

Reconfigurable Lattice Agreement and Applications

Petr Kuznetsov, Thibault Rieutord, Sara Tucci-Piergiovanni

► **To cite this version:**

Petr Kuznetsov, Thibault Rieutord, Sara Tucci-Piergiovanni. Reconfigurable Lattice Agreement and Applications. [Research Report] Institut Polytechnique Paris; CEA List. 2019. cea-02321547

HAL Id: cea-02321547

<https://hal-cea.archives-ouvertes.fr/cea-02321547>

Submitted on 21 Oct 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



1 Reconfigurable Lattice Agreement and 2 Applications

3 **Petr Kuznetsov**

4 LTCI, Télécom Paris, Institut Polytechnique Paris

5 petr.kuznetsov@telecom-paris.fr

6 **Thibault Rieutord**

7 CEA LIST, PC 174, Gif-sur-Yvette, 91191, France

8 thibault.rieutord@cea.fr

9 **Sara Tucci-Piergiovanni**

10 CEA LIST, PC 174, Gif-sur-Yvette, 91191, France

11 sara.tucci@cea.fr

12 — Abstract —

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

22 **2012 ACM Subject Classification** Theory of computation → Design and analysis of algorithms
23 → Distributed algorithms

24 **Keywords and phrases** Reconfigurable services, lattice agreement

25 **Digital Object Identifier** 10.4230/LIPIcs...

26 **1** Introduction

27 A decentralized service [6, 14, 24, 27] runs on a set of fault-prone *servers* that store replicas
28 of the system state and run a synchronization protocol to ensure consistency of concurrent
29 data accesses. In the context of a storage system exporting read and write operations,
30 several proposals [2, 3, 18, 20, 23, 30] came out with a reconfiguration interface that allows the
31 servers to join and leave, while ensuring consistency of the stored data. Early proposals [20]
32 were based on using *consensus* [16, 21] to ensure that replicas *agree* on the evolution of
33 the system membership. Consensus, however, is expensive and difficult to implement, and
34 recent solutions [2, 3, 18, 23, 30] replace consensus with weaker abstractions capturing the
35 minimal coordination required to safely change the servers configuration. These solutions,
36 however, lack of a uniform way of deriving reconfigurable versions of static objects.

37 **Reconfiguration lattices.** In this paper, we propose a universal construction for a large
38 class of objects. Unlike a consensus-based reconfiguration proposed earlier for generic state-
39 machine replication [25], our construction is asynchronous, at the expense of assuming a
40 restricted object behavior. More precisely, we assume that the set \mathcal{L} of the object's states
41 can be represented as a (join semi-) *lattice* $(\mathcal{L}, \sqsubseteq)$, where \mathcal{L} is partially ordered by the binary
42 relation \sqsubseteq such that for all elements of $x, y \in \mathcal{L}$, there exists the *least upper bound* in \mathcal{L} ,



© Author: Please provide a copyright holder;

licensed under Creative Commons License CC-BY

Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

denoted $x \sqcup y$, where \sqcup is an associative, commutative, and idempotent binary operator on \mathcal{L} . Many important data types, such as atomic snapshots, sets and counters, as well as useful concurrent abstractions, such as commit-adopt [17], can be expressed this way. Intuitively, $x \sqcup y$ can be seen as a *merge* of two alternatively proposed updated states x and y . As long as an implementation of the object ensures that all “observable” states are ordered by \sqsubseteq , it cannot be distinguished from an atomic object.

Consider, for example, the *max-register* [4] data type which exports two operations: *writeMax* that writes values and *readMax* that 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 concurrent 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 \sqsubseteq).

Reconfigurable lattice agreement. The observation above inspires an elegant approach to build reconfigurable objects. In this paper, we introduce 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 merge of configurations, possibly concurrently proposed by different processes, 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 these diverging configurations to safely implement lattice agreement.

We assume that a configuration is a set of servers provided with a quorum system [19], i.e., a set system ensuring the intersection property¹ and, possibly, other configuration 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 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, 30]).

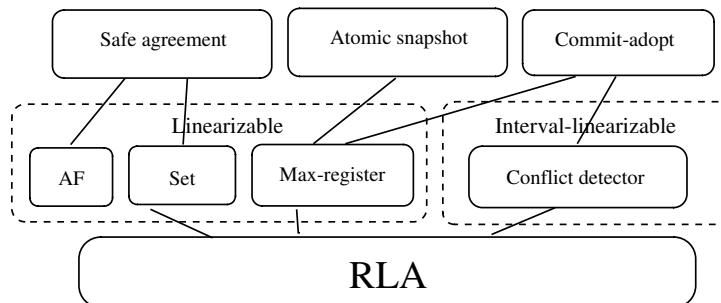
Reconfigurable L-ADT and applications. We show that our reconfigurable lattice agreement, defined on a product of a *configuration lattice* and an *object lattice*, immediately implies reconfigurable versions of many sequential types, such as *max-register* and *conflict detector*. More generally, any *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 to specify 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. When 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 is also linearizable².

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 This class of “update-query” L-ADTs is known as state-based convergent replicated data types (CvRDT) [28]. These types include *max-register*, *set* and *abort flag* (a new type introduced in this

86 We then use our reconfigurable implementations of *max-register*, *conflict detector*, *set*
 87 and *abort-flag* to devise reconfigurable versions of *atomic snapshot* [1], *commit-adopt* [17]
 88 and *safe agreement* [10]. Figure 1 shows how are constructions are related.



■ **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 max-registers and conflict detector, we construct a commit-adopt abstraction, and on top of set and abort flag, we implement safe agreement.

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

- 90 ■ Asynchronous, unlike consensus-based solutions [13,20,25], and not assuming an external
- 91 lattice agreement service [23];
- 92 ■ Uniformly applicable to a large class of objects, unlike existing reconfigurable systems
- 93 that either focus on read-write storage [2,18,20,23] or require data type-specific imple-
- 94 mentations of exported reconfiguration interfaces [30];
- 95 ■ Allowing for a straightforward composition of reconfigurable objects;
- 96 ■ Maintaining configurations with abstract *quorum systems* [19], not restricted to *majority-*
- 97 *based* quorums [2,18];
- 98 ■ Exhibiting optimal time complexity and message complexity comparable with the best
- 99 known implementations [2,23,30];
- 100 ■ Logically separating *clients* (external entities that use the implemented service) from
- 101 *servers* (entities that maintain the service and can be reconfigured).

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

104 **Roadmap.** The rest of the paper is organized as follows. We give basic model definitions
 105 in Section 2. In Section 3, we define our type of reconfigurable objects, followed by the
 106 related notion of reconfigurable lattice agreement in Section 4. In Section 5, we describe
 107 our implementation of reconfigurable lattice agreement, and, in Section 6, we show how to
 108 use it to implement a reconfigurable L-ADT object. In Section 7 we describe some possible
 109 applications. Then, we conclude in Section 8 with an overview of the related work.

110 2 Definitions

111 **Replicas and clients.** Let Π be a (possibly infinite) set of potentially participating pro-
 112 cesses. A subset of the processes, called *replicas*, are used to maintain the implemented

113 object. A process can also act as a *client*, proposing operations on the implemented object
 114 and system reconfigurations. Replicas and clients are subject to crash faults: a process *fails*
 115 when it prematurely stops taking steps of its algorithm. A process is *correct* if it never fails.

116 **Abstract data types.** An abstract data type (*ADT*) is defined as a tuple $T =$
 117 $(A, B, Z, z_0, \tau, \delta)$. Here A and B are countable sets called the *inputs* and *outputs*. Z is
 118 a countable set of abstract object *states*, $z_0 \in Z$ being the initial state of the object. The
 119 map $\tau : Z \times A \rightarrow Z$ is the *transition function*, specifying the effect of an input on the object
 120 state and the map $\delta : Z \times A \rightarrow B$ is the *output function*, specifying the output returned for
 121 a given input and object local state. The input represents an operation with its parameters,
 122 where (i) the operation can have a side-effect that changes the abstract state according to
 123 transition function τ and (ii) the operation can return values taken in the output B , which
 124 depends on the state in which it is called and the output function δ (for simplicity, we only
 125 consider deterministic types here, check, e.g., [26], for more details.)

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

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

$$131 \quad z_0, I_1, R_1, z_1, I_2, R_2, z_2 \dots, I_m, R_m, z_m,$$

132 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
 133 *interval-sequential specification* is a set of interval-sequential histories.

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

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

142 **Lattice agreement.** An abstract (join semi-)lattice is a tuple $(\mathcal{L}, \sqsubseteq)$, where \mathcal{L} is a set
 143 partially ordered by the binary relation \sqsubseteq such that for all elements of $x, y \in \mathcal{L}$, there
 144 exists the least upper bound for the set $\{x, y\}$. The least upper bound is an associative,
 145 commutative, and idempotent binary operation on \mathcal{L} , denoted by \sqcup and called the *join*
 146 *operator* on \mathcal{L} . We write $x \sqsubset y$ whenever $x \sqsubseteq y$ and $x \neq y$. With a slight abuse of notation,
 147 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 .

148 Notice that two lattices $(\mathcal{L}_1, \sqsubseteq_1)$ and $(\mathcal{L}_2, \sqsubseteq_2)$ naturally imply a *product* lattice $(\mathcal{L}_1 \times$
 149 $\mathcal{L}_2, \sqsubseteq_1 \times \sqsubseteq_2)$ with a product join operator $\sqcup = \sqcup_1 \times \sqcup_2$. Here for all $(x_1, x_2), (y_1, y_2) \in$
 150 $\mathcal{L}_1 \times \mathcal{L}_2$, $(x_1, x_2) \sqsubseteq (\sqsubseteq_1 \times \sqsubseteq_2)(y_1, y_2)$ if and only if $x_1 \sqsubseteq_1 y_1$ and $x_2 \sqsubseteq_2 y_2$.

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

157 ■ **Validity.** If a *propose*(v) operation returns a value v' then v' is a join of some proposed
 158 values including v and all values learnt before the invocation of the operation.

- 159 ■ **Consistency.** The learnt values are totally ordered by \sqsubseteq .
- 160 ■ **Liveness.** Every *propose* operation invoked by a correct process eventually returns.
- 161 **A historical remark.** The original definition of long-lived lattice agreement [15] separates
 162 “receive” events and “learn” events. Here we suggest a simpler definition that represents the
 163 two events as the invocation and the response of a *propose* operation. This also allows us
 164 to slightly strengthen the validity condition so that it accounts for the *precedence* relation
 165 between *propose* operations. As a result, we can directly relate lattice agreement to lineariz-
 166 able [22] and interval-linearizable [12] implementations, without introducing artificial “nop”
 167 operations [15].

168 3 Lattice Abstract Data Type

169 In this section, we introduce a class of types that we call *lattice abstract data types* or
 170 *L-ADT*. In an *L-ADT*, the set of states forms a join semi-lattice with a partial order \sqsubseteq^Z .
 171 A lattice object is therefore defined as a tuple $L = (A, B, (Z, \sqsubseteq^Z, \sqcup^Z), z_0, \tau, \delta)$.³ Moreover,
 172 the transition function δ must comply with the partial order \sqsubseteq^Z , that is $\forall z, a \in Z \times A$:
 173 $z \sqsubseteq^Z \tau(z, a)$, and the composition of transitions must comply with the join operator, that
 174 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
 175 that the transition function is “commutative”.

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

- 178 ■ there exists a special “dummy” response \perp (z_0 may also be used) such that $\forall u \in U, z \in Z$,
 179 $\delta(u, z) = \perp$, i.e., updates do not return informative responses;
- 180 ■ $\forall q \in Q, z \in Z, \tau(u, z) = z$, i.e., queries do not modify the states.

181 This class of types is also known as a state-based convergent replicated data types
 182 (CvRDT) [28]. Typical examples of update-query L-ADTs are *max-register* [4] (see Sec-
 183 tion 1) or *sets*. Note that any (L-)ADT can be transformed into an update-query (L-)ADT
 184 by “splitting its operations” into an update and a query (see [26]).

185 **Composition of L-ADTs.** The composition of two ADTs $T = (A, B, Z, z_0, \tau, \delta)$ and $T' =$
 186 $(A', B', Z', z'_0, \tau', \delta')$ is denoted $T \times T'$ and is equal to $(A + A', B \cup B', Z \times Z', (z_0, z'_0), \tau'', \delta'')$;
 187 where $A + A'$ denotes the disjoint union and where τ'' and δ'' apply, according to the domain
 188 A or A' of the input, either τ and δ or τ' and δ' on their respecting half of the state (see [26]).

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

193 **Configurations as L-ADTs.** The reconfiguration service can similarly defined as follows.
 194 Let us define a *configuration L-ADT* as a tuple $(A^C, B^C, (\mathcal{C}, \sqsubseteq^C, \sqcup^C), C_0, \tau^C, \delta^C)$. For each
 195 element C of the *configuration lattice* \mathcal{C} , the input set A includes the operation *members()*,
 196 such that $\delta^C(C, \text{members}()) \subseteq \Pi$, and the operation *quorums()* such that $\delta^C(C, \text{quorums}())$
 197 is $\subseteq 2^{\delta^C(C, \text{members}())}$, a *quorum system*, where every two subsets in $\delta^C(C, \text{quorums}())$ have
 198 a non-empty intersection. In the following we will denote these two operations, with a
 199 slight abuse of notation, as *members(C)* and *quorums(C)*. Here C_0 is called the *initial*
 200 *configuration*.

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

201 For example, \mathcal{C} can be the set of tuples (In, Out) , where $In \subseteq \Pi$ is a set *activated*
 202 processes, and $Out \subseteq \Pi$ is a set of *removed* processes. Then $\sqsubseteq^{\mathcal{C}}$ can be defined as the
 203 piecewise set inclusion on (In, Out) . The set of members of (In, Out) will simply be $In - Out$
 204 and the set of quorums (pairwise-intersecting subsets of $In - Out$), e.g., all majorities of
 205 $In - Out$. Operations in $A^{\mathcal{C}}$ can be $add(s)$, $s \in \Pi$, that adds s to the set of activated processes
 206 and $remove(s)$, $s \in \Pi$, that adds s to the set of removed processes of a configuration. One
 207 can easily see that updates “commute” and that the type is indeed an L-ADT. Let us note
 208 that L-ADTs allow for more expressive reconfiguration operations than simple $adds$ and
 209 $removes$, e.g., maintaining a minimal number of members in a configuration or adapting the
 210 quorum system dynamically, as studied in detail by Jehl et al. in [23].

211 **Interval-sequential specifications of L-ADTs.** Let $L = (A, B, (Z, \sqsubseteq^Z, \sqcup^Z), z_0, \tau, \delta)$ be
 212 an L-ADT. As τ “commutes”, the state reached after a sequence of transitions is order-
 213 independent. Hence, we can define a natural interval-sequential specification of L , \mathcal{S}_L , as
 214 the set of interval-sequential histories $z_0, I_1, R_1, z_1, I_2, R_2, z_2, \dots, I_m, R_m, z_m$ such that:

- 215 ■ $\forall i = 1, \dots, m, z_i = \bigsqcup_{a \in I_{i-1}}^Z \tau(a, z_{i-1})$, i.e., every state z_i is a join of operations in I_{i-1}
 216 applied to z_{i-1} .
- 217 ■ $\forall i = 1, \dots, m, \forall r \in R_i, r = \delta(a, z_i)$, where a is the matching invocation operation for
 218 r , i.e., every response in R_i is based on the result of the corresponding input applied to
 219 state z_i .

220 4 Reconfigurable lattice agreement: definition

221 We define a reconfigurable object as a composition of two L-ADTs, an *object* L-
 222 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). Our main tool is the reconfigurable lattice agreement,
 223 a generalization of lattice agreement operating on the product $(\mathcal{L}, \sqsubseteq) = (\mathcal{O} \times \mathcal{C}, \sqsubseteq^{\mathcal{O}} \times \sqsubseteq^{\mathcal{C}})$
 224 with the product join operator $\sqcup = \sqcup^{\mathcal{O}} \times \sqcup^{\mathcal{C}}$. We say that \mathcal{L} is the set of *states*. For a state
 225 $u = (O, C) \in \mathcal{L}$, we use notations $u.O = O$ and $u.C = C$.

227 When a process p invokes $propose((O, C))$, we say p *proposes* object state O and config-
 228 uration $\{C\} \in \mathcal{C}$.

229 We say that p *learns* an object state O' and a configuration C' if its *propose* invocation
 230 returns (O', C') .

231 The idea is to maintain replicas of a reconfigurable object on active members of installed
 232 but not yet superseded configurations. Formally, we say that a proposed configuration C
 233 is *installed* as soon as some process learns $(*, C')$ such that $C \sqsubseteq^{\mathcal{C}} C'$. A configuration C
 234 is *available* if some set in $quorums(C)$ contains only correct processes. A configuration is
 235 *superseded* as soon some process learns a state $(*, C')$ such that $C \sqsubseteq^{\mathcal{C}} C'$ and $C \neq C'$.

236 In a constantly reconfigured system, we may not be able to ensure liveness to all opera-
 237 tions. A slow client can be always behind the installed and not superseded configuration: the
 238 set of servers it believes to be currently active can always be found to constitute a superseded
 239 configuration. Therefore, for liveness, we assume that only finitely many reconfigurations
 240 occur.

241 Moreover, we require that any join of proposed configurations that is never superseded
 242 must be available:

- 243 ■ **Configuration availability.** Let C_1, \dots, C_k be proposed configurations such that $C =$
 244 $\bigsqcup_{i=1, \dots, k}^{\mathcal{C}} C_i$ is never superseded. Then C is available.

245 Therefore, any configuration constructed as a join of proposed configurations
 246 and “superseded” by a strictly larger (w.r.t. \sqsubseteq^C) configuration does not have to be available,
 247 so it can safely remove some servers for maintenance. In the rest of the paper, we implicitly
 248 assume configuration availability in arguing liveness.

249 As a client may not be aware of the current installed and not superseded configuration,
 250 we can only guarantee liveness to slow clients assuming that, eventually, every *correct* system
 251 participant (client or replica) is informed of the currently active configuration. Here we need
 252 to amend the notion of a correct process, having a reconfigurable system in mind.

253 We say a replica joins the system when the first configuration it belongs to is proposed,
 254 and leaves the system when the first configuration it does not belong to is learnt. Now a
 255 replica is called *correct* if it joined the system and never failed or left. A *client* is correct if
 256 it does not fail while executing its propose operation.

257 We assume that a reliable broadcast primitive [11] is available, ensuring that (i) if a
 258 correct process broadcasts a message, then it eventually delivers it and (ii) every message
 259 delivered by a correct process is eventually delivered by every correct process.

260 To get a reconfigurable object, we therefore replace the liveness property of lattice agree-
 261 ment with the following one:

262 ■ **Reconfigurable Liveness.** In executions with finitely many distinct proposed config-
 263 urations, every *propose* operation invoked by a correct client eventually returns.

264 Note that the desired liveness guarantees are ensured as long as only finitely many distinct
 265 *configurations* are proposed. However, the clients are free to perform infinitely many *object*
 266 updates without making any correct process starve.

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

270 5 Reconfigurable lattice agreement: implementation

271 We now present our main technical result, a reconfigurable implementation of generalized
 272 lattice agreement. This algorithm will then be used to implement reconfigurable objects.

273 **Overview.** The algorithm is specified by the pseudocode of Figure 2. Note that we as-
 274 sume that all procedures (including sub-calls to the *updateState* procedure) are executed
 275 *sequentially* until they terminate or get interrupted by the wait condition in line 9.

276 In the algorithm, every process (client or server) p maintains a *state* variable $v_p \in \mathcal{L}$
 277 storing its local estimate of the greatest committed object ($v_p.O$) and configuration ($v_p.C$)
 278 states, initialized to the initial element of the lattice (O_0, C_0). We say that a state is
 279 committed if a process broadcasted it in line 13. Note that all learnt states are committed
 280 (possibly indirectly by another process), but a process may fail before learning its committed
 281 value. Every process p also maintains T_p , the set of *active input* configuration states, i.e.,
 282 input configuration states that are not superseded by the committed state estimate v_p . For
 283 the object lattice, processes stores in obj_p the join of all known proposed objects states.

284 To propose *prop*, client p update its local variables using the *updateState* procedure using
 285 its input object and configuration states, *prop.O* and *prop.C* (line 1). Clients then enter a
 286 while loop where they send *requests* associated with their current sequence number seq_p
 287 and containing the triplet (v_p, obj_p, T_p) , to all replicas from *every possible join* of active
 288 base configurations and wait until either (1) they get interrupted by discovering a greater
 289 committed configuration through the underlying reliable broadcast, or (2) for each possible

290 join of active base configurations, a quorum of its replicas responded with messages of the
 291 type $\langle (resp, seq_p), (v, s_O, S_C) \rangle$, where (v, s_O, S_C) correspond to the replica updated values of
 292 its triplet (v_p, obj_p, T_p) (lines 8–9).

293 Whenever a process (client or replica) p receives a new request, response or broadcast
 294 of the type $\langle msgType, (v, s_O, S_C) \rangle$, it updates its commit estimate and object candidate by
 295 joining its current values with the one received in the message. It also merge its set of input
 296 configurations T_p with the received input configurations, but the values superseded by the
 297 updated commit estimate are trimmed off T_p (lines 18–20). For replicas, they also send a
 298 response containing the updated triplet (v_p, obj_p, T_p) to the sender of the request (line 17).

299 If responses from quorums of all queried configurations are received and no response
 300 contained a *new*, not yet known, input configuration or a greater object state, then the couple
 301 formed by obj_p and the join the commit estimate configuration with all input configurations
 302 $\bigsqcup^C (\{v.C\} \cup T_p)$ is broadcasted and returned as the new learnt state (lines 12–14). Otherwise,
 303 clients proceed to a new round.

304 To ensure wait-freedom, we integrate a helping mechanism simply consisting in having
 305 clients adopt their committed state estimate (line 15). But, to know when a committed state
 306 is great enough to be returned, clients must first complete a communication round without
 307 interference from reconfigurations (line 11). After such a round, the join of all known states,
 308 stored in *learnLB*, can safely be used as lower bound to return a committed value.

20 **Correctness.** Let us first show that elements of the type $(v, s_O, S_C) \in \mathcal{L} \times \mathcal{O} \times 2^C$ in which
 21 we have that $\forall u \in S_C, u \not\sqsubseteq^C v.C$ admits a partial order \sqsubseteq^* defined as follows:

$$22 \quad (v, s_O, S_C) \sqsubseteq^* (v', s'_O, S'_C) \Leftrightarrow v \sqsubseteq v' \wedge s_O \sqsubseteq^O s'_O \wedge \{u \in S_C \mid u \not\sqsubseteq^C v'.C\} \subseteq S'_C.$$

23 Note that, since \sqsubseteq and \sqsubseteq^O are partial orders, the reflexivity and transitivity properties
 24 are verified if they are verified by the relation $\{u \in S_C \mid u \not\sqsubseteq^C v'.C\} \subseteq S'_C$. Hence, the
 25 symmetry property is trivially verified as for any property \mathcal{P} , we have $\{u \in S_C \mid \mathcal{P}(u)\} \subseteq S_C$.
 26 For transitivity, $(v, s_O, S_C) \sqsubseteq^* (v', s'_O, S'_C)$ and $(v', s'_O, S'_C) \sqsubseteq^* (v'', s''_O, S''_C)$ implies that:

$$27 \quad \{u \in S_C \mid u \not\sqsubseteq^C v''.C\} \subseteq \{u \in \{w \in S_C \mid w \not\sqsubseteq^C v'.C\} \mid u \not\sqsubseteq^C v''.C\} \subseteq \{u \in S'_C \mid u \not\sqsubseteq^C v''.C\} \subseteq S''_C.$$

28 Hence that $(v, s_O, S_C) \sqsubseteq^* (v'', s''_O, S''_C)$. For antisymmetry, given $(v, s_O, S_C) \sqsubseteq^* (v', s'_O, S'_C)$
 29 and $(v', s'_O, S'_C) \sqsubseteq^* (v, s_O, S_C)$, the relations \sqsubseteq and \sqsubseteq^C implies that $v = v'$ and $s_O = s'_O$. But
 30 as by assumption $\forall u \in S_C, u \not\sqsubseteq^C v.C$, we have $S_C = \{u \in S_C \mid u \not\sqsubseteq^C v.C\}$. But since $v = v'$
 31 then $S_C = \{u \in S_C \mid u \not\sqsubseteq^C v'.C\}$, we obtain that $S_C \subseteq S'_C$. Likewise, we have $S'_C \subseteq S_C$, and
 32 thus, we obtain that $S_C = S'_C$, completing the verification of the antisymmetry property.

33 Intuitively, the set of elements $(v, s_O, S_C) \in \mathcal{L} \times \mathcal{O} \times 2^C$, in which we have that $\forall u \in$
 34 $S_C, u \not\sqsubseteq^C v.C$, equipped with the partial order \sqsubseteq^* is a join semi-lattice in which the procedure
 35 *updateState* replaces the triple (v_p, obj_p, T_p) with a join of itself and the procedure argument.
 36 But, we will only prove that the procedure *updateState* replace (v_p, obj_p, T_p) with an upper
 37 bound of itself and the procedure argument (v, s_O, S_C) :

38 **► Lemma 1.** Let $(v_p^{old}, obj_p^{old}, T_p^{old})$ and $(v_p^{new}, obj_p^{new}, T_p^{new})$ be the value of (v_p, obj_p, T_p)
 39 respectively before and after an execution of the *updateState* procedure with argument
 40 (v, s_O, S_C) , then, we have:

$$41 \quad (v_p^{old}, obj_p^{old}, T_p^{old}) \sqsubseteq^* (v_p^{new}, obj_p^{new}, T_p^{new}) \wedge (v, s_O, S_C) \sqsubseteq^* (v_p^{new}, obj_p^{new}, T_p^{new}).$$

42 **Proof.** Let us first note that we can rewrite the operation as follows:

43 ■ Line 18: $v_p^{new} = v_p^{old} \sqcup v$

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 \emptyset { The set of proposed configuration states }
 obj_p , initially O_0 { The candidate object state }

operation *propose(prop)* { Propose a new state *prop* }
1 *updateState*($v_p, prop.O, \{prop.C\}$)
2 *learnLB* := \perp
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 $V := \{\sqcup^C(\{v_p.C\} \cup S) \mid S \subseteq T_p\}$ { Queried configurations }
8 send $\langle (REQ, seq_p), (v_p, obj_p, T_p) \rangle$ to $\bigcup_{u \in V} members(u)$
9 **wait until** $oldCommit.C \neq v_p.C$ or $\forall u \in V$, received responses of the type
 $\langle (RESP, seq_p), _ \rangle$ from some $Q \in quorums(u)$
10 **if** $oldCommit.C = v_p.C \wedge oldCandidates = (_, T_p)$ **then** { Stable configurations }
11 **if** $learnLB = \perp$ **then** $learnLB = (obj_p, \sqcup^C(\{v_p.C\} \cup T_p))$
12 **if** $oldCandidates = (obj_p, _)$ **then** { No greater object received }
13 broadcast $\langle COMMIT, ((obj_p, \sqcup^C(\{v_p.C\} \cup T_p), obj_p, \emptyset)) \rangle$
14 **return** $(obj_p, \sqcup^C(\{v_p.C\} \cup T_p))$
15 **if** $learnLB \neq \perp \wedge learnLB \sqsubseteq v_p$ **then return** v_p { Adopt learnt state }

upon receive $\langle msgType, msgContent \rangle$ from process q

16 *updateState*($msgContent$) { Update tracked states }
17 **if** $msgType = (REQ, seq)$ **then** send $\langle (RESP, seq), (v_p, obj_p, T_p) \rangle$ to q

procedure *updateState*(v, s_O, S_C) { Merge tracked states }
18 $v_p := v_p \sqcup v$ { Update the commit estimate }
19 $obj_p := obj_p \sqcup^O s_O$ { Update the object candidate }
20 $T_p := \{u \in (T_p \cup S_C) \mid u \not\sqsubseteq^C v_p.C\}$ { Update and trim input candidates }

■ **Figure 2** Reconfigurable universal construction: code for process p .

44 ■ Line 19: $obj_p^{new} = obj_p^{old} \sqcup s_O$
45 ■ Line 20: $T_p^{new} = \{u \in (T_p^{old} \cup S_C) \mid u \not\sqsubseteq^C (v_p^{old} \sqcup^C v).C\}$
46 Hence, the use of $(v_p^{old}, obj_p^{old}, T_p^{old})$ and (v, s_O, S_C) are symmetrical. Moreover, it is trivial
47 to check that, w.l.o.g., $(v, s_O, S_C) \sqsubseteq^* (v_p^{new}, obj_p^{new}, T_p^{new})$. Indeed, $v \sqsubseteq v_p^{old} \sqcup v$, $s_O \sqsubseteq^O$
48 $obj_p^{old} \sqcup^O s_O$ and $\{u \in S_C \mid u \not\sqsubseteq^C v_p^{new}.C\} \subseteq \{u \in (T_p^{old} \cup S_C) \mid u \not\sqsubseteq^C (v_p^{old} \sqcup^C v).C\} = T_p^{new}$. ◀

49 Note that it is also trivial to check that initially we have $\forall u \in T_p, u \not\sqsubseteq^C v_p.C$ as $T_p = \emptyset$
50 and that it remains true after a complete execution of the *updateState* procedure as T_p is
51 taken as the set of elements of $(T_p^{old} \cup S_C)$ satisfying this condition.

52 Let us now check that \sqsubseteq^* is a refinement of the order \sqsubseteq for the projection *decide*()
53 defined such that $decide(v, s_O, S_C) = (s_O, \sqcup^C(\{v.C\} \cup S_C))$. Formally:

54 ▶ **Lemma 2.** $(v, s_O, S_C) \sqsubseteq^* (v', s'_O, S'_C) \implies decide(v, s_O, S_C) \sqsubseteq decide(v', s'_O, S'_C)$.

55 **Proof.** This result follows directly from the definition of \sqsubseteq^* . Indeed, as $(v, s_O, S_C) \sqsubseteq^*$
56 (v', s'_O, S'_C) , we have $s_O \sqsubseteq^O s'_O$. Moreover, we have $\{u \in S_C \mid u \not\sqsubseteq^C v'.C\} \subseteq S'_C$. Hence we

57 have $\bigsqcup^c(\{v'.C\} \cup S_C) \sqsubseteq^c \bigsqcup^c(\{v'.C\} \cup S'_C)$. But, as moreover we have $v \sqsubseteq v'$, we obtain
 58 that $\bigsqcup^c(\{v.C\} \cup S_C) \sqsubseteq^c \bigsqcup^c(\{v'.C\} \cup S'_C)$. ◀

59 We are now going to show the main technical result required for the proof of correctness
 60 of Algorithm 2. Consider any run of the algorithm in Figure 2. Let s be any state committed
 61 in the considered run. Let $p(s)$ denote *the first* client that committed s in line 13. Let $V(s)$,
 62 $v(s)$, $obj(s)$ and $T(s)$ denote the value of respectively the variables V , $v_{p(s)}$, $obj_{p(s)}$ and
 63 $T_{p(s)}$ at the moment when $p(s)$ committed s in line 13. Note that, as $p(s)$ passed the tests
 64 in lines 10 and 12, $v_{p(s)}.C$, $obj_{p(s)}$ and $T_{p(s)}$ must have remained unchanged and equal to
 65 respectively $v(s).C$, $obj(s)$ and $T(s)$ since the last computation of V in line 7. In particular,
 66 we have $V(s) = \{\bigsqcup^c(\{v(s).C\} \cup S) \mid S \subseteq T(s)\}$.

67 Let G be the graph whose vertices are all committed states plus $s_0 = (O_0, C_0)$ and whose
 68 edges are defined as follows:

$$69 \quad s \rightarrow s' \Leftrightarrow s \sqsubseteq s' \wedge s.C \in V(s').$$

70 Let us now show that G is connected, i.e., there exists a path between any couple of vertices
 71 in G :

72 ▶ **Lemma 3.** *The graph G is connected.*

73 **Proof.** As \sqsubseteq is a partial order, G is acyclic. Let s be any committed state, we have $v(s).C \in$
 74 $V(s)$ as $v(s).C$ is the value of $v_{p(s)}.C$ used in the computation of $V(s)$ in line 7. Hence,
 75 as $v(s) \sqsubseteq s$ since $s = \bigsqcup^c(\{v(s)\} \cup T(s))$ and as $v(s) \neq s$ since $p(s)$ is the first process to
 76 commit s , any committed state admits a predecessor in G . Thus, the only source of G is s_0 .

77 Let us show that G is connected by contradiction. Hence, let us assume that we can
 78 select s and s' , a *minimal* (w.r.t. \sqsubseteq) pair of vertices of G that are not connected via a path,
 79 i.e., for all couple of vertices $(t, t') \neq (s, s')$ such that $t \sqsubseteq s$ and $t' \sqsubseteq s'$, there is path from t
 80 to t' or from t' to t in G .

81 Let us first show that s and s' share the same set of ancestors in G . Indeed, consider an
 82 ancestor u of s in G . As $u \sqsubseteq s$ and as (s, s') is chosen minimal, there exists a path from u
 83 to s' or from s' to u . There is no path from s' to u as it would imply a path from s' to s .
 84 Hence, u is an ancestor of s' . By symmetry between s and s' , we get that s and s' share the
 85 same set of ancestors in G . Let \bar{s} be a locally maximal (w.r.t. \sqsubseteq) ancestor of s and s' . As
 86 there are no ancestors of s and s' greater than \bar{s} , the paths from \bar{s} to s and s' are edges, i.e.:

$$87 \quad \bar{s}.C \in V(s) \wedge \bar{s}.C \in V(s') \implies \bar{s}.C \in V(s) \cap V(s'),$$

$$88 \quad \bar{s} \sqsubseteq s \wedge \bar{s} \sqsubseteq s'.$$

89 Let us now look back at the algorithm to show that a path must exist from s to s' or
 90 from s' to s . By the algorithm, as $\bar{s}.C \in V(s) \cap V(s')$, in the last round of requests before
 91 committing s (resp. s'), $p(s)$ (resp. $p(s')$) sent a request to all processes in $\bar{s}.C$. As,
 92 in their last round, $p(s)$ and $p(s')$ passed the test of line 10, they received responses from
 93 replicas of $\bar{s}.C$ forming *quorums* in $\bar{s}.C$, hence, as quorums intersect, from a common process
 94 $r \in \bar{s}.C$. Let us assume, w.l.o.g, that, for their last round of requests, r responded to $p(s)$
 95 before responding to $p(s')$.

96 Recall that, as $p(s)$ passed the tests in lines 10 and 12, the values of $v_p.C$, obj_p and T_p
 97 did not change in the last round. Hence the content of the request sent to r by $p(s)$ is equal
 98 to $((v_O, v(s).C), obj(s), T(s))$, with v_O some arbitrary value. By Lemma 1, after r responded
 99 to $p(s)$, (v_r, obj_r, T_r) must become and remain greater or equal to (w.r.t. \sqsubseteq^*) the message
 100 content $((v_O, v(s).C), obj(s), T(s))$. Hence, the latter response to $p(s)$ by r must contain a
 101 greater or equal content, and $(v_{p(s')}, obj_{p(s')}, T_{p(s')})$ becomes and remains greater or equal
 102 to $((v_O, v(s).C), obj(s), T(s))$, thus $((v_O, v(s).C), obj(s), T(s)) \sqsubseteq^* (v(s'), obj(s'), T(s'))$.

103 By Lemma 2, $s = \text{decide}((v_O, v(s).C), \text{obj}(s), T(s)) \sqsubseteq \text{decide}(v(s'), \text{obj}(s'), T(s')) = s'$.
 104 As $v(s')$ is an ancestor of s' , it is an ancestor of s , so $v(s).C \sqsubseteq^C v(s').C \sqsubseteq^C s.C$. Thus:

$$105 \quad s.C = \bigsqcup^C (\{v(s).C\} \cup T(s)) \sqsubseteq^C \bigsqcup^C (\{v(s').C\} \cup T(s)) \sqsubseteq^C \bigsqcup^C (\{s.C\} \cup T(s)) = s.C.$$

106 So $s.C = \bigsqcup^C (\{v(s').C\} \cup T(s))$, and hence, $s.C = \bigsqcup^C (\{v(s').C\} \cup \{u \in T(s), u \not\sqsubseteq^C v(s').C\})$.
 107 From $((v_O, v(s).C), \text{obj}(s), T(s)) \sqsubseteq^* (v(s'), \text{obj}(s'), T(s'))$, we get that $\{u \in T(s), u \not\sqsubseteq^C$
 108 $v(s').C\} \subseteq T(s')$, and therefore, we obtain that:

$$109 \quad s.C = \bigsqcup^C (\{v(s').C\} \cup \{u \in T(s), u \not\sqsubseteq^C v(s').C\}) \in \left\{ \bigsqcup^C (\{v(s').C\} \cup S) \mid S \subseteq T(s') \right\} = V(s').$$

110 We have shown that $s \sqsubseteq s'$ and that $s.C \in V(s')$ and therefore that there is an edge, hence
 111 a path, from s to s' in G — A contradiction. ◀

112 We now have all the main ingredients to show the correctness of Algorithm 2.

113 ▶ **Theorem 4.** *The algorithm in Figure 2 implements reconfigurable lattice agreement.*

114 **Proof.** For the **Consistency** property, Lemma 3 says that G is connected, and hence that
 115 all committed values are totally ordered, thus, that all learnt states are totally ordered.

116 From Lemma 3, we can also infer that $\forall s, s' \in G$, if $s \rightarrow s'$, then the first *propose*
 117 procedure returning s cannot precede the first procedure returning s' . Indeed, In their
 118 last round of requests $p(s)$ and $p(s')$ both queried $s'.C$, as $s'.C \in V(s)$ and $s'.C \in V(s')$,
 119 and received responses from intersecting quorums, hence from a common process r . As
 120 shown in the proof of Lemma 3, this implies that the value committed by the first client
 121 r responded to is smaller than the other. Hence the procedure associated with s cannot
 122 precede the procedure associated with s' . The same argument also holds for any other
 123 *propose* procedure committing s . hence, a client returning in line 14, return a state greater
 124 than all previously committed states, hence all previously learnt states.

125 For a process returning in line 15, to show that learnt states are greater than any preceed-
 126 ing learnt states, it is sufficient to check that *LearnLB* is greater than all. The selected state
 127 *LearnLB* is not a committed state as the value of obj_p may have changed during the round
 128 of requests. But we can say that it is *semi-committed* as configurations did not change. This
 129 part is the most important as it is the property used in Lemma 3 to show that the client
 130 communicating later with the common process r get a greater *decide()* state than the one
 131 committed by the first. Intuitively, this is sufficient to add semi-committed to the graph
 132 and show that there are path from semi-committed states to all smaller committed states,
 133 and hence that it is large enough to be greater than all previously committed, and hence,
 134 learnt states.

135 For the **Validity** property, we have shown that clients return states greater than all
 136 previously learnt states. By a trivial induction, as a committed state is a join of input states
 137 and committed states, it is easy to check that committed states, and hence learnt states,
 138 are joins of the initial state and input states. Moreover, as triples (v_p, obj_p, T_p) becomes and
 139 remains greater after the execution of line 1, then clients commit and set *LearnLB* to states
 140 greater than the procedure proposal. Hence returned states are greater than the procedure
 141 proposal. Therefore the **Validity** property is satisfied.

142 To prove the **Reconfigurable-Liveness** property, consider a run in which only finitely
 143 many distinct *configurations* are proposed. Hence, there exists a greatest learnt configuration
 144 state C_f . By the properties of the reliable-broadcast mechanism (line 13), eventually all

145 correct processes will receive a commit message including C_f . Hence, eventually, all correct
 146 processes will have $v_p.C = C_f$.

147 Assuming **configuration availability**, we have that every join of proposed configura-
 148 tions that are not yet superseded must have an available quorum. Thus, eventually, every
 149 configuration $u.C$ queried by correct processes are available. Therefore, correct processes
 150 cannot be blocked forever waiting in line 9 and, thus, has to perform infinitely many iter-
 151 ations of the while loop. Moreover, since configurations eventually no new configuration is
 152 discovered, all correct processes will eventually always pass the test in line 10 and therefore
 153 set a state for $learnLB$. In a round of requests after setting $learnLB$ based on the triple
 154 (v_l, obj_l, T_l) , the triple (v_r, obj_r, T_r) in all replicas from a quorum of C_f must become and
 155 remain greater (w.r.t \sqsubseteq^*) than (v_l, obj_l, T_l) .

156 Now, let us assume that a correct process p never terminates, thus, it must observe greater
 157 object candidate at each round. This implies that infinitely many *propose* procedures are
 158 initiated, hence that a process commits infinitely many states. A committed state must be
 159 computed based on a triple (v_p, obj_p, T_p) greater than thoses in all received messages, in
 160 particular thoses from a quorum in C_f which must eventually be greater than (v_l, obj_l, T_l) .
 161 Hence, eventually a committed state greater than $learnLB$ is broadcasted, and this value is
 162 adopted and returned by p after receiving it — A contradiction. ◀

163 6 Reconfigurable objects

164 In this section, we use our reconfigurable lattice agreement (RLA) abstraction to construct
 165 an interval-linearizable reconfigurable implementation of any L-ADT L .

166 6.1 Defining and implementing reconfigurable L-ADTs

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

169 The corresponding *reconfigurable L-ADT* implementation, defined on the composition
 170 $L = L^O \times L^C$, exports operations in $A^O \times A^C$. It must be interval-linearizable (respectively
 171 to \mathcal{S}_L) and ensure Reconfigurable Liveness (under the configuration availability assumption,
 172 Section 4).

173 In the reconfigurable implementation of L presented in Figure 3, whenever a process
 174 invokes an operation $a \in A^O$, it proposes a state, $\tau(a, O_p)$ —the result from applying a to
 175 the last learnt state (initially, C_0)—to RLA, updates O_p and returns the response $\delta(a, O_p)$
 176 corresponding to the new learnt state. Similarly, to update the configuration, the process
 177 applies its operation to the last learnt configuration and proposes the resulting state to RLA.

4 ▶ **Theorem 5.** *The algorithm in Figure 3 is a reconfigurable implementation of an L-ADT.*

5 **Proof.** Consider any execution of the algorithm in Figure 3.

6 By the Validity and Consistency properties of the underlying RLA abstraction, we can
 7 represent the states and operations of the execution as a sequence $z_0, I_1, z_1, \dots, I_m, z_m$,
 8 where $\{z_1, \dots, z_m\}$ is the set of learnt values, and each $I_i, i = 1, \dots, m$, is a set of operations
 9 invoked in this execution, such that $z_i = \bigsqcup_{a \in I_i} \tau(a, z_{i-1})$.

10 A construction of the corresponding interval-sequential history is immediate. Consider an
 11 operation a that returned a value in the execution based on a learnt state z_i (line 2). Validity
 12 of RLA implies that $a \in I_j$ for some $j \leq i$. Thus, we can simply add a to set R_i . By repeating
 13 this procedure for every complete operation, we get a history $z_0, I_1, R_1, z_1, \dots, I_m, R_m, z_m$

Shared: *RLA*, reconfigurable lattice agreement

Local:

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

■ **Figure 3** Interval-linearizable implementation of L-ADT $L = L^O \times L^C$: code for process p .

14 complying with \mathcal{S}_L . By construction, the history also preserves the precedence relation of
15 the original history.

16 Reconfigurable liveness of the implementation is implied by the properties of RLA (as-
17 suming reconfiguration availability). ◀

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

20 ▶ **Theorem 6.** *The algorithm in Figure 3 is a reconfigurable linearizable implementation of*
21 *an update-query L-ADT.*

22 **Proof.** Consider any execution of the algorithm in Figure 3 and assume that L is update-
23 query.

24 By Theorem 5, there exists a history $z_0, I_1, R_1, z_1, \dots, I_m, R_m, z_m$ that complies with \mathcal{S}_L ,
25 the interval-sequential specification of L . We now construct a *sequential* history satisfying
26 the *sequential* specification of L as follows:

- 27 ■ For every update u in the history, we match it with immediately succeeding matching
28 response \perp (remove the other response of u if any);
- 29 ■ For every response of a query q in the history we match it with an immediately preceding
30 matching invocation of q (remove the other invocation of q if any);

31 As the updates of an L-ADT are commutative, the order in which we place them in the
32 constructed sequential history S does not matter, and it is immediate that every response
33 in S complies with τ and δ in a sequential history of L . ◀

34 6.2 L-ADT examples

35 We give three examples of L-ADTs that allow for interval-linearizable (Theorem 5) and
36 linearizable (Theorem 6) reconfigurable implementations.

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

$$41 \quad MR_V = (writeMax(v)_{v \in V} \cup \{readMax\}, V \cup \{\perp\}, (V \cup \{\perp\}, \leq_V, max_V), \perp, \tau_{MR_V}, \delta_{MR_V}).$$

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

XX:14 Reconfigurable Lattice Agreement and Applications

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

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

$$50 \quad Set_V = (addSet(v)_{v \in V} \cup \{readSet\}, 2^V \cup \{\perp\}, (2^V, \subseteq, \cup), \emptyset, \tau_{Set_V}, \delta_{Set_V}).$$

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

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

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

$$57 \quad AF = (\{abort, check\}, \{\perp, \top\}, (\{\perp, \top\}, \sqsubseteq^{AF}, \sqcup^{AF}), \perp, \tau_{AF}, \delta_{AF})$$

58 where $\perp \sqsubseteq^{AF} \top$, $\forall z \in \{\perp, \top\} : \top \sqcup^{AF} z = \top$, $\perp \sqcup^{AF} \perp = \perp$, $\tau_{AF}(z, abort) = \delta_{AF}(z, abort) = \top$,
 59 and where $\tau_{AF}(z, check) = \delta_{AF}(z, check) = z$.

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

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

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

$$66 \quad CD = (V, \{true, false\}, (V \times \{\top, \perp\}, \sqsubseteq^{CD}, \sqcup^{CD}), \perp, \tau_{CD}, \delta_{CD})$$

67 where

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

71 Also, we can see that $v \sqcup^Z v' = v'$ if $v = v'$ or $v = \perp$, and \top otherwise.

72 ► **Theorem 7.** *Any interval-linearizable implementation of CD is a conflict detector.*

73 **Proof.** Consider any execution of an interval-linearizable implementation of CD . Let S be
 74 the corresponding interval-sequential history.

75 For any two $check(v)$ and $check(v')$, $v \neq v'$, in S , the response to one of these operations
 76 must appear *after* the invocations of both of them. Hence, one of the outputs must be
 77 computed on a value greater than the join of the two proposals, equal to \top . Therefore, if
 78 both operations return, at least one of the them must return *true*.

79 The state used to compute the output must be a join of some invoked operations, hence
 80 operations can only return *true* if not all *check* operations share the same input. ◀

81 **7 Applications**

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

85 We show, however, that certain such objects can be *implemented from* L-ADT objects.
 86 As L-ADTs are naturally composable, the resulting implementations can be seen as using a
 87 single (composed) L-ADT object. By using a reconfigurable version of this L-ADT object,
 88 we obtain a reconfigurable version of the implemented type. In our implementations we
 89 omit talking about reconfigurations explicitly: to perform an operation on the configuration
 90 component of the system state, a process simply proposes it to the underlying RLA (see,
 91 e.g., Figure 3).

92 Our examples are atomic snapshots [1] and commit-adopt [17].

93 Atomic snapshots

94 An m -sized atomic-snapshot memory maintains an array of m positions and exports two
 95 operations, $update(i, v)$, where $i \in \{1, \dots, m\}$ is a location in the array and $v \in V$ —the value
 96 to be written, that returns a predefined value `ok` and $snapshot()$ that returns an m -vector
 97 of elements in V . Its sequential specification stipulates that every $snapshot()$ operation
 98 returns a vector that contains, in each index $i \in \{1, \dots, m\}$, the value of the last preceding
 99 $update$ operation on the i^{th} position (or a predefined initial value, if there is no such $update$
 100 operation).

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

106 The idea is to associate each written value val with a *sequence number* seq and to store
 107 them in MR as a tuple (seq, val) . To execute an operation $update(v)$, the process first reads
 108 MR to get the “maximal” sequence number s written to MR so far. Then it writes $(s + 1, v)$
 109 back to MR . Notice that multiple processes may concurrently use $s + 1$ in their $update$
 110 operations. Ties are then broken by choosing the maximal value in the second component in
 111 the tuple. However, it is guaranteed that $s + 1$ will be larger than the sequence number used
 112 by any *preceding* $update$ operation. A $snapshot$ operation simply reads MR and returns the
 113 value in the tuple.

114 Using any reconfigurable linearizable implementation of MR (Theorem 6), we obtain
 115 a reconfigurable implementation of an atomic (linearizable) register. Intuitively, all values
 116 returned by $snapshot$ (read) operations on MR can be totally ordered based on the corre-
 117 sponding sequence numbers (ties broken using \leq^V), which gives the order of *reads* in the
 118 corresponding sequential history S .

119 Let $update(v)$ be an operation such that (1) it writes tuple (s, v) to MR and (2) some
 120 read operation returned v after reading (s, v) in MR . We then insert this $update$ operation
 121 in the sequential history S just before the first such read operation (if there are multiple
 122 such $update$ operations, they can be inserted in a batch). Each remaining complete $update$
 123 operation is inserted either just before the first $update$ in the history with a greater couple
 124 of sequence number and value or (if no such $update$ exists) at the end of the history.

125 By construction, S is legal: every read returns the value of the last preceding write.
 126 Moreover, as only concurrent $updates$ can use the same sequence number and the $snapshot$
 127 operations are ordered respecting the sequence numbers, S complies with the real-time
 128 precedence of the original history. We delegate the complete proof to the more general case
 129 of an m -sized snapshot.

130 **Atomic snapshots.** Our implementation of an m -sized atomic snapshot (described in

```

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  $\forall i \in \{1, \dots, m\}, r[i] = (-, \text{snap}[i])$ 

```

■ **Figure 4** Simulation of an m -component atomic snapshot using an L-ADT.

131 Figure 4) is a straightforward generalization of the register implementation above. Consider
 132 the L-ADT defined as the product of m max-register L-ADTs. In particular, the partial
 133 order of the L-ADT is the product of m (total) orders $\leq^{snap}: \leq^{reg_1} \times \dots \times \leq^{reg_m}$.

134 We also enrich the interface of the type with a new query operation *readAll* that returns
 135 the vector of m values found in the m max-register components. Notice that the resulting
 136 type is still an update-query L-ADT, as its (per-component) updates are commutative.

137 By Theorem 6, we can use a reconfigurable linearizable implementation of this type, let
 138 us denote it by *MRset*.

139 Now to execute *update*(v, i) on the implemented atomic snapshot, a process performs a
 140 read on the i^{th} component of *MRset* to get sequence number s of the returned tuple and
 141 performs *writeMax*($s + 1, v$) on the i^{th} component. To execute a snapshot, the process
 142 performs *readAll* on *MR* and returns the array of the second elements in the tuples of the
 143 returned array.

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

4 ▶ **Theorem 8.** *Algorithm in Figure 4 implements an m -component MWMMR atomic snapshot*
 5 *object.*

6 The Commit-Adopt Abstraction

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

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

14 We assume here that V , the set of values that can be proposed to the commit-adopt
 15 abstraction, is totally ordered. The assumption can be relaxed at the cost of a slightly more
 16 complicated algorithm.

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

20 Our commit-adopt implementation is presented in Figure 5. In its *propose* operation,
 21 a process first accesses the *conflict-detector* object *CD* (line 1). Intuitively, the conflict
 22 detector makes sure that committing processes share a common proposal.

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  $val = \perp$  then return ( $adopt, v$ )           { adopt the input }
9      else return ( $adopt, val$ )           { adopt the possibly committed value }
```

■ **Figure 5** Commit-adopt implementation using L-ADTs.

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

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

9 ▶ **Theorem 9.** *Algorithm in Figure 5 implements commit-abort.*

10 **Proof.** The Validity property is trivially satisfied as processes return either their own pro-
 11 posal or the proposal of another process found in the max-register MR_V .

12 To prove Convergence, consider an execution in which all processes share the same input
 13 v . The conflict detector must return false to processes since it is accessed with a unique
 14 input. As no conflict is observed, no process could have called an *abort* operation on AF ,
 15 and hence, the *check* operations on AF can only return \perp . Therefore all processes return
 16 with (*commit*, v).

17 To prove Agreement, suppose, by contradiction, that the algorithm has an execution in
 18 which process p commits value v (line 4) and process q adopts or commits value $v' \neq v$ (in
 19 lines 4, 8 or 9).

20 We observe first that q cannot return in line 4, as otherwise the conflict detector would
 21 return *false* to p or q . For the same reason no value other than v could have been written to
 22 MR_V in this execution. Also, q must have completed line 6 before p checked AF in line 3,
 23 as otherwise p would not be able to commit v in line 4. Thus, q reads MR_V (line 7) *after*
 24 p has written v in it (line 2). Hence, q must have adopted the value read in MR_V (line 9),
 25 and this value must have been v —a contradiction. ◀

26 The Safe-Agreement Abstraction

27 Another popular shared-memory abstraction is *safe agreement* [10]. It is defined through
 28 a single operation *propose*(v), $v \in V$ (we assume that V is totally ordered). The operation
 29 returns a value $v \in V$ or a special value $\perp \notin V$, so that the following conditions are satisfied:

- 30 ■ **Validity:** Every non- \perp output has been previously proposed.
- 31 ■ **Agreement:** All non- \perp outputs are identical.
- 32 ■ **Non-triviality:** If all participating processes return, then at least one returns a non- \perp
- 33 value.

```

operation propose( $v$ )
1   $In.addSet(id)$            { enter the doorway }
2  if  $MR_V.readMax = \perp$  then  $MR_V.writeMax(v)$       { write proposal if empty }
3   $Out.addSet(id)$          { exit the doorway }
4   $outSet := Out.readSet$ 
5   $inSet := In.readSet$ 
6  if  $inSet = outSet$  then return  $MR_V.readMax$       { no process in doorway }
7  else return  $\perp$ 

```

■ **Figure 6** Safe-agreement implementation using L-ADTs for process with identifier id .

34 Our implementation of safe agreement (Figure 6) uses two (add-only) sets denoted In
 35 and Out (Section 6) and a max-register MR_V .

36 The *propose* operation consists of two phases. In the first phase (lines 1-3) that we call
 37 the *doorway protocol*, the process add its identifier to In . Then the process reads MR_V . If
 38 \perp is read, then the process writes its proposal to the max-register, and adds its identifier to
 39 the Out set.

40 In the second phase (lines 4-7), the process first reads Out and then— In . If the two sets
 41 match, then the process reads the max-register again and return the read value. Otherwise,
 42 the special value \perp is returned.

43 Intuitively, the processes use the doorway protocol to ensure that only the first set of
 44 concurrently participating processes may write a value in the max-register. The second phase
 45 of the algorithm checks if there still can be processes poised to write to the max-register,
 46 and return the value of the max-register only if it is not the case.

7 ▶ **Theorem 10.** *Algorithm in Figure 6 implements safe agreement.*

8 **Proof.** The Validity property is trivially satisfied, as any non- \perp returned value must have
 9 been read in the max-register (line 6). As a process can read the max-register only after it
 10 has written its input in it (line 2), every such value must be an input value of some process.

11 To prove Agreement, consider, by contradiction, an execution in which two processes,
 12 p and q , return different non- \perp values. Let p be the first process to read MR_V in line 6.
 13 Thus, the max-register MR_V has been written *after* it has been read (in line 6) by p and
 14 *before* it has been read (in line 6) by q . Let s be the process that performed the first such
 15 write.

16 Notice that before writing its input in MR_V , s must have read \perp in it (line 2). Moreover,
 17 it must have executed line 2 *before* p has finished its doorway: otherwise s would find in
 18 MR_V the value written by p or an earlier written value. Thus, s has already added itself to
 19 the set In when p reads it in line 5. Furthermore, s is still in its doorway at the moment
 20 when p reads MR_V in line 6. In particular, s has not yet added itself to the set Out at that
 21 moment.

22 Thus, when p reaches line 6 its local variables $inSet$ and $outSet$ are not equal. Hence, p
 23 cannot return in line 6—a contradiction.

24 To prove Non-triviality, assume that all participating processes return and let p be the
 25 last process to write to the Out set. By that moment, all participating processes appear
 26 both in In and Out . Thus, p must return the value read in MR_V (line 6), which is non- \perp ,
 27 as p has ensured before that (line 2). ◀

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.

CRDT. Conflict-free replicated data types (CRDT) were introduced by Shapiro et al. [28] 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. [29] argue that avoiding consensus in such constructions may bring performance gains. In this paper, we considered a more general class of types (L-ADT) that are “state-commutative” but not necessarily “update-query” and leveraged 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 proposal, such as RAMBO [20] focused on read-write storage services and used consensus to ensure that the clients agree on the evolution of configurations.

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 $\mathcal{O}(n)$. 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 [30] improved this work by proposing a solution with time complexity $\mathcal{O}(n)$ by obtaining an amortized (per process) time complexity $\mathcal{O}(1)$ for speculative snapshot operations.

71 **9** Concluding Remarks

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

74 **Round-trip complexity.** The main complexity metric considered in the literature is the
75 maximal number of communication round-trips needed to complete a reconfiguration when
76 n operations are concurrently proposed. In our solution, each time a round of requests
77 is completed, a new input state was discovered to modify T_p or obj_p , hence we have at
78 most n round-trips. Note that a round of requests might be interrupted by receiving a
79 greater committed state, at most n times as committed states are totally ordered joins of
80 input states. The only other optimal solution with a linear round-trip complexity is from
81 Spiegelman et al. [30]. In their solution the maximal number of round-trips is at least $4n$,
82 that is twice than us. This has to do with the use of a shared memory simulation preventing
83 to read and a write at the same time and preventing from sending requests to distinct
84 configurations in parallel.

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

88 **Message complexity.** The second metric that is studied in the literature is the number
89 and size of the exchanges messages. In our protocol as in other solutions, messages are of
90 linear size either for the distinct proposed configurations or the use of collect operations on
91 the simulated memories.

92 The number of exchanged messages by our protocol may however greatly vary with the
93 configuration object that is implemented. With at most k members per configuration, each
94 client may send at most $k * 2^n$ messages per round as there is an exponential number of
95 potential configuration to query. But this upper bound may be reached only if joins of
96 proposed configurations do not share any replica. However, defining such configuration
97 objects does not make much sense. In our example replicas may be added or removed, the
98 one used in particular in most proposed solutions, clients may send at most $k + \Delta * n$ requests
99 per round, where Δ the maximal number of replica added per proposal. In this case, the
100 number of requests is comparable with k , the number of messages send to query a single
101 configuration as done for solution based on a shared memory simulation.

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

106 **Optimizations.** If the cost of querying many configurations in parallel outweigh the cost
107 of contacting fewer configurations sequentially, one can proceed to a reconfigurable lattice
108 agreement based on the methodology from [30]. Intuitively, it would consists in solving a
109 generalized lattice agreement on the current configuration before switching the used configu-
110 ration while using a carefully designed tracking mechanism of potentially used configurations.

111 A lighter modification to the RLA protocol may consists in leveraging timing constraints
112 to wait for responses during a delay sufficient to obtain most responses, while waiting for
113 responses from quorums only when no new information is received and an operation may
114 return. Such modification may yield a great efficiency gain in practice as clients should be
115 less constrained by slow responses while increasing the number of distinct inputs expected
116 to discover per round.

117 Improvements can also be made for implemented objects when its lattice is well struc-

118 tured. A pertinent example is the fully ordered lattice states of max registers. For them,
119 processes can directly return the state stored in *LearnLB* in line 11. Indeed, not returning
120 a committed states might only violate the *consistency* property. But if states are totally
121 ordered, then the *consistency* property is necessarily verified. Such a modification would
122 yield to operations in a single round-trip when no reconfiguration occurs. Hence, it might
123 be interesting to further investigate how the lattice structure might be leveraged in general.

124 ——— References ———

- 125 **1** Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of
126 shared memory. *J. ACM*, 40(4):873–890, 1993.
- 127 **2** M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without
128 consensus. *J. ACM*, 58(2):7:1–7:32, 2011.
- 129 **3** E. Alchieri, A. Bessani, F. Greve, and J. da Silva Fraga. Efficient and modular consensus-
130 free reconfiguration for fault-tolerant storage. In *21st International Conference on Princi-*
131 *ples of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017*, pages
132 26:1–26:17, 2017.
- 133 **4** J. Aspnes, H. Attiya, and K. Censor. Max registers, counters, and monotone circuits. In
134 *ACM Symposium on Principles of Distributed Computing, PODC*, pages 36–45, 2009.
- 135 **5** J. Aspnes and F. Ellen. Tight bounds for adopt-commit objects. *Theory of Computing*
136 *Systems*, 55(3):451–474, Oct 2014.
- 137 **6** H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems.
138 *J. ACM*, 42(2):124–142, Jan. 1995.
- 139 **7** H. Attiya, H. C. Chung, F. Ellen, S. Kumar, and J. L. Welch. Emulating a shared register
140 in a system that never stops changing. *IEEE Trans. Parallel Distrib. Syst.*, 30(3):544–559,
141 2019.
- 142 **8** H. Attiya, M. Herlihy, and O. Rachman. Atomic snapshots using lattice agreement. *Dis-*
143 *tributed Computing*, 8(3):121–132, 1995.
- 144 **9** R. Baldoni, S. Bonomi, A. Kermarrec, and M. Raynal. Implementing a register in a dynamic
145 distributed system. In *ICDCS*, pages 639–647, 2009.
- 146 **10** P. Berman and A. A. Bharali. Quick atomic broadcast. pages 189–203. 93.
- 147 **11** C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to reliable and secure distributed*
148 *programming*. Springer Science & Business Media, 2011.
- 149 **12** A. Castañeda, S. Rajsbaum, and M. Raynal. Unifying concurrent objects and distributed
150 tasks: Interval-linearizability. *J. ACM*, 65(6):45:1–45:42, 2018.
- 151 **13** M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM*
152 *Transactions on Computer Systems (TOCS)*, 20(4):398–461, Nov. 2002.
- 153 **14** G. V. Chockler, R. Guerraoui, I. Keidar, and M. Vukolic. Reliable distributed storage.
154 *IEEE Computer*, 42(4):60–67, 2009.
- 155 **15** J. Faleiro, S. Rajamani, K. Rajan, G. Ramalingam, and K. Vaswani. Generalized lattice
156 agreement. In *PODC*, pages 125–134, 2012.
- 157 **16** M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus
158 with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.
- 159 **17** E. Gafni. Round-by-round fault detectors (extended abstract): Unifying synchrony and
160 asynchrony. In *PODC*, 1998.
- 161 **18** E. Gafni and D. Malkhi. Elastic configuration maintenance via a parsimonious speculating
162 snapshot solution. In *DISC*, pages 140–153, 2015.
- 163 **19** D. K. Gifford. Weighted voting for replicated data. In *SOSP*, pages 150–162, 1979.
- 164 **20** S. Gilbert, N. A. Lynch, and A. A. Shvartsman. Rambo: a robust, reconfigurable atomic
165 memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, 2010.

- 166 **21** M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):123–149, 1991.
- 167 **22** M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects.
168 *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
- 169 **23** L. Jehl, R. Vitenberg, and H. Meling. Smartmerge: A new approach to reconfiguration for
170 atomic storage. In *DISC*, pages 154–169, 2015.
- 171 **24** L. Lamport. The Part-Time parliament. *ACM Transactions on Computer Systems*,
172 16(2):133–169, May 1998.
- 173 **25** L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *SIGACT News*,
174 41(1):63–73, 2010.
- 175 **26** M. Perrin. Concurrency and consistency. In *Distributed Systems*. Elsevier, 2017.
- 176 **27** F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A
177 tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.
- 178 **28** M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data
179 types. In *SSS*, pages 386–400, 2011.
- 180 **29** J. Skrzypczak, F. Schintke, and T. Schütt. Linearizable state machine replication of state-
181 based crdts without logs. *CoRR*, abs/1905.08733, 2019.
- 182 **30** A. Spiegelman, I. Keidar, and D. Malkhi. Dynamic reconfiguration: Abstraction and
183 optimal asynchronous solution. In *DISC*, pages 40:1–40:15, 2017.