brings an interesting idea of using an external reconfiguration service based on lattice agreement [15], which allows us to reduce the number of traversed configurations to $O(n)$. However, this solution assumes that this "reconfiguration lattice" is always available and non-reconfigurable (as we showed in this paper, lattice agreement is a powerful tool that can itself be used to implement a large variety of objects).

Gafni and Malkhi [18] proposed the *parsimonious speculative snapshot* task based on the commit-adopt abstraction [17]. Reconfiguration, built on top of the proposed abstraction, has complexity $\mathcal{O}(n^2)$: n for the traversal and n for the complexity of the parsimonious speculative snapshot implementation. Spiegelman, Keidar and Malkhi [31] improved this work by proposing an optimal solution with time complexity $\mathcal{O}(n)$ by obtaining an amortized (per process) time complexity $\mathcal{O}(1)$ for speculative snapshots operations.

9 Concluding Remarks

To conclude, let us briefly discuss the complexity of our solution to the reconfiguration problem and give an overview of how our solution could be further extended.

Round-trip complexity. The main complexity metric considered in the literature is the maximal number of communication round-trips needed to complete a reconfiguration when c operations are concurrently proposed. In the worst case, each time a round of requests is completed in our algorithm, a new state is affecting T_p or obj_p, and hence we have at most c round-trips. Note that a round might be interrupted by receiving a greater committed state at most c times as committed states are totally ordered joins of proposed states. We are aware of only one other optimal solution with linear round-trip complexity, proposed Spiegelman et al. [31]. In their solution, the maximal number of round-trips is at least $4c$, twice more than ours. This has to do with the use of a shared memory simulation preventing to read and write at the same time and preventing from sending requests to distinct configurations in parallel. Moreover, they also use a similar interruption mechanism.

Querying multiple configurations at the same time might increase the round-trip delay as we need to wait for more responses. Still, we believe that when the number of requests scales with a constant factor, this impact is negligible.

Message complexity. As in earlier solutions, messages are of linear size in the number of distinct proposed configurations or collect operations on the implemented object.

The number of exchanged messages depends on the configuration lattice. With at most k members per configuration, each client may send at most $k * 2^n$ messages per round as there are, in the worst case, exponentially many configurations to query. But this upper bound may be reached only if joins of proposed configurations do not share replicas. We expect, however, that in most cases the concurrently proposed configurations have large overlaps: configuration updates are typically gradual. For example, when a configuration is defined as a set of updates (added and removed replicas), clients may send at most $k + \Delta * n$ requests per round, where Δ the number of replicas added per proposal. For small Δ, the total number of messages is of order k.

An interesting question is whether we can construct a composite complexity metric that combines the number of messages a process sends and the time it takes to complete a *propose* operation. Indeed, one may try to find dependencies between accessing few configurations sequentially versus accessing many configurations in parallel.

Complexity trade-offs. If the cost of querying many configurations in parallel outweigh the cost of contacting fewer configurations sequentially, we can use the approach from [31]. Intuitively, it boils down to solving an instance of generalized lattice agreement on the configurations and then querying the produced configurations, there can be $O(c)$ of them, where c is the number of concurrently proposed configurations.

Objects with "well-structured" its lattices can be implemented very efficiently. Take, for example, the totally ordered lattice of a max-register. In this case, processes can directly return the state stored in *LearnLB* in line 11. Indeed, not returning a committed state might only violate the *consistency* property. But if states are already totally ordered, then the *consistency* property always holds. Therefore, in the absence of reconfiguration calls, operations can return in a single round trip. It is in general interesting to investigate how the lattice structure might be leveraged.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic Snapshots of Shared Memory. *jacm*, 40(4):873–890, 1993.
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 James Aspnes, Hagit Attiya, and Keren Censor. Max Registers, Counters, and Monotone Circuits. In *PODC*, pages 36–45, 2009.
5 James Aspnes and Faith Ellen. Tight Bounds for Adopt-Commit Objects. *Theory Comput. Syst.*, 55(3):451–474, 2014. doi:10.1007/s00224-013-9448-1.
6 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing Memory Robustly in Message Passing Systems. *Journal of the ACM (JACM)*, 42(2):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 Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC*, pages 91–100, 1993.
11 Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011.
12 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Unifying Concurrent Objects and Distributed Tasks: Interval-Linearizability. *J. ACM*, 65(6):45:1–45:42, 2018.
13 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.
14 Gregory V. Chockler, Rachid Guerraoui, Idit Keidar, and Marko Vukolic. Reliable Distributed Storage. *IEEE Computer*, 42(4):60–67, 2009.
15 Jose Faleiro, Sriram Rajamani, Kaushik Rajan, Ganesan Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In *PODC*, pages 125–134, 2012.
16 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with one Faulty Process. *jacm*, 32(2):374–382, 1985.
17 Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *PODC*, pages 143–152, 1998.
18 Eli Gafni and Dahlia Malkhi. Elastic Configuration Maintenance via a Parsimonious Speculating Snapshot Solution. In *DISC*, pages 140–153, 2015.
19 David K. Gifford. Weighted Voting for Replicated Data. In *SOSP*, pages 150–162, 1979.
20 Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Comput.*, 23(4):225–272, 2010.

21 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):123–149, 1991.

22 Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

23 Leander Jehl, Roman Vitenberg, and Hein Meling. SmartMerge: A New Approach to Reconfiguration for Atomic Storage. In *DISC*, pages 154–169, 2015.

24 Petr Kuznetsov, Thibault Rieutord, and Sara Tucci-Piergiovanni. Reconfigurable Lattice Agreement and Applications. *CoRR*, abs/1910.09264, 2019. `arXiv:1910.09264`.

25 Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

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

27 Matthieu Perrin. Concurrency and Consistency. In *Distributed Systems*. Elsevier, 2017. `doi:10.1016/B978-1-78548-226-7.50008-2`.

28 Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

29 Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In *SSS*, pages 386–400, 2011.

30 Jan Skrzypczak, Florian Schintke, and Thorsten Schütt. Linearizable State Machine Replication of State-Based CRDTs without Logs. *CoRR*, abs/1905.08733, 2019. `arXiv:1905.08733`.

31 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic Reconfiguration: Abstraction and Optimal Asynchronous Solution. In *DISC*, pages 40:1–40:15, 2017.