Information and Computation 285 (2022) 104869

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Store-collect in the presence of continuous churn with application to snapshots and lattice agreement ☆,☆☆

Hagit Attiya [a,*], Sweta Kumari [a], Archit Somani [a], Jennifer L. Welch [b]

[a] Department of Computer Science, Technion, Israel
[b] Department of Computer Science and Engineering, Texas A&M University, United States of America

ARTICLE INFO

Article history:
Received 3 January 2021
Received in revised form 25 November 2021
Accepted 30 January 2022
Available online 3 February 2022

ABSTRACT

We present an algorithm for implementing a store-collect object in an asynchronous crash-prone message-passing dynamic system, where nodes continually enter and leave. The algorithm is very simple and efficient, requiring just one round trip for a store operation and two for a collect. We then show the versatility of the store-collect object for implementing churn-tolerant versions of useful data structures, while shielding the user from the complications of the underlying churn. In particular, we present elegant and efficient implementations of atomic snapshot and generalized lattice agreement objects that use store-collect.

1. Introduction

A popular programming technique that contributes to designing provably-correct distributed applications is to use shared objects for interprocess communication, instead of more low-level mechanisms such as message-passing. Although shared objects are a convenient abstraction, they are not generally provided in large-scale distributed systems; instead, nodes keep copies of the data and communicate by sending messages to keep the copies consistent.

Dynamic distributed systems allow computing nodes to enter and leave the system at will, either due to failures and recoveries, moving in the real world, or changes to the systems' composition, a process called *churn*. Motivating applications include those in peer-to-peer, sensor, mobile, and social networks, as well as server farms. We focus on the situation when the network is always fully connected, which could be due to, say, an overlay network. A broadcast mechanism is assumed through which a node can send a message to all nodes present in the system; the broadcast is not necessarily reliable and a message sent by a failing node may not reach some of the nodes.

The usefulness of shared memory programming abstractions has been long established for static systems (e.g., [4,6]), which have known bounds on the number of fixed computing nodes and the number of possible failures. This success has inspired work on providing the same for newer, dynamic, systems. However, most of this work has shown how to simulate a shared read-write register (e.g., [2,7,11,12,20]). We discuss a couple of exceptions [13,24] below.

☆ Supported by ISF grant 380/18 and NSF grant 1816922.

☆☆ A preliminary version of this paper has appeared in SSS 2020 [10], and the paper was invited to the special issue of the conference. In this version, we have added many proofs that were not present in the conference version, and expanded some of the explanations.

* Corresponding author.
E-mail addresses: hagit@cs.technion.ac.il (H. Attiya), sweta@cs.technion.ac.il (S. Kumari), archit@cs.technion.ac.il (A. Somani), welch@cse.tamu.edu (J.L. Welch).

https://doi.org/10.1016/j.ic.2022.104869
0890-5401/

In this paper, we promote the *store-collect* shared object [8] (defined in Section 2) as a primitive well-suited for dynamic message-passing systems with an ever-changing set of participants. Each node can store a value in a store-collect object with a STORE operation and can collect the latest value stored by each node with a COLLECT operation. Inherent in the specification of this object is an ability to track the set of participants and to read their latest values.

Below we elaborate on three advantageous features of the store-collect object: The store-collect semantics is well-suited to dynamic systems and can be implemented easily and efficiently in them; the widely-used atomic snapshot object can be implemented on top of a store-collect object; and a variety of other commonly-used objects can be implemented either directly on top of a store-collect or on top of an atomic snapshot object. These implementations are simple and inherit the properties of being churn-tolerant and efficient, showing that store-collect combines algorithmic power and efficiency.

A churn-tolerant store-collect object can be implemented fairly easily. We adopt essentially the same system model as in [7], which allows ongoing churn as long as not too many churn events take place during the length of time that a message is in transit. To capture this constraint, there is an assumed upper bound D on the maximum message delay, but no (positive) lower bound. Nodes do not know D and have no local clocks, causing consensus to be unsolvable, even if the system is static [7]. The model differentiates between nodes that crash and nodes that leave; nodes that have entered but not left are considered present even if crashed. The number of nodes that can be crashed at any time is bounded by a fraction of the number of nodes present at that time. During any time interval of length D, the number of nodes entering or leaving is a fraction of the number of nodes present in the system at the beginning of the interval. (See Section 3 for model details.)

Our algorithm for implementing a churn-tolerant store-collect object is based on the read-write register algorithm in [7]. It is simple and efficient: once a node joins, it completes a store operation within one round-trip, and a collect operation within two round-trips. The store-collect object satisfies a variant of the "regularity" consistency condition, which is weaker than linearizability [22]. In contrast to our single-round-trip store operation, the write operation in the algorithm of [7] requires two round trips. Another difference between the algorithms is that in ours, each node keeps a local set of tuples with an entry for each known node and its value instead of a single value; when receiving new information, instead of overwriting the single value, our algorithm merges the new information with the old. One contribution of our work in this paper is a significantly revised proof of the churn management protocol that is much simpler than that in [7], consequently making it easier to build on the results. (See Section 4.)

Building an atomic snapshot on top of a store-collect object is easy! We present a simple algorithm with an elegant correctness proof (Section 6.2). One may be tempted to implement an atomic snapshot in our model by plugging churn-tolerant registers (e.g., [7]) into the original algorithm of [1]. Besides needlessly sequentializing accesses to the registers, such an implementation would have to track the current set of participants. In contrast, using our store-collect object, which encapsulates the changing participants and collects information from them in parallel, yields a simple algorithm very similar in spirit to the original but whose round complexity is linear instead of quadratic in the number of participants. The key subtlety of the algorithm is the mechanism for detecting when a scan can be borrowed in spite of difficulties caused by the churn, in order to ensure termination.

Atomic snapshot objects have numerous uses in static systems, e.g., to build multi-writer registers, concurrent timestamp systems, counters, and accumulators, and to solve approximate agreement and randomized consensus (cf. [1,4]). In addition to analogous applications, we show (Section 6.3) how a churn-tolerant atomic snapshot object can be used to provide a churn-tolerant generalized lattice agreement object [17]. This object supports a PROPOSE operation whose argument is a value belonging to a lattice and whose response is a lattice value that is the join of some subset of all prior input values, including its own argument. Generalized lattice agreement is an extension of (single-shot) *lattice agreement*, well-studied in the static shared memory model [9]. Generalized lattice agreement has been used to implement many objects [15,17], including atomic snapshots [9] and *conflict-free replicated data types* [24,28,29], e.g., linearizable abort flags, sets, and max registers [24].

The store-collect object specification is versatile. Our atomic snapshot and generalized lattice agreement algorithms demonstrate that layering linearizability on top of a store-collect object is easy. Yet not every application needs the costs associated with linearizability, and store-collect gives the flexibility to avoid them. Our approach to providing churn-tolerant shared objects is modular, as the underlying complications of the message-passing and the churn are hidden from higher layers by our store-collect implementation. As evidence, we observe (Section 6.1) that store-collect allows very simple implementations of max-registers, abort flags, and sets, in which an implemented operation takes at most a couple of store and collect operations. The choice of problems and the algorithms follow [24] but the algorithms inherit good efficiency and churn-tolerance properties from our store-collect implementation.

Related work An algorithm that directly implements an atomic snapshot object in a *static message-passing system*, bypassing the use of registers, is presented by Delporte-Gallet et al. [16]. This algorithm includes several nice optimizations to improve the message and round complexities. These include speeding up the algorithm by parallelizing the collect, as is already encapsulated in our store-collect algorithm. Our atomic snapshot algorithm works in a *dynamic* system and has a shorter and simpler proof of linearizability.

Aguilera [3] presents a specification and algorithm for atomic snapshots in a *dynamic model* in which nodes can continually enter and communicate via *shared registers*. This algorithm is then used for group membership and mutual exclusion in that model. Variations of the model were proposed by Gafni, Merritt and Taubenfeld [19,26], who also provided algorithms for election, mutual exclusion, consensus, collect, snapshot, and renaming. Spiegelman and Keidar [27] present atomic snap-

shot algorithms for a crash-prone dynamic system in which processes communicate via shared registers. Their algorithms uniquely identify each scan operation with a version number to help determine when a scan can be borrowed; we use a similar mechanism in our snapshot algorithm. However, our atomic snapshot algorithm uses a shared store-collect object which tolerates *ongoing* churn. Our use of a non-linearizable building block requires a more delicate approach to proving linearizability, as we cannot simply choose, say, a specific write to an atomic register as the linearization point of an update, as done by Spiegelman and Keidar [27].

The problem of implementing shared objects in the presence of *ongoing* churn and crash failures in *message-passing systems* is studied in [11,12], which considers read-write registers, and [13], which considers sets. Unlike our results, these papers assume the system size is restricted to a fixed window and the system is eventually synchronous. Like our algorithms, the set algorithm of Baldoni, Bonomi and Raynal [13] uses unbounded local memory at the nodes.

A popular alternative way to model churn in *message-passing systems* is as a sequence of quorum configurations, each of which consists of a set of nodes and a quorum system over that set (e.g., [2,18,20,21,23,24]). Explicit reconfiguration operations replace older configurations with newer ones. The assumptions made in these papers are incomparable with those made by Attiya et al. [7] and in our paper, as the former assume churn eventually stops while the latter assume the churn is bounded.

Most papers on generalized lattice agreement have assumed static systems (cf. [9,15,17,28,29]. A notable exception is the work of Kuznetsov, Rieutord, and Tucci-Piergiovanni [24], who consider dynamic systems subject to changes in the composition due to reconfiguration. Their paper provides an implementation for a large class of shared objects, including conflict-free replicated data types, that can be modeled as a lattice. By showing how to view the state of the system as a lattice as well, the paper elegantly combines the treatment of the reconfiguration and the operations on the object. Unlike our work, the algorithms of Kuznetsov, Rieutord, and Tucci-Piergiovanni [24] require that changes to the system composition eventually cease in order to ensure progress.

2. The store-collect problem

A shared *store-collect object* [8] supports concurrent *store* and *collect* operations performed by some set of clients. Each operation has an invocation and response. For a *store* operation, the invocation is of the form $\text{STORE}_p(v)$, where v is a value drawn from some set and p indicates the invoking client, and the response is of the form ACK_p, indicating that the operation has completed. For a *collect* operation, the invocation is of the form COLLECT_p and the response is of the form $\text{RETURN}_p(V)$, where V is a *view*, that is, a set of client-value pairs without repetition of client ids. We use the notation $V(p)$ to indicate v if $\langle p, v\rangle \in V$ and $\perp$ if no pair in V has p as its first element.

Informally, the behavior required of a store-collect object is that each *collect* operation should return a view containing the latest value stored by each client. We do not require the *store* and *collect* operations to appear to occur instantaneously, that is, the object is not necessarily linearizable. Instead, we give a precise definition of the required behavior that is along the lines of *interval linearizability* [14] or the specification of *regular* registers [25].

A sequence σ of invocations and responses of *store* and *collect* operations is a *schedule* if, for each client id p, the restriction of σ to invocations and responses by p consists of alternating invocations and matching responses, beginning with an invocation. Each invocation and its matching following response (if present) together make an operation. If the response of operation op comes before the invocation of operation op' in σ, then we say op *precedes* op' (in σ) and op' *follows* op. We assume that every value written in a *store* operation in a schedule is unique (a condition that can be achieved using sequence numbers and client ids).

Given two views V_1 and V_2 returned by two *collect* operations in a schedule σ, we write $V_1 \preceq V_2$ if, for every $\langle p, v_1\rangle \in V_1$, there exists v_2 such that $\langle p, v_2\rangle \in V_2$ such that either $v_1 = v_2$ or the $\text{STORE}_p(v_1)$ invocation does not occur after the response of $\text{STORE}_p(v_2)$ in σ.

A schedule σ satisfies *regularity for the store-collect problem* if:

- For each *collect* operation cop in σ that returns V and every client p, if $V(p) = \perp$, then no *store* operation by p precedes cop in σ. If $V(p) = v \neq \perp$, then there is a $\text{STORE}_p(v)$ invocation that occurs in σ before cop completes and no other *store* operation by p occurs in σ between this invocation and the invocation of cop.
- For every two *collect* operations in σ, cop_1 which returns V_1 and cop_2 which returns V_2, if cop_1 precedes cop_2 in σ, then $V_1 \preceq V_2$.

3. System model

We model each node p as a state machine with a set of states, containing two initial states s_p^i and s_p^ℓ. Initial state s_p^i is used if p is initially in the system, whereas s_p^ℓ is used if p enters the system later. State transitions are triggered by the occurrences of events. Possible triggering events are: node p enters the system (ENTER_p), node p leaves the system (LEAVE_p), node p receives a message m ($\text{RECEIVE}_p(m)$), node p invokes an operation (COLLECT_p or $\text{STORE}_p(v)$), and node p crashes (CRASH_p).

A *step* of a node p is a 5-tuple (s', T, m, R, s) where s' is the old state of p, T is the triggering event, m is the message to be sent, R is a response ($\text{RETURN}_p(V)$, ACK_p, or JOINED_p) or $\perp$, and s is the new state of p. The values of m, R and s are

determined by a transition function applied to s' and T. $\text{RETURN}_p(V)$ is the response to COLLECT_p, ACK_p is the response to STORE_p, and JOINED_p is the response to ENTER_p. If T is CRASH_p, then m is $\perp$ and R is $\perp$.

A *local execution* of a node p is a sequence of steps such that:

- the old state of the first step is an initial state;
- the new state of each step equals the old state of the next step;
- if the old state of the first step is s_p^i, then no ENTER_p event occurs;
- if the old state of the first step is s_p^ℓ, then the triggering event in the first step is ENTER_p and there is no other occurrence of ENTER_p; and
- at most one of CRASH_p and LEAVE_p occurs and if so, it is in the last step.

In our model, a node that leaves the system cannot re-enter with the same id. It can, however, re-enter with a new id. Likewise, a node that crashes does not recover, instead, it may re-enter with a new id.

A point in time is represented by a nonnegative real number. A *timed local execution* is a local execution whose steps occur at nondecreasing times. If a local execution is infinite, the times at which its steps occur must increase without bound. Given a timed local execution of a node, if (s', T, m, R, s) is the step with the largest time less than or equal to t, then s is the *state* of that node *at time* t. A node p is said to be *present* (resp., *a member*) at time t if ENTER_p (resp., JOIN_p) occurs at or before t, or p's initial state is s_p^i, but LEAVE_p does not occur at or before t. The number of nodes that are present at time t is denoted by $N(t)$. A *crashed* node (i.e., a node for which CRASH_p occurs at or before t) is still considered to be present (resp., a member). A node is said to be *active* at time t if it is present and not crashed at t.

An *execution* e is a possibly infinite set of timed local executions, one for each node that is ever present in the system, such that there is a nonempty finite set of nodes p, denoted S_0, whose initial state is s_p^i. These nodes are initially members of the system.

We assume a reliable broadcast communication service that provides nodes with a mechanism to send the same message to all nodes[1] in the system; message delivery is FIFO. However, if a broadcast invocation is the last thing that a node does before crashing, the message is not guaranteed to be received by all the nodes; this is a weaker assumption than that made in [7]. If a message m sent at time t is received by a node at time t', then the *delay* of this message is $t' - t$. This encompasses transmission delay as well as time for handling the message at both the sender and receiver. Let $D > 0$ denote the *maximum message delay* that can occur in the system. Formally:

- Every sent message has at most one matching receipt at each node and every message receipt has exactly one matching message send.
- If a message m is sent by a node p at time t, p's next event is not CRASH_p, and node q is active throughout $[t, t + D]$ (i.e., q enters by time t and does not leave or crash by time $t + D$), then q receives m. The delay of every received message is in $(0, D]$.
- Messages from the same sender are received in the order they are sent (i.e., if node p sends message m_1 before sending message m_2, then no node receives m_2 before it receives m_1). This can be achieved by tagging each message with the id of its sender and a sequence number.

Let $\alpha > 0$ and $0 < \Delta \leq 1$ be real numbers that denote the *churn rate* and *failure fraction*, respectively. Let N_{min} be a positive integer, the minimum system size. The parameters α and Δ are known to the nodes, but N_{min} and D are not. We assume executions satisfy three assumptions:

Churn Assumption For all times $t > 0$, there are at most $\alpha \cdot N(t)$ ENTER and LEAVE events in $[t, t + D]$.
Minimum System Size For all times $t \geq 0$, $N(t) \geq N_{min}$.
Failure Fraction Assumption For all times $t \geq 0$, at most $\Delta \cdot N(t)$ nodes are crashed at time t.[2]

We assume "well-formed" interactions between client threads and their users: An invocation occurs at node p only if p has already joined but has not left or crashed, i.e., p is a member. Furthermore, no previous invocation at p is pending, i.e., at most one operation is pending at each node.

An algorithm is a *correct implementation of a store-collect object* in our model if the following are true for all executions with well-formed interactions:

- For every node $p \notin S_0$, if ENTER_p occurs, then at least one of LEAVE_p, CRASH_p, or JOINED_p occurs subsequently; that is, every node that enters the system and remains active eventually joins. For every node $p \in S_0$, JOINED_p never occurs.

[1] Sending a message to a single recipient can be accomplished by broadcasting the message and indicating the intended recipient so that others will ignore the message.

[2] Since crashed nodes are counted in $N(t)$, there is a limit on how many nodes can leave before new nodes enter in order to maintain the failure fraction assumption.

- For every node p, if $\text{STORE}_p(v)$ (respectively, COLLECT_p) occurs, then at least one of LEAVE_p, CRASH_p, or ACK_p (respectively, $\text{RETURN}_p(V)$) occurs subsequently; that is, every store or collect operation invoked at a node that remains active eventually completes.
- The schedule resulting from the restriction of the execution to the *store* and *collect* invocations and responses satisfies regularity for the store-collect problem.

4. The Continuous Churn Collect (CCC) algorithm

In our algorithm, nodes run *client* threads, which invoke *collect* and *store* operations, and *server* threads. We assume that the code segment that is executed in response to each event executes without interruption.

To track the composition of the system (Algorithm 1), a node p maintains a set *Changes* of events concerning the nodes that have entered the system. When an ENTER_p event occurs, p adds *enter(p)* to its *Changes* set (Line 1) and broadcasts an **enter** message requesting information about prior events (Line 2). When p finds out that another node q has entered the system, either by receiving an **enter** message directly from q or by receiving an **enter-echo** message for q from a third node, it adds *enter(q)* to its *Changes* set (Line 3 or 6). When p receives an **enter** message from a node q, it replies with an **enter-echo** message containing its *Changes* set, its current estimate *LView (local view)* of the state of the simulated object, its flag *is_joined* indicating whether p has joined yet, and the id q (Line 4). The first time that p receives an **enter-echo** in response to its own **enter** message (i.e., one that ends with its own id) from a joined node, it computes *join_threshold*, the number of **enter-echo** messages it needs to get before it can join (Line 9) and increments its *join_counter* (Line 10).

The fraction γ is used to calculate *join_threshold*, the number of **enter-echo** messages that should be received before joining, based on the size of the *Present* set (nodes that have entered, but have not left, see Line 9). Setting γ is a key challenge in the algorithm as setting it too small might not propagate updated information, whereas setting it too large might not guarantee termination of the join.

When the required number of replies to the **enter** message sent by p is received (Line 11), p adds *join(q)* to its *Changes* set, sets its *is_joined* flag to *true* (Line 12), broadcasts a message saying that it has joined (Line 14) and outputs JOINED_p (Line 15). When p finds out that another node q has joined, either by receiving a **join** message directly from q or by receiving a **join-echo** message for q from a third node, it adds *join(q)* to its *Changes* set (Line 16 or 19). When a LEAVE_p event occurs, p broadcasts a **leave** message (Line 21) and halts (Line 22). When p finds out that another node q is leaving the system, either by receiving a **leave** message directly from q or by receiving a **leave-echo** message for q from a third node, it adds *leave(q)* to its *Changes* set (Line 23 or 25).

Initially, node p's *Changes* set equals $\{enter(q) | q \in S_0\} \cup \{join(q) | q \in S_0\}$, if $p \in S_0$, and $\emptyset$ otherwise. Node p also maintains a set of nodes that it believes are present: $Present = \{q | enter(q) \in Changes \wedge leave(q) \notin Changes\}$, i.e., nodes that have entered, but have not left, as far as p knows. Essentially, Algorithm 1 of CCC is the same as CCREG [7] except for Line 5, which merges newly received information with current local information instead of overwriting it.

Once a node has joined, its client thread can handle *collect* and *store* operations (Algorithm 2) and its server thread (Algorithm 3) can respond to clients. The client at node p maintains a derived variable $Members = \{q \mid join(q) \in Changes \wedge leave(q) \notin Changes\}$ of nodes that p considers as members, i.e., nodes that have joined but not left.

Our implementation adds a sequence number, *sqno*, to each value in a view, which is now a set of triples, $\{\langle p, v, sqno\rangle, \ldots\}$, without repetition of node ids. We use the notation $V(p) = v$ if there exists *sqno* such that $\langle p, v, sqno\rangle \in V$, and $\perp$ if no triple in V has p as its first element.

A merge of two views, V_1 and V_2, picks the latest value stored by each node according to the highest *sqno*. If a triple for node p is in V_2 but not in V_1 then the merge includes the triple for p from V_2 as well, and vice versa. That is,

Definition 1. Given two views V_1 and V_2, $merge(V_1, V_2)$ is defined as the subset of $V_1 \cup V_2$ consisting of every triple whose node id is in one of V_1 and V_2 but not the other, and, for node ids that appear in both V_1 and V_2, it contains only the triple with the larger sequence number.

Note that $V_1, V_2 \preceq merge(V_1, V_2)$.

Each node keeps a local copy of the current view in its *LView* variable. In a *collect operation*, a client thread requests the latest value of servers' local views using a **collect-query** message (Line 33). When a server node p receives a **collect-query** message, it responds with its local view (*LView*) through a **collect-reply** message (Line 60) if p has joined the system. When the client receives a **collect-reply** message, it merges its *LView* with the *received view (RView)*, to get the latest value corresponding to each node (Line 35). Then the client waits for sufficiently many **collect-reply** messages before broadcasting the current value of its *LView* variable in a **store** message (Line 40). When server p receives a **store** message with a view *RView*, it merges *RView* with its local *LView* (Line 55) and, if p is joined, it broadcasts **store-ack** (Line 57). The client waits for sufficiently many **store-ack** messages before returning *LView* to complete the *collect* (Line 51); this threshold is recalculated in Line 38 to reflect possible changes to the system composition that the client has observed.

In a *store operation*, a client thread updates its local variable *LView* to reflect the new value by doing a merge (Line 43) and broadcasts a **store** message (Line 46). When server p receives a **store** message with view *RView*, it merges *RView* with its local *LView* (Line 55) and, if p is joined, it broadcasts **store-ack** (Line 57). The client waits for sufficiently many **store-ack** messages before completing the *store* (Line 50).

Algorithm 1 CCC—Common code managing churn, for node p.

Local Variables:
LView: set of (node id, value, sequence number) triples, initially $\emptyset$ // local view
is_joined: Boolean, initially false // true iff p has joined the system
join_threshold: int, initially 0 // number of enter-echo messages needed for joining
join_counter: int, initially 0 // number of enter-echo messages received so far
Changes: set of *enter*(q), *leave*(q), and *join*(q) // active membership events known to p
initially $\{enter(q) | q \in S_0\} \cup \{join(q) | q \in S_0\}$ if $p \in S_0$, and $\emptyset$ otherwise
Derived Variable:
$Present = \{q \mid enter(q) \in \text{Changes} \wedge leave(q) \notin Changes\}$

When ENTER$_p$ occurs:
1: **add** *enter(p)* **to** *Changes*
2: **broadcast** ⟨**enter,** p⟩

When RECEIVE$_p$ ⟨enter, q⟩ occurs:
3: **add** *enter(q)* **to** *Changes*
4: **broadcast** ⟨**enter-echo,***Changes*, *LView*, *is_joined*, q⟩

When RECEIVE$_p$ ⟨enter-echo, C, *RView*, j, q⟩ occurs:
5: *LView* := *merge(LView, RView)*
6: *Changes* := *Changes* ∪ C
7: **if** ¬*is_joined* ∧ ($p = q$) **then**
8: **if** (j = *true*) ∧ (*join_threshold* = 0) **then**
9: *join_threshold* := $\gamma \cdot |Present|$
10: *join_counter*++
11: **if** *join_counter* ≥ *join_threshold* > 0 **then**
12: *is_joined* := *true*
13: **add** *join(p)* **to** *Changes*
14: **broadcast** ⟨**join,** p⟩
15: **return JOINED$_p$**

When RECEIVE$_p$ ⟨join, q⟩ occurs:
16: **add** *join(q)* **to** *Changes*
17: **add** *enter(q)* **to** *Changes*
18: **broadcast** ⟨**join-echo,** q⟩

When RECEIVE$_p$ ⟨join-echo, q⟩ occurs:
19: **add** *join(q)* **to** *Changes*
20: **add** *enter(q)* **to** *Changes*

When LEAVE$_p$ occurs:
21: **broadcast** ⟨**leave,** p⟩
22: **halt**

When RECEIVE$_p$ ⟨leave, q⟩ occurs:
23: **add** *leave(q)* **to** *Changes*
24: **broadcast** ⟨**leave-echo,** q⟩

When RECEIVE$_p$ ⟨leave-echo, q⟩ occurs:
25: **add** *leave(q)* **to** *Changes*

Algorithm 2 CCC—Client code, for node p.

Local Variables:
optype: string, initially $\perp$ // indicates which type of operation (*collect* or *store*) is pending
tag: int, initially 0 // counter to identify currently pending operation by p
threshold: int, initially 0 // number of replies/acks needed for current phase
counter: int, initially 0 // number of replies/acks received so far for current phase
sqno: int, initially 0 // sequence number for values stored by p
Derived Variable:
$Members = \{q | \ join(q) \in Changes \wedge leave(q) \notin Changes\}$

When COLLECT$_p$ occurs:
30: *optype* := *collect; tag*++
31: *threshold* := $\beta \cdot |Members|$
32: *counter* := *0*
33: **broadcast** ⟨**collect-query,** *tag*, p⟩

When RECEIVE$_p$ ⟨collect-reply, *RView*, *t*, q⟩ occurs:
34: **if** (t = *tag*) ∧ ($q = p$) **then**
35: *LView* := *merge(LView, RView)*
36: *counter*++
37: **if** (*counter* ≥ *threshold*) **then**
38: *threshold* := $\beta \cdot |Members|$
39: *counter* := *0*
40: **broadcast** ⟨**store,** *LView, tag, p*⟩

When STORE$_p$(v) occurs:
41: *optype* := *store; tag*++
42: *sqno*++
43: *LView* := *merge(LView,*{⟨$p, v,$ *sqno*⟩})
44: *threshold* := $\beta \cdot |Members|$
45: *counter* := *0*
46: **broadcast** ⟨**store,** *LView, tag, p*⟩

When RECEIVE$_p$ ⟨store-ack, t, q⟩ occurs:
47: **if** ($t = tag$) ∧ ($q = p$) **then**
48: *counter*++
49: **if** (*counter* ≥ *threshold*) **then**
50: **if** (*optype* = *store*) **then return** ACK
51: **else return** *LView*

Algorithm 3 CCC—Server code, for node p.

When RECEIVE$_p$ ⟨store, *RView*,*tag*, q⟩ occurs:
55: *LView* := *merge(LView, RView)*
56: **if** *is_joined* **then**
57: **broadcast** ⟨**store-ack,** *tag*, q⟩
58: **broadcast** ⟨**store-echo,** *LView*⟩

When RECEIVE$_p$ ⟨collect-query, *tag*, q⟩ occurs:
59: **if** *is_joined* **then**
60: **broadcast** ⟨**collect-reply,** *LView*, *tag*, q⟩

When RECEIVE$_p$ ⟨store-echo, *RView*⟩ occurs:
61: *LView* := *merge(LView, RView)*

Table 1
Explanation of notation, and two sets of possible values for the parameters.

Notation	Meaning	set 1	set 2
α	churn rate	0	0.04
Δ	failure fraction	0.21	0.01
β	fraction for store/collect threshold	0.79	0.80
γ	fraction for join threshold	0.79	0.77
N_{min}	minimum system size	≥ 2	≥ 2
$N(t)$	number of nodes present at time t		
S_0	set of nodes initially present and joined		
D	maximum message delay		

The fraction β is used to calculate the number of messages that should be received (stored in local variable *threshold*) based on the size of the *Members* set, for the operation to terminate. Setting β is a key challenge in the algorithm as setting it too small might not return correct information from *collect* or *store*, whereas setting it too large might not guarantee termination of the *collect* and *store*.

We define a *phase* to be the execution by a client node p of one of the following intervals of its code:

- lines 30 through 37, the first part of a *collect* operation,
- lines 38 through 40 and 47 through 51, the second part of a *collect* operation called the "store-back", or
- lines 41 through 50, the entirety of a *store* operation.

The first kind of phase is called a *collect phase* while the second and third kinds are called a *store phase*.

For any completed phase φ executed by node p, define $view(\varphi)$ to be the value of $LView_p^t$, where t is the time at the end of the phase. Since a *store* operation consists solely of a store phase, we also apply the notation to an entire *store* operation.

5. Proof of CCC store-collect algorithm

The correctness of the algorithm relies on the following four constraints. They are stated in terms of a quantity $Z = (1-\alpha)^3 - \Delta \cdot (1+\alpha)^3$, which will be shown to be the fraction of nodes that are guaranteed to remain active throughout an interval of length $3D$.

$$N_{min} \geq \frac{1}{Z + \gamma - (1+\alpha)^3} \tag{A}$$

$$\gamma \leq Z/(1+\alpha)^3 \tag{B}$$

$$\beta \leq Z/(1+\alpha)^2 \tag{C}$$

$$\beta > \frac{(1-Z)(1+\alpha)^5 + (1+\alpha)^6}{((1-\alpha)^3 - \Delta \cdot (1+\alpha)^2)((1+\alpha)^2+1)} \,. \tag{D}$$

Fortunately, there are values for the parameters α, Δ, γ, and β that satisfy these constraints. In the extreme case when $\alpha = 0$ (i.e., no churn), the failure fraction Δ can be as large as 0.21; in this case, it suffices to set both γ and β to 0.79 for any value of N_{min} that is at least 2. As α increases up to 0.04, Δ must decrease approximately linearly until reaching 0.01; in this case, it suffices to set γ to 0.77 and β to 0.80 for any value of N_{min} that is at least 2. Table 1 summarizes the notation we use and lists two sets of possible values for α, Δ, β, γ and N_{min}.

Consider any execution of the algorithm with well-formed interactions. We will show that it satisfies the three properties required for correctness given at the end of Section 3. We begin by analyzing the dynamics of nodes entering, leaving, and crashing in Lemma 1 through Corollary 4. Then, Observation 5 through Lemma 8 capture how a node's *Changes* local variable (and thus *Present* local variable) tracks the ENTER, JOIN, and LEAVE events that occur. Armed with these results, we show in Theorem 9 that an active node eventually joins, which is the first correctness property. Then Theorem 10 states the second correctness property, that every operation invoked by an active node eventually completes. Observation 11 through Lemma 13 describe how a node's *LView* local variable tracks the occurrences of *store* operations, while Lemma 14 describes

how a node's *Members* local variable tracks the ENTER, JOIN, and LEAVE events that occur. Lemma 15 is the key for proving regularity: it states that if a *store* phase precedes a *collect* phase, then the *collect* returns information that is at least as up-to-date as the *store*. Finally, Theorem 16 states the third correctness property, regularity for store-collect.

Lemma 1. *For all $i \in \mathbb{N}$ and all $t \geq 0$,*
(a) at most $((1+\alpha)^i - 1) \cdot N(t)$ nodes enter during $(t, t + i \cdot D]$; and
(b) $N(t + i \cdot D) \leq (1+\alpha)^i \cdot N(t)$.

Proof. The proof is by induction on i.

Basis: $i = 0$. For all t, the interval $(t, t + 0 \cdot D] = (t, t]$ is empty and (a) and (b) are true.

Induction: Assume (a) and (b) are true for i and show for $i+1$. Partition the interval $(t, t + (i+1) \cdot D]$ into $(t, t + D]$ and $(t + D, t + (i+1) \cdot D]$. Since the latter interval is of length $i \cdot D$, the inductive hypothesis applies (replacing t with $t + D$) and we get:
(a) at most $((1+\alpha)^i - 1) \cdot N(t + D)$ nodes enter during $(t + D, t + (i+1) \cdot D]$; and
(b) $N(t + (i+1) \cdot D) \leq (1+\alpha)^i \cdot N(t + D)$.

By the churn assumption, (i) at most $\alpha \cdot N(t)$ nodes enter during $(t, t + D]$ and thus (ii) $N(t + D) \leq (1+\alpha) \cdot N(t)$. To show (a) for $i+1$, combine (i) with the inductive hypothesis for part (a) to see that the number of nodes that enter during $(t, t + (i+1) \cdot D]$ is

$$
\begin{aligned}
&\leq \alpha \cdot N(t) + ((1+\alpha)^i - 1) \cdot N(t + D) \\
&\leq \alpha \cdot N(t) + ((1+\alpha)^i - 1) \cdot (1+\alpha) \cdot N(t) \qquad \text{by (ii)} \\
&= \alpha \cdot N(t) + (1+\alpha)^{i+1} \cdot N(t) - (1+\alpha) \cdot N(t) \\
&= ((1+\alpha)^{i+1} - 1) \cdot N(t).
\end{aligned}
$$

To show (b) for $i+1$:

$$
\begin{aligned}
N(t + (i+1) \cdot D) &\leq (1+\alpha)^i \cdot N(t + D) \qquad \text{by the inductive hypothesis for (b)} \\
&\leq (1+\alpha)^i \cdot (1+\alpha) \cdot N(t) \qquad \text{by (ii)} \\
&= (1+\alpha)^{i+1} \cdot N(t). \quad \square
\end{aligned}
$$

Calculating the maximum number of nodes that can leave in an interval of length $i \cdot D$ as a function of the number of nodes at the beginning of the interval (i.e., the analog of part (a) of Lemma 1) is somewhat complicated by the possibility of nodes entering during the interval, allowing additional nodes to leave.

Lemma 2. *For all α, $0 < \alpha < .206$, all non-negative integers $i \leq 3$, and every time $t \geq 0$, at most $(1 - (1-\alpha)^i) \cdot N(t)$ nodes leave during $(t, t + i \cdot D]$.*

Proof. The proof is by induction on i.

Basis: $i = 0$. For all t, the interval $(t, t + 0 \cdot D] = (t, t]$ is empty and so no nodes leave during it.

Induction: Suppose the lemma is true for i and prove it for $i+1$. Partition the interval $(t, t + (i+1) \cdot D]$ into $(t, t + D]$ and $(t + D, t + (i+1) \cdot D]$. Since the latter interval is of length $i \cdot D$, the inductive hypothesis applies (replacing t with $t + D$), stating that the number of nodes that leave in the latter interval is at most $(1 - (1-\alpha)^i) \cdot N(t + D)$.

Let e be the *exact* number of nodes that enter in $(t, t + D]$ and ℓ be the *exact* number of nodes that leave in $(t, t + D]$. The number of nodes that leave in the entire interval is:

$$
\begin{aligned}
&\leq \ell + (1 - (1-\alpha)^i) \cdot N(t + D) \qquad \text{by the inductive hypothesis} \\
&\leq \ell + (1 - (1-\alpha)^i) \cdot [(1+\alpha) \cdot N(t) - 2\ell] \ .
\end{aligned}
$$

The last line is true since $N(t + D) = N(t) + e - \ell$ which equals $N(t) + (\ell + e) - 2\ell$, which is at most $N(t) + \alpha \cdot N(t) - 2\ell$ by the churn assumption. Algebraic manipulations show that this is

$$
\begin{aligned}
&\leq (1 - (1-\alpha)^i) \cdot (1+\alpha) \cdot N(t) + (2(1-\alpha)^i - 1)\ell \\
&\leq (1 - (1-\alpha)^i) \cdot (1+\alpha) \cdot N(t) + (2(1-\alpha)^i - 1) \cdot \alpha \cdot N(t) \ .
\end{aligned}
$$

The last line is true since $\ell \leq \alpha \cdot N(t)$ by the churn assumption and $(2(1-\alpha)^i - 1)$ is non-negative by the constraints on α and i in the premise of the lemma. This expression equals $(1 - (1-\alpha)^{i+1}) \cdot N(t)$. $\square$

Recall that a node is *active* at time t if it has entered, but not left or crashed, by time t. The next lemma counts how many of the nodes that are active at a given time are still active after $3D$ time has elapsed. It introduces the quantity Z which is the fraction of nodes that must survive an interval of length $3D$.

Lemma 3. *For any interval $[t_1, t_2]$ with $t_2 - t_1 \leq 3D$, where S is the set of nodes present at t_1, at least $Z \cdot |S|$ of the nodes in S are active at t_2. (Recall that $Z = (1-\alpha)^3 - \Delta \cdot (1+\alpha)^3$.)*

Proof. Consider any interval $[t_1, t_2]$ with $t_2 - t_1 \leq 3D$ and let S be the set of nodes present at t_1.

By Lemma 2, at most $(1 - (1-\alpha)^3) \cdot |S|$ nodes leave during the interval. In the worst case, all of the leavers are among the original set of nodes S.

By Lemma 1, part (b), the number of nodes present at t_2 is at most $(1+\alpha)^3 \cdot |S|$. By the crash assumption, up to a Δ fraction of them crash, and in the worst case all of these are among the original set of nodes S.

Thus the number of nodes in S that remain active at the end of the interval is at least

$$|S| - (1 - (1-\alpha)^3) \cdot |S| - \Delta \cdot (1+\alpha)^3 \cdot |S| = \left[(1-\alpha)^3 - \Delta \cdot (1+\alpha)^3\right] \cdot |S|. \quad \square$$

As an immediate corollary, since $|S|$ must be at least N_{min}, the lower bound on N_{min} given in Constraint (A) shows that at least one node survives. To match its use cases, the corollary is stated with respect to a time that is in the middle of the interval.

Corollary 4. *For every $t > 0$, at least one node is active throughout the interval* $[\max\{0, t - 2D\}, t + D]$.

Throughout the proof, a local variable name is subscripted with p and superscripted with t to denote its value in node p at time t; e.g., v_p^t is the value of node p's local variable v at time t.

In the analysis, we will frequently be comparing the data in nodes' *Changes* sets to the set of ENTER, JOINED, and LEAVE events that have actually occurred in a certain interval. We refer to these as *membership events*. We are especially interested in these events that trigger a broadcast invoked by a node that is not in the middle of crashing, as these broadcasts are guaranteed to be received by all nodes that are present for the requisite interval. We call these *active membership events*. Because of the assumed initialization of the nodes in S_0, we use the convention that the set of active membership events occurring in the interval $[0, 0]$ is $\{enter(p) | p \in S_0\} \cup \{join(p) | p \in S_0\}$.

The next lemmas describe how a node's *Changes* set relates to prior active membership events. Lemma 6 states that a node that has been present in the system sufficiently long (at least $2D$ time), has all the information about active membership events up until D time in the past. Lemma 8 states that a joined node, no matter how recently it entered the system, has all the information about active membership events up until $2D$ time in the past. The later parts of the correctness proof only use Lemma 8, but its proof relies on Lemma 6. The proof of Lemma 8 relies on Lemma 7, which is rather technical and states that under certain circumstances a node receives an **enter-echo** message from a long-lived node; we have extracted it as a separate lemma as it is also used later in the proof of Lemma 13. Throughout the proof we denote by t_p^e the time when event ENTER_p occurs.

The proofs of Lemmas 6 and 8 use the next observation, which follows from the fact that nodes broadcast **enter/join/leave** messages when they enter/join/leave and these messages take at most D time to arrive at active nodes (unless the broadcast is the very last step by a crashing node, in which case the message might not be received by some nodes).

Observation 5. For every node p and all times $t \geq t_p^e + D$ such that p is active at time t, $Changes_p^t$ contains all the active membership events for $[t_p^e, t - D]$.

Lemma 6. *For every node p and all times $t \geq t_p^e + 2D$ such that p is active at t, $Changes_p^t$ contains all the active membership events for* $[0, t - D]$.

Proof. The proof is by induction on the order in which nodes enter. In particular, we consider the nodes in increasing order of ENTER events, breaking ties arbitrarily, and show the properties are true for the current node at all relevant times.

Basis: The first nodes to enter are those in S_0 and they are assumed to do so at time 0. Consider $p \in S_0$. For $t \geq 2D$, Observation 5 gives the result.

Induction: Let p be the next node (not in S_0) to enter, at time t_p^e, and assume the lemma is true for all nodes that entered previously.

Consider any time $t \geq t_p^e + 2D$ such that p is active at t. By Corollary 4, there exists a node q that is active throughout $[t_p^e - 2D, t_p^e + D]$. Let t' be the time when q receives p's **enter** message and t'' be the time when p receives q's **enter-echo** response. We will show that $Changes_p^t$ contains all the active membership events for $[0, t - D]$ in three steps: one for $[0, t' - D]$, one for $[t' - D, t_p^e]$, and one for $[t_p^e, t - D]$.

1. Note that q enters the system at least $2D$ time before it sends its **enter-echo** message to p at time t'. By the inductive hypothesis, when q sends that message, its *Changes* set contains all the active membership events for $[0, t' - D]$. Once p receives the message, at time t'' which is less than or equal to t, its *Changes* set also contains all the active membership events for $[0, t' - D]$.
2. Suppose some node r enters, joins, or leaves in $[t' - D, t_p^e]$ and r does not crash during that event. Node r's **enter/join/leave** message is received by q either before t_p^e, in which case the information is included in q's **enter-echo** message to p, or after t_p^e, in which case q sends an **enter/join/leave-echo** message for r, which is received by p before t. In either case, the information about r's event propagates to p before t. Thus the result holds for $[t' - D, t_p^e]$.
3. Observation 5 gives the result for $[t_p^e, t - D]$. □

Lemma 7. *Suppose node p is joined and active at some time t and the first **enter-echo** response that p receives from a joined node q is sent at time $t' \le t$. If $Changes_q^{t'}$ contains all the active membership events for $[0, \max\{0, t' - 2D\}]$, then before p joins, it receives an **enter-echo** response from some node q' that is active throughout the interval $[\max\{0, t' - 2D\}, t' + D]$.*

Proof. Let S be the set of nodes present at time $\max\{0, t' - 2D\}$. We will show that at least one of the **enter-echo** responses received by p before joining is from a node in S, which is our desired q'. We start with the value of *join_threshold*, which is the number of **enter-echo** responses for which p waits before joining, and then subtract (1) the maximum number of **enter-echo** responses that could come from nodes not in S, (2) the maximum number of nodes in S that could leave too soon (before $t' + D$), and (3) the maximum number of nodes in S that could crash too soon (before $t' + D$).

The value of *join_threshold* is based on the size of p's *Present* set at time t'', immediately after p receives the **enter-echo** response from q (cf. Line 9 of Algorithm 1). By the premise of the lemma, $Changes_q^{t'}$ contains all the active membership events for $[0, \max\{0, t' - 2D\}]$. Thus when p receives the **enter-echo** response from q at time $t'' \le t' + D$, its *Present* variable contains, at a minimum, all the nodes in S minus those that left during $[\max\{0, t' - 2D\}, t'']$—call this quantity ℓ—and minus those that crashed while broadcasting their **enter** message so that p did not receive it—call this quantity f. Thus *join_threshold* $\ge \gamma \cdot (|S| - \ell - f)$.

We now count the maximum number of **enter-echo** responses that p can receive from nodes not in S before joining. These would be from nodes that enter after $\max\{0, t' - 2D\}$ but no later than $t' + D$, as nodes entering after $t' + D$ do not receive p's **enter** message. The number of such nodes is $((1+\alpha)^3 - 1) \cdot |S|$ by Lemma 1 part (a).

We then subtract the maximum number of nodes in S that leave before $t' + D$. By Lemma 2, at most $(1 - (1-\alpha)^3) \cdot |S|$ nodes leave during $(\max\{0, t' - 2D\}, t' + D]$. In the worst case, all these leavers are in S. Recall that we have already charged ℓ to this budget.

Finally we subtract the maximum number of nodes in S that crash before $t' + D$. The system size at $t' + D$ is at most $(1+\alpha)^3 \cdot |S|$ by Lemma 1 part (b). At most $\Delta \cdot (1+\alpha)^3 \cdot |S|$ nodes are crashed at time $t' + D$ by the crash assumption. In the worst case, all these crashed nodes are in S. Recall that we have already charged f crashes to this budget.

What remains is at least

$$\gamma \cdot (|S| - \ell - f) - ((1+\alpha)^3 - 1) \cdot |S| \\ - [(1 - (1-\alpha)^3) \cdot |S| - \ell] - [\Delta \cdot (1+\alpha)^3 \cdot |S| - f] ,$$

which after doing some algebra and using fact that $(1 - \gamma)(\ell + f) \ge 0$ is equal to

$$|S| \cdot (\gamma - (1+\alpha)^3 + (1-\alpha)^3 - \Delta(1+\alpha)^3) .$$

Since $|S|$ must be at least N_{min}, Constraint (A) ensures that the expression is at least one. Thus before p joins, it receives an **enter-echo** response from at least one node q' that is active throughout $(\max\{0, t' - 2D\}, t' + D]$. □

Lemma 8. *For every node p and all times t such that p is joined and active at t, $Changes_p^t$ contains all the active membership events for $[0, \max\{0, t - 2D\}]$.*

Proof. The proof is by induction on the order in which nodes join. In particular, we consider the nodes in increasing order of JOIN events, breaking ties arbitrarily, and show the properties are true for the current node at all relevant times.

Basis: The first nodes to join are those in S_0 and they are assumed to do so at time 0. Consider $p \in S_0$. When $t \le 2D$, we just need to show that $Changes_p^t$ contains all the active membership events for $[0, 0]$, which is true by the assumed initialization of nodes in S_0. When $t > 2D$, Observation 5 implies the result.

Induction: Let p be the next node (not in S_0) to join and assume the lemma is true for all nodes that previously joined. Consider any time t when p is joined and active.

When $t - t_p^e \ge 2D$, Lemma 6 gives the result. So we suppose $t - t_p^e < 2D$. If $t \le 2D$, then all that's required is for $Changes_p^t$ to include all the active membership events in $[0, 0]$. Since p joined, it received an **enter-echo** message from some previously joined node, which by the inductive hypothesis had all the active membership events for $[0, 0]$ in its *Changes* set when it sent the **enter-echo**. Thus p receives all the active membership events for $[0, 0]$ before it joins. For rest of the proof, assume $t > 2D$.

We will show that $Changes_p^t$ contains all the active membership events for $[0, t-2D]$ in two steps: one for $[0, \max\{0, t'-2D\}]$ and one for $[\max\{0, t'-2D\}, t-2D]$ for an appropriately chosen $t' < t$.

1. Let q be the first joined node from which p gets an **enter-echo** response to its **enter** message. Let t' be the time when q sends the **enter-echo** message. By the inductive hypothesis, since q is joined at t', $Changes_q^{t'}$ contains all the active membership events for $[0, \max\{0, t'-2D\}]$, and thus so does $Changes_p^t$.
2. By Lemma 7, p receives an **enter-echo** message at some time before it joins from a node q' that is active throughout the interval $[\max\{0, t'-2D\}, t'+D]$. Let u' be the time when q' sends its **enter-echo** response to p. Suppose some node r enters, joins or leaves in $[\max\{0, t'-2D\}, t-2D]$ and does not crash during the event. Our goal is to show that p receives the information about r by time t. The latest that r's message is sent is $t-2D$. Since $t-t' \leq t-t_p^e < 2D$, it follows that $t-D \leq t'+D$ and thus q' is guaranteed to receive r's message, as q' is still active at $t-D$, the latest that the message could arrive. If q' receives r's message before u', then p gets the information about r by time t via the **enter-echo** response from q'. Otherwise, q' receives r's message after u'; the latest this can be is $t-D$. Then q' sends an **enter-echo** message for r which is received by p by time t. □

We can prove that a node that is active sufficiently long eventually joins.

Theorem 9. *Every node p that enters at some time t and is active for at least $2D$ time joins by time $t+2D$.*

Proof. The proof is by induction on the order in which nodes enter the system.

Basis: The first nodes to enter are those in S_0 and they are assumed to do so at time 0. Since they also are assumed to join at time 0, the theorem follows.

Induction: Let p be the next node (not in S_0) to enter, at time t_p^e, and assume the lemma is true for all nodes that entered previously. Suppose p is active at $t_p^e + 2D$.

First we show that p receives an **enter-echo** response to its **enter** message from at least one joined node.

Suppose $t_p^e < 2D$. By Corollary 4, at least one node in S_0 is active throughout $[0, 3D]$ and thus responds to p's **enter** message.

Suppose $t_p^e \geq 2D$. By Corollary 4, there is a node q that is active throughout $[t_p^e - 2D, t_p^e + D]$. Then q enters at least $2D$ time before t_p^e and by the inductive hypothesis is joined by t_p^e. Since q is active at least until $t_p^e + D$, it receives p's **enter** message by time $t_p^e + D$ and sends back an **enter-echo** which is received by p by time $t_p^e + 2D$.

We now calculate an upper bound on *join_threshold*, the number of **enter-echo** responses for which p waits before joining. This value is based on the size of p's *Present* set when it first receives an **enter-echo** response from a joined node (cf. Line 9 of Algorithm 1). Let q' be the sender of this message, let t' be the time when the message is sent and t'' the time when it is received. Since $t' \geq t_p^e \geq 2D$, it follows that $t' - 2D \geq 0$. By Lemma 8, $Changes_{q'}^{t'}$ contains all the active membership events for $[0, t'-2D]$ and thus so does $Changes_p^{t''}$. As a result, $Present_p^{t''}$ contains, at most, all the nodes that are present at time $t'-2D$ (call this set S) plus the maximum set of nodes that could have entered since then. Since $t'' \leq t' + D$, it follows from Lemma 1 part (a) that at most $((1+\alpha)^3 - 1) \cdot |S|$ nodes enter during $(t'-2D, t'']$. Thus $join\_threshold \leq \gamma \cdot (1+\alpha)^3 \cdot |S|$.

We now show that p is guaranteed to receive at least *join_threshold* **enter-echo** responses from nodes in S by time $t_p^e + 2D$. Each node in S that does not leave or crash by $t_p^e + D$ receives p's **enter** message and sends an **enter-echo** response by time $t_p^e + D$, which is received by p by time $t_p^e + 2D$. The minimum number of such nodes is, by Lemma 3 and considering the interval $(t'-2D, t'+D]$:

$$\begin{aligned} Z \cdot |S| &\geq \gamma \cdot (1+\alpha)^3 \cdot |S| \qquad \text{by Constraint (B)} \\ &\geq join\_threshold. \quad \square \end{aligned}$$

We prove that a phase terminates if the invoking client node is active long enough.

Theorem 10. *A phase invoked by a client that remains active completes within $2D$ time.*

Proof. Consider a phase invoked by node p at time t. We show that the number of nodes that respond to p's **collect-query** or **store** message is at least as large as the value of *threshold* computed by p in Line 31 or 38 or 44 of Algorithm 2.

Let S be the set of nodes present at time $\max\{0, t-2D\} = t'$. By Lemma 3, the number of those nodes that are still active at time $t+D$ is at least $Z \cdot |S|$. If $t' = 0$, then $S = S_0$ and all these nodes are joined throughout; otherwise, by Theorem 9 all these nodes are joined by time t.

We now show that $|S| \geq |Present_p^t|/(1+\alpha)^2$. By Lemma 8, $Changes_p^t$ contains all the active membership events for $[0, t']$. $Present_p^t$ is as large as possible if every node in S succeeds in the broadcast of its **enter** message, none of the nodes in S leave during $[t', t]$, and the maximum number of nodes enter during that interval and their **enter** messages get to p by time t. Lemma 1 part (a) implies that the maximum number of nodes that can enter is $(1+\alpha)^2 \cdot |S|$. Thus $|Present_p^t| \leq (1+\alpha)^2 \cdot |S|$.

Thus the number of nodes that are joined by time t and are still active at time $t + D$, guaranteed to respond to p, is at least

$$\begin{aligned} Z \cdot |S| &\geq Z \cdot |Present_p^t|/(1+\alpha)^2 \\ &\geq \beta \cdot |Present_p^t| \qquad \text{by Constraint (C)} \\ &\geq \beta \cdot |Members_p^t| \,. \end{aligned}$$

The last inequality holds since $enter(q)$ is added to $Changes_p$ together with $join(q)$. Since $threshold$ is set to $\beta \cdot |Members_p^t|$ at time t, p receives the required number of **collect-reply** or **store-ack** messages by time $t + 2D$ and the phase completes. □

The following observation is true since in this case node p receives phase s's **store** message directly within D time.

Observation 11. For any store phase s that starts at time t_s and calls broadcast (Line 46 of Algorithm 2) without crashing during the broadcast, and any node p that is active throughout $[t_s, t]$ where $t \geq t_s + D$, $view(s) \preceq LView_p^t$.

The next lemma is the analog of Lemma 6: a node that has been active for at least $2D$ time "knows about" store phases that started up to D in the past.

Lemma 12. *If node p is active at any time $t \geq t_p^e + 2D$, then $view(s) \preceq LView_p^t$ for every store phase s that starts at or before $t - D$ and calls broadcast (Line 46 of Algorithm 2) without crashing during the broadcast.*

Proof. The proof is by induction on the order in which nodes enter the system.

Basis: The first nodes to enter are those in S_0 and they do so at time 0. The claim holds by Observation 11.

Induction: Let p be the next node (not in S_0) to enter and assume the claim is true for all nodes that entered previously. Consider any time $t \geq t_p^e + 2D$ when p is active. Let s be a store phase that starts at $t_s \leq t - D$ and calls broadcast without crashing. If $t_s \geq t_p^e$, the claim holds by Observation 11.

Suppose $t_s < t_p^e$. By Corollary 4, there is at least one node q that is active throughout $[\max\{0, t_p^e - 2D\}, t_p^e + D]$. Since $t \geq t_p^e + 2D$, p receives q's **enter-echo** response by time t. Since views and sequence numbers are included in **enter-echo** messages, $LView_q^{t'} \preceq LView_p^t$, where t' is the time when q receives p's **enter** message.

Case 1: $t_s < \max\{0, t_p^e - D\}$. We show that the inductive hypothesis applies for node q, time t', and store phase s. Thus $view(s) \preceq LView_q^{t'}$, and by transitivity, $view(s) \preceq LView_p^t$. To show that the inductive hypothesis holds, note that q enters before p, q has been active for at least $2D$ time by t' and store phase s starts at or before $t' - D$.

Case 2: $t_s \geq \max\{0, t_p^e - D\}$. The **store** message sent during s is guaranteed to arrive at q either before t_p^e or at or after t_p^e. In the former case, q's **enter-echo** response, which p receives by $t_p^e + 2D \leq t$, contains a view V such that $view(s) \preceq V$. In the latter case, q's **store-echo** message contains a view V with $view(s) \preceq V$ and p receives this message by $t_s + 2D < t_p^e + 2D \leq t$. In both situations, $view(s) \preceq LView_p^t$. □

The next lemma is the analog of Lemma 8: a node that is joined "knows about" store phases that started up to $2D$ in the past.

Lemma 13. *If node p is joined and active at any time t, then $view(s) \preceq LView_p^t$ for every store phase s that starts at or before $t - 2D$ and calls broadcast (Line 46 of Algorithm 2) without crashing.*

Proof. The proof is by induction on the order in which nodes join the system.

Basis: The first nodes to join are those in S_0 and they do so at time 0, which is also the time that they enter. The claim holds by Observation 11.

Induction: Let p be the next node (not in S_0) to join and assume the claim is true for all nodes that joined previously. Consider any time t at which p is joined and active. Let s be any store phase that starts at $t_s \leq t - 2D$ and calls broadcast without crashing. If $t \geq t_p^e + 2D$, then the claim follows from Lemma 12.

Suppose $t < t_p^e + 2D$. For every store phase that starts at or after t_p^e, the claim follows from Observation 11.

Consider any store phase that starts at some time $t_s < t_p^e$. Let q be the sender of the first **enter-echo** response received by p from a joined node; suppose the message is sent at t' and received at t''.

Case 1: $t_s < t' - 2D$. We show that the inductive hypothesis holds for node q, time t', and store phase s. Thus $view(s) \preceq LView_q^{t'}$, and by transitivity, $view(s) \preceq LView_p^t$. To show that the inductive hypothesis holds, note that q joins before p, it is joined at time t', and store phase s starts before $t' - 2D$.

Case 2: $t_s \geq t' - 2D$. Since q is joined at t', Lemma 8 implies that $Changes_q^{t'}$ contains all the active membership events for $[0, \max\{0, t' - 2D\}]$. Thus Lemma 7 applies and before p joins it receives an **enter-echo** response from a node q' that is active throughout $[\max\{0, t' - 2D\}, t' + D]$. The **store** message sent during s is guaranteed to arrive at q' either before t_p^e

or at or after t_p^e. In the former case, the **enter-echo** message from q' that is sent to p contains a view V with $view(s) \preceq V$; this message is received by p before it joins. In the latter case, the **store-echo** message from q' that is sent to p contains a view V with $view(s) \preceq V$; this message is received by p by $t_s + 2D < t_p^e + 2D \leq t$. In both situations, $view(s) \preceq LView_p^t$. □

The next lemma gives a lower bound on the size of a node's *Members* set as a function of the size of the system $3D$ time in the past.

Lemma 14. *For every node p and every time t at which p is joined and active,* $|Members_p^t| \geq ((1-\alpha)^3 - \Delta \cdot (1+\alpha)^2) \cdot N(\max\{0, t-3D\})$.

Proof. Let S be the set of nodes that are present at time $\max\{0, t-3D\}$, so $|S| = N(\max\{0, t-3D\})$.

First, assume $t \geq 3D$. Since Theorem 9 implies that it takes at most $2D$ time for a node to join, at a minimum $Members_p^t$ contains all the nodes in S except for those that leave during $[t-3D, t]$ and those that crash while broadcasting their **join** message so that p does not receive the message. By Lemma 2, the maximum number of nodes that leave during $[t-3D, t]$ is $(1-(1-\alpha)^3) \cdot |S|$. To maximize the number of nodes that crash while sending their **join** message, we consider the largest that the system can be by time $t-D$ (the latest that the nodes in S can join), which is $(1+\alpha)^2 \cdot |S|$ by Lemma 1. We assume that the maximum number of nodes crash, which is $\Delta \cdot (1+\alpha)^2 \cdot |S|$, and that all the crashed nodes are in S. Thus, $|Members_p^t| \geq ((1-\alpha)^3 - \Delta \cdot (1+\alpha)^2) \cdot |S|$.

Now, assume $t < 3D$. Then S equals S_0, the set of nodes initially in the system, and $Members_p^t$ is minimized if no nodes enter and the maximum number of nodes in S_0 leave and p receives all their **leave** messages by time t. Since Lemma 2 implies that the maximum number of nodes that can leave during $[0, t]$ is $(1-(1-\alpha)^3) \cdot |S|$, it follows that $|Members_p^t| \geq (1-\alpha)^3 \cdot |S|$, which is bigger than the desired lower bound. □

To prove the following lemmas, we consider two cases: If the two phases are sufficiently far apart in time, then an information-propagation argument, analogous to that used for the *Changes* sets, applies. If the two phases are close together in time, then an argument relating to overlapping sets of contacted nodes is used.

Lemma 15. *For any store phase s and any collect phase c, if s finishes before c starts and c terminates, then $view(s) \preceq view(c)$.*

Proof. Let p_1 be the client node that executes s and t_s the start time of s. Let p_2 be the client node that invokes c and t_c the start time of c. Let Q_s be the set of nodes that p_1 hears from during s (i.e., that sent messages causing p_1 to increment *counter* on Line 48 of Algorithm 2) and Q_c be the set of nodes that p_2 hears from during c (i.e., that sent messages causing p_2 to increment *counter* on Line 36 of Algorithm 2).

Case I: $t_c - t_s \geq 2D$. Consider any node $q \in Q_c$. Since q is in Q_c, q is joined when it receives c's **collect-query** message at some time, say $t \geq t_c$. By the assumption of the case, $t - t_s \geq 2D$. Thus by Lemma 13, $view(s) \preceq LView_q^t$. Since p_2 receives an **enter-echo** message from q containing $LView_q^t$ before completing c, it follows that $view(s) \preceq view(c)$.

Case II: $t_c - t_s < 2D$. We will show that Q_c and Q_s have a nonempty intersection and thus Q_c contains a node whose *LView* variable is $\preceq view(s)$ before it sends its **collect-reply** message to p_2, ensuring that $view(s) \preceq view(c)$. We define the following sets of nodes.

- Let J be the set of all nodes that are joined and active at some time in $[t_c, t_c + D]$. These are the nodes that could possibly respond to c's **collect-query** message. Thus $Q_c \subseteq J$.
- Let $K \subseteq Q_s$ be the set of nodes in Q_s that are still active at t_c. Note that $K \subseteq J$.

We will show that $|Q_c| + |K| > |J|$. Since Q_c and K are both subsets of J, it follows that they intersect, and thus Q_c and Q_s intersect. We show the inequality by calculating an upper bound on $|J|$ and lower bounds on $|Q_c|$ and $|K|$. All three bounds are stated in terms of a common quantity, which is the system size at a particular time $t^* = \max\{0, t_c - 2D\}$.

First we calculate an upper bound on $|J|$. Since it takes at most $2D$ time to join after entering by Theorem 9, every node in J is either present at t^* or enters during $[t^*, t_c + D]$. By Lemma 1(b), $|J| \leq (1+\alpha)^3 \cdot N(t^*)$.

Next we calculate a lower bound on Q_c.

$$
\begin{aligned}
|Q_c| &= \beta \cdot |Members_{p_2}^{t_c}| && \text{by the code} \\
&\geq \beta \cdot [(1-\alpha)^3 - \Delta \cdot (1+\alpha)^2] \cdot N(\max\{0, t_c - 3D\}) && \text{by Lemma 14} \\
&\geq \beta \cdot [(1-\alpha)^3 - \Delta \cdot (1+\alpha)^2] \cdot (1+\alpha)^{-1} \cdot N(t^*) && \text{by Lemma 1 (b).}
\end{aligned}
$$

We now calculate a lower bound on $|K|$. By Lemma 3, at most $(1-Z) \cdot N(t_s)$ nodes crash or fail during $[t_s, t_c + D]$, since the length of the interval is at most $3D$. In the worst case, all the nodes that crash or fail are in Q_s.

$$
\begin{aligned}
|K| &\geq |Q_s| - (1-Z)\cdot N(t_s) \\
&= \beta \cdot |Members^{t_s}_{p_1}| - (1-Z)\cdot N(t_s) \qquad \text{by the code} \\
&\geq \beta \cdot [(1-\alpha)^3 - \Delta\cdot(1+\alpha)^2]\cdot N(\max\{0, t_s - 3D\}) - (1-Z)\cdot N(t_s) \\
&\quad \text{by Lemma 14} \\
&\geq \beta \cdot [(1-\alpha)^3 - \Delta\cdot(1+\alpha)^2]\cdot (1+\alpha)^{-3}\cdot N(t^*) - (1-Z)\cdot N(t_s) \\
&\quad \text{by Lemma 1(b) since } 0 < t_c - t_s < 2D \\
&\geq \beta \cdot [(1-\alpha)^3 - \Delta\cdot(1+\alpha)^2]\cdot (1+\alpha)^{-3}\cdot N(t^*) - (1-Z)\cdot (1+\alpha)^2\cdot N(t^*) \\
&\quad \text{by Lemma 1(b) since } 0 < t_s - t^* < 2D \text{ and } 1-Z > 0 \\
&= \left[\beta \cdot [(1-\alpha)^3 - \Delta\cdot(1+\alpha)^2]\cdot (1+\alpha)^{-3} - (1-Z)\cdot(1+\alpha)^2\right]\cdot N(t^*)\,.
\end{aligned}
$$

Finally, we show $|Q_c| + |K| > |J|$.

$$
\begin{aligned}
|Q_c| + |K| &\geq [\beta\cdot[(1-\alpha)^3 - \Delta\cdot(1+\alpha)^2]\cdot(1+\alpha)^{-1} + \beta\cdot[(1-\alpha)^3 \\
&\qquad - \Delta\cdot(1+\alpha)^2]\cdot(1+\alpha)^{-3} - (1-Z)\cdot(1+\alpha)^2]\cdot N(t^*) \\
&> (1+\alpha)^3\cdot N(t^*) \qquad \text{by Constraint (D)} \\
&\geq |J|\,. \quad \square
\end{aligned}
$$

Theorem 16. *The schedule resulting from the restriction of the execution to the* store *and* collect *invocations and responses satisfies regularity for the store-collect problem.*

Proof. (1) Suppose *cop* is a *collect* operation that returns view V. Let c be the collect phase of *cop*. Let p be a node. If $V(p) = \perp$ and a *store* operation by p, consisting of store phase s, precedes *cop*, then, by Lemma 15, $view(s) \preceq view(c)$. Hence, $view(s)$ contains a tuple for p with a non-$\perp$ value, which is a contradiction.

Therefore, $V(p) = v \neq \perp$. We show that a $\textsc{Store}_p(v)$ invocation occurs before *cop* completes and no other *store* operation by p occurs between this invocation and the invocation of *cop*. A simple induction shows that every (non-$\perp$) value for one node in another node's *LView* variable at some time comes from a STORE invocation by the first node that has already occurred. Since V is the value of the invoking node's *LView* variable when *cop* completes, there is a previous $\textsc{Store}_p(v)$ invocation.

Now suppose for the sake of contradiction that the $\textsc{Store}_p(v)$ completes—call this operation *sop*—and there is another *store* operation by p, call it *sop*′, that follows *sop* and precedes *cop*. Let v' be the value of *sop*′; by the assumption of unique values, $v \neq v'$. Since *sop* and *sop*′ are executed by the same node, it is easy to see from the code that $view(sop) \preceq view(sop')$. By Lemma 15, $view(sop') \preceq view(c) = V$. But then value v is superseded by value $v' \neq v$, contradicting the assumption that $V(p) = v$.

(2) Suppose cop_1 and cop_2 are two *collect* operations such that cop_1 returns V_1, cop_2 returns V_2, and cop_1 precedes cop_2. Note that cop_1 contains a store phase s which finishes before the collect phase c of cop_2 begins. By Lemma 15, $view(s) \preceq view(c)$. Regularity holds since $view(s) = V_1$ and $view(c) = V_2$, implying that $V_1 \preceq V_2$. $\square$

By Theorem 9, every node that enters and remains active sufficiently long eventually joins. Since a *store* operation consists of a store phase and a *collect* operation consists of a collect phase followed by a store phase, by Theorem 10, which states that every phase eventually completes as long as the invoker remains active, every operation eventually completes as long as the invoker remains active. Finally, Theorem 16 ensures regularity.

Corollary 17. CCC *is a correct implementation of a store-collect object, in which each* Store *or* Collect *completes within a constant number of communication rounds.*

6. Implementing distributed objects despite continuous churn

In this section we show how to implement a variety of objects using store-collect. For all applications, we assume that the conditions for store-collect termination hold, which guarantees termination of the operations.

6.1. Simple, non-linearizable objects

We start with three simple applications of store-collect for implementing other (non-linearizable) shared objects.[3] The choice of problems and algorithms follows [24], but the algorithms inherit good efficiency properties from our store-collect implementation.

Max register Holds the largest value written into it [5]; provides two operations:

- WRITEMAX(v) takes a value v as an argument and returns ACK.
- READMAX() has no arguments and returns a value.

A max register ensures that the value returned by a READMAX is equal to the value of some WRITEMAX that starts before the READMAX completes and is at least as large as the value of any WRITEMAX that ends before the READMAX starts. If there is no preceding WRITEMAX, then the READMAX returns $\perp$ or the value of an overlapping WRITEMAX.

Algorithm 4 uses a single store-collect object, holding a single value *val* for each node, a local variable V for each node, holding a store-collect view, and a local variable *lmax* for each node, initialized to $-\infty$. WRITEMAX stores the new value (Line 56) if it is the largest value to be stored so far by this node, and returns ACK (Line 58). READMAX collects a view (Line 59) and returns the maximum value stored in it (Line 62).

Algorithm 4 Max-Register: code for node p.

```
When WRITEMAX_p(v) occurs:
55: if lmax < v then
56:     STORE_p(v)
57:     lmax := v
58: return ACK

When READMAX_p() occurs:
59: V := COLLECT_p()
60: if V = ∅ then
61:     return ⊥
62: else return max(V.val)
```

To see that this implements a max register, first note that the WRITEMAX code ensures that each node stores a sequence of increasing values. Consider a READMAX that returns v. If $v = \perp$, then the READMAX's collect returned nothing, which means by the regularity property of store-collect that there is no store that precedes the collect and thus no WRITEMAX that precedes the READMAX. If $v \neq \perp$, then v is the argument of a store that starts before the collect finishes, and thus v is the argument of a WRITEMAX that starts before the READMAX finishes. Assume in contradiction there exists $v' > v$ such that a WRITEMAX(v') ends before the READMAX begins. By the regularity property of store-collect, the READMAX would observe v' or a larger value in its collect, and thus would return v' or a larger value, not v.

Abort flag A Boolean flag that can only be raised from *false* to *true* [24]; provides two operations:

- ABORT() has no arguments and returns ACK.
- CHECK() has no arguments and returns *true* or *false*.

An abort flag ensures that if CHECK returns *true* then an ABORT starts before CHECK ends; furthermore, if CHECK returns *false* then no ABORT completes before CHECK starts.

Algorithm 5 follows [24]. It uses a single store-collect object, holding a single *flag* for each node, and a local variable F for each node, holding a view. ABORT stores true (Line 59) and returns ACK (Line 60). CHECK collects the *flags* (Line 61). If any of the flags is true then CHECK returns true (Line 62). Otherwise, returns false (Line 63).

Algorithm 5 Abort Flag: code for node p.

```
When ABORT_p() occurs:
59: STORE_p(true)
60: return ACK

When CHECK_p() occurs:
61: F := COLLECT_p()
62: if ∃q s.t. F(q) = true then
63:     return true
64: else return false
```

To see that this implements an abort flag, assume an ABORT completes before a CHECK starts. Then, the store of this ABORT sets the flag before the start of the collect by CHECK. By the regularity of store-collect, the CHECK returns true. Conversely, if no ABORT starts before a CHECK completes, then the regularity of store-collect implies that the collect of CHECK does not read any *false* value, and returns false.

[3] The behavior of these objects can be formalized through *interval linearizability* [14].

Set Contains all values added into it [24]; provides two operations:

- ADDSET(v) takes a value v as an argument and returns ACK.
- READSET() has no arguments and returns a set of values.

This object ensures that if an ADDSET(v) completes before a READSET starts, then the READSET returns a set of values that includes v; furthermore, if a READSET's return value includes v, then an ADDSET(v) starts before the READSET completes.

Algorithm 6 uses a store-collect object, holding a set of values for each node, and two local variables for each node: S, a view, and *LSet*, holding all values previously stored by p. ADDSET adds the value to the local set (Line 65), stores it (Line 66), and returns ACK (Line 67). READSET collects the set of values (Line 68) and returns the union of all the sets of values (Line 69).

Algorithm 6 Set: code for node p.

When ADDSET$_p(v)$ **occurs:**
65: $LSet := LSet \cup \{v\}$
66: STORE$_p$(*LSet*)
67: **return** ACK

When READSET$_p()$ **occurs:**
68: $S :=$ COLLECT$_p()$
69: **return** $\cup S.set$

To see that this implements a set, assume an ADDSET(v) completes before a READSET starts. Since the store of ADDSET completes before the collect of READSET starts, the regularity property for store-collect implies that the collect returns a set of values that includes v. Conversely, if a READSET returns a set that includes the value v, then v appears in the information returned by the enclosed collect. By the regularity property of store-collect, there is a store of v that starts before the collect ends and thus its enclosing ADDSET starts before the READSET ends.

6.2. Atomic snapshots

Like other atomic snapshot algorithms [1,16,27], our algorithm uses repeated collects to identify an atomic scan when two collects return the same collected views. Updates help scans to complete by embedding an atomic scan that can be borrowed by overlapping scans they interfere with. A major challenge in our algorithm is to identify an atomic scan to borrow, when there are no two identical views. In the classical approach [1], if a scanner fails to obtain two identical views but observes *two* changes by the same node, then it can safely borrow the embedded scan in that node's second store, which is guaranteed to be sufficiently recent. When the number of nodes is static, the scanner is bound to eventually either obtain two identical views or observe two changes by the same node. In our dynamic model, however, since new nodes can continue to arrive, each new node can do a single store, causing the scanner to fail to get two identical views but also fail to observe two changes by the same node. We thus need a new mechanism for the scanner to identify a view that is recent enough to borrow. Similar to the Spiegelman and Keidar [27] algorithm, we attach sequence numbers to scan operations. Scanning node p can borrow q's embedded scan from the collected view if that scan is enclosed within p's scan, as indicated by q having read the sequence number of p's current scan before q performs its scan.

The set from which the values to be stored in the snapshot object are taken is denoted Val_{AS}. A *snapshot view* is a subset of $\Pi \times Val_{AS}$, i.e., a set of (node id, value) pairs, without duplicate node ids.

Formally, an *atomic snapshot* [1] provides two operations:

- SCAN(), which has no arguments and returns a snapshot view, and
- UPDATE(v), which takes a value $v \in Val_{AS}$ as an argument and returns ACK.

Its sequential specification consists of all sequences of updates and scans in which the snapshot view returned by a SCAN contains the value of the last preceding UPDATE for each node p, if such an UPDATE exists, and no value, otherwise.

An implementation should be *linearizable* [22]. Roughly speaking, for every execution α, we should find a sequence of operations, $lin(\alpha)$, containing all completed operations in α and some of the pending operations, such that:

- $lin(\alpha)$ is in the sequential specification of an atomic snapshot, and
- $lin(\alpha)$ preserves the real-time order of non-overlapping operations in α.

Our algorithm to implement an atomic snapshot uses a store-collect object, whose values are taken from the set ($\mathcal{P}$ indicates the power set of its argument):

$$Val_{SC} = Val_{AS} \times \mathbb{N} \times \mathbb{N} \times \mathcal{P}(\Pi \times Val_{AS}) \times \mathcal{P}(\Pi \times \mathbb{N})$$

The first component (*val*) holds the argument of the most recent update invoked at p. The second component (*usqno*) holds the number of updates performed by p. The third component (*ssqno*) holds the number of scans performed by p. The fourth component (*sview*) holds a snapshot view that is the result of a recent scan done by p; it is used to help other nodes

complete their scans. The fifth component (*scounts*) holds a set of counts of how many scans have been done by the other nodes, as observed by p. The projection of an element v in Val_{SC} onto a component is denoted, respectively, $v.val$, $v.usqno$, $v.ssqno$, $v.sview$, $v.scounts$.

A *store-collect view* is a subset of $\Pi \times Val_{SC}$, i.e., a set of (node id, value) pairs, with no duplicate node ids. We extend the projection notation to a store-collect view V, so that $V.comp$ is the result of replacing each tuple $\langle p, v\rangle$ in V with $\langle p, v.comp\rangle$. Recall that for any kind of view V, $V(p)$ is the second component of the pair whose first component is p ($\perp$ if there is no such pair). Sometimes we restrict attention to those tuples in a view V whose *val* component is a "real" value, reflecting a store; to this end we use the notation

$$r(V) = \{\langle p, v\rangle | \langle p, v\rangle \in V \text{ and } v.val \neq \perp\}.$$

To execute a SCAN, Algorithm 7 increments the scan sequence number (*ssqno*) (Line 70) and stores it in the shared store-collect object with all the other components unchanged, indicated by the $-$ notation. Then, a view is collected (Line 72). In a while loop, the last collected view is saved and a new view is collected (Line 74). If the two most recently collected views reflect the same set of updates (Line 75), the latest collected view is returned (Line 76); Lines 75 and 76 consider only tuples with "real" values. We call this a *successful double collect*, and say that this is a *direct scan*. Otherwise, the algorithm checks whether the last collected view contains a node q that has observed its own *ssqno*, by checking the *scounts* component (Line 77). If this condition holds, the snapshot view of q is returned (Line 78); we call this a *borrowed scan*.

An UPDATE first obtains all scan sequence numbers from a collected view and assigns them to a local variable *scounts* (Line 79). Next, the value of an embedded scan is saved in a local variable *sview* (Line 80). Then it sets its *val* variable to the argument value and increments its update sequence number (Lines 81 and 82). Finally the new value, update sequence number, collected view, and set of scan sequence numbers are stored; the node's own scan sequence number is unchanged (Line 83).

Algorithm 7 Atomic snapshot: code for node p.

Local Variables:
ssqno: int, initially 0 // counts how many scans p has invoked so far
scounts: set of (node id, integer) pairs with no duplicate node ids; initially $\emptyset$
val: an element of Val_{AS}, initially $\perp$ // argument of most recent update invoked by p
usqno: int, initially 0 // number of updates p has invoked so far
sview: a snapshot view, initially $\emptyset$ // the result of recent embedded scan by p
V_1, V_2: store-collect views, both initially $\emptyset$

When SCAN$_p$() occurs:
70: *ssqno*++
71: STORE$_p(\langle -, -, ssqno, -, -\rangle)$
72: $V_1 :=$ COLLECT$_p()$
73: **while** *true* **do**
74: $V_2 := V_1$; $V_1 :=$ COLLECT$_p()$
75: **if** $(r(V_1).usqno = r(V_2).usqno)$ **then**
76: **return** $r(V_1).val$ // **direct scan**
77: **if** $\exists q$ *such that* $\langle p, ssqno\rangle \in V_1(q).scounts$ **then**
78: **return** $V_1(q).sview$ // **borrowed scan**

When UPDATE$_p(v)$ occurs:
79: *scounts* := COLLECT$_p()$.*ssqno*
80: *sview* := SCAN$_p()$ // **embedded scan**
81: *val* := v
82: *usqno*++
83: STORE$_p(\langle val, usqno, -, sview, scounts\rangle)$
84: **return** ACK

To prove linearizability, we consider an execution and specify an ordering of all the completed scans and all the updates whose store on Line 83 started. The ordering takes into consideration the embedded scans, which are inside updates, as well as the "free-standing" scans; since scans do not change the state of the atomic snapshot object, it is permissible to do so.

We first show that the snapshot views returned by direct scans are comparable in the following order: Let W_1 be the snapshot view returned by a direct scan based on the collect view V_1 (cf. Line 76) and W_2 be the snapshot view returned by a direct scan based on the collect view V_2 (cf. Line 76). We define $W_1 \preceq W_2$ if for every $\langle p, v\rangle \in W_1$, there exists $\langle p, v'\rangle \in W_2$ where the *usqno* associated with v in V_1 is less than or equal to the *usqno* associated with v' in V_2.

Lemma 18. *If a direct scan by node p returns W_1 and a direct scan by node q returns W_2, then either $W_1 \preceq W_2$ or $W_2 \preceq W_1$.*

Proof. Let cop_p^1, returning V_1', followed by cop_p^2, returning V_1, be the successful double collect at the end of p's direct scan and let cop_q^1, returning V_2', followed by cop_q^2, returning V_2, be the successful double collect at the end of q's direct scan. Note that $W_1 = r(V_1).val$, which is equal to $r(V_1').val$, and similarly $W_2 = r(V_2).val = r(V_2').val$.

Case 1: cop_p^1 completes before cop_q^2 starts. Consider any $\langle r, w\rangle \in W_2$. Then $\langle r, v\rangle$ is in both V_1' and V_1, with $v.val = w$ and $v.usqno > 0$. By regularity of store-collect, $V_1' \preceq V_2$. Thus there is an entry $\langle r, v'\rangle \in V_2$ such that either $v = v'$ or v' is

stored by r after v is stored by r. Since the *usqno* variable at r takes on increasing values, $v.usqno \preceq v'.usqno$. Thus there is an entry $\langle r, w' \rangle \in W_2$ where the *usqno* associated with w' is at least as large as that associated with w. Hence $W_1 \preceq W_2$.

Case 2: cop_q^1 completes before cop_p^2 starts. An analogous argument shows that $W_2 \preceq W_1$. □

Consider all direct scans in the order they complete and place them by the comparability order. Suppose a direct scan returning snapshot view W_1, obtained from collect view V_1, precedes another direct scan returning snapshot view W_2, obtained from collect view V_2. The regularity of store-collect ensures that $V_1 \preceq V_2$, and thus $W_1 \preceq W_2$. Hence, this ordering preserves the real-time order of non-overlapping direct scans.

The next lemma helps to order borrowed scans. Its statement is based on the observation that if a scan sop_p by node p borrows the snapshot view in $V_1(q)$, then there is an update uop_q by q that writes this view (via a store).

Lemma 19. *If a scan sop_p by node p borrows from a scan sop_q by node q, then sop_q starts after sop_p starts and completes before sop_p completes.*

Proof. Let uop_q be the update in which sop_q is embedded. Since sop_p borrows the snapshot view of sop_q, its *ssqno* appears in *scounts* of q's value in the view collected in Line 74. The properties of store-collect imply that the collect of uop_q (Line 79) does not complete before the store of p (Line 71) starts. Hence, sop_q (called in Line 80) starts after sop_p starts. Furthermore, since the collect of p returns the snapshot view stored after sop_q completes (Line 83), sop_q completes before sop_p completes. □

For every borrowed scan sop_1, there exists a chain of scans sop_2, sop_3, …, sop_k such that sop_i borrows from sop_{i+1}, $1 \leq i < k$, and sop_k is a direct scan *from which sop_1 borrows*. Consider all borrowed scans in the order they complete and place each borrowed scan after the direct scan it borrows from, as well as all previously linearized borrowed scans that borrow from the same direct scan. Applying Lemma 19 inductively, sop_k starts after sop_1 starts and completes before sop_1 completes, i.e., the direct scan from which a scan borrows is completely contained, in the execution, within the borrowing scan. This fact, together with the rule for ordering borrowed scans, implies that the real-time order of any two scans, at least one of which is borrowed, is preserved since direct scans have already been shown to be ordered properly.

Finally, we consider all updates in the order their stores (Line 83) start. Place each update, say *uop* by node p with argument v, immediately before the first scan whose returned view includes $\langle p, v' \rangle$, where either $v' = v$ or v' is the argument of an update by p that follows *uop*. If there is no such scan, then place *uop* at the end of the ordering. Note that all later scans return snapshot views that include $\langle p, v' \rangle$, where either $v' = v$ or v' is the argument of an update by p that follows *uop*. This rule for placing updates ensures that the ordering satisfies the sequential specification of atomic snapshots.

Note that if a scan completes before an update starts, then the scan's returned view cannot include the update's value; similarly, if an update completes before a scan starts, then the scan's returned view must includes the update's value or a later one. This shows that the ordering respects the real-time order between non-overlapping updates and scans. The next lemma deals with non-overlapping updates.

Lemma 20. *Let V be the snapshot view returned by a scan sop. If $V(p)$ is the value of an update uop_p by node p and an update uop_q by node q precedes uop_p, then $V(q)$ is the value of uop_q or a later update by q.*

Proof. Let sop' be *sop* if *sop* is a direct scan and otherwise the direct scan from which *sop* borrows. Let W be the (store-collect) view returned by the last two collects, cop_1 and cop_2, of sop'.

We now show that $V = W.val$. If $sop' = sop$, then $V = W.val$ by Line 76, since *sop* is a direct scan. Otherwise, $V = W.val$ because $W.val$ is returned to the enclosing scan, assigned to *sview*, and then stored (cf. Lines 80 and 83). This snapshot view is then propagated through the chain of borrowed-from scans and their enclosing updates until reaching *sop* where it is returned as V.

Since V includes the value of uop_p, so does W. It follows that both stores of uop_p start before cop_1 completes and thus before cop_2 starts. Since uop_q precedes uop_p, the store of uop_q at Line 83 completes before either store of uop_p starts. Thus the store of uop_q completes before cop_2 starts, and by the store-collect property, the view W returned by cop_2 must include the value of uop_q or a later update by q. Since $V = W.val$, the same is true for V. □

Consider an update uop_p, by node p, that follows an update uop_q, by node q, in the execution. If uop_p is placed at the end of the (current) ordering because there is no scan that observes its value or a later update by p, then it is ordered after uop_q. If uop_p is placed before a scan, then the same must be true of uop_q. By construction, the next scan after uop_p in the ordering, call it *sop*, returns view V with $V(p)$ equal to the value of uop_p or a later update by p. By Lemma 20, $V(q)$ must equal the value of uop_q or a later update by q. Thus uop_q cannot be placed after *sop*, and thus it is placed before uop_p.

We now consider the termination property of the algorithm. Suppose scan sop_q by node q contains two consecutive collects cop_1, which returns V_1, followed by cop_2, which returns V_2, and this double collect is unsuccessful. Then $V_1(p).usqno$ is not equal to $V_2(p).usqno$ for some node p and V_2 does not contain an *scounts* that includes sop_q's scan sequence number

ssqno. Thus, there exists an update uop_p by node p that is observed by V_2 but not by V_1. The correctness of store-collect implies that uop_p finishes after V_1 starts. Yet uop_p does not include sop_q's *ssqno*, which means that uop_p starts before the store at the beginning of sop_q completes. Let t be the time when the store at the beginning of sop_q completes and recall that $N(t)$ denotes the number of nodes present at time t. Thus at most $N(t)$ updates are pending at time t, implying that sop_q has at most $N(t)$ unsuccessful double collects before it can borrow a scan view. Hence, UPDATE executes at most $O(N(t))$ collects and stores. Putting the pieces together, we have:

Theorem 21. *Algorithm 7 is a linearizable implementation of an atomic snapshot object. The number of communication rounds in a* SCAN *or an* UPDATE *operation is at most linear in the number of nodes present in the system when the operation starts.*

6.3. Generalized lattice agreement

Let $\langle L, \sqsubseteq \rangle$ be a lattice, where L is the domain of lattice values, ordered by $\sqsubseteq$. We assume a join operator, $\sqcup$, that merges lattice values. A node p calls a PROPOSE operation with a lattice input value, and gets back a lattice output value. The input to p's i-th PROPOSE is denoted v_i^p and the response is w_i^p. The following conditions are required:

Validity Every response value w_i^p is the join of some values proposed before this response, including v_i^p, and all values returned to any node before the invocation of p's i-th PROPOSE.
Consistency Any two values w_i^p and w_j^q are comparable.

This definition is a direct extension of *one-shot* lattice agreement [9], following [24]. The version studied in [17] is weaker and lacks real-time guarantees across nodes.

Algorithm 8 uses an atomic snapshot object, in which each node stores a single lattice value (*val*), initialized to $\emptyset$. A PROPOSE operation is simply an UPDATE of a lattice value which is the join of all the node's previous inputs, followed by a SCAN returning the analogous values for all nodes, whose join is the output of PROPOSE.

Algorithm 8 Generalized lattice agreement: code for node p.

```
When PROPOSE_p(v) occurs:
85: val := val ⊔ v      // track previous inputs of p
86: UPDATE_p(val)
87: sview := SCAN_p()
88: return ⊔sview.val
```

To show validity of a PROPOSE operation *op*, first note that its output value is the join of values obtained from the atomic snapshot object, including *op*'s own input value. By the specification of atomic snapshot, the obtained values are those proposed by other PROPOSE operations that start before *op* finishes. The output is also the join of all values returned before *op* begins for the following reason: Suppose op' finishes before *op* begins and returns v'. We argue that every value contributing to v' is also observed by *op*. Suppose the input v'' of op'' is in the scan by op' and thus used to compute v'. Then the update by op'' is linearized before the scan of op'. Since op' finishes before *op* begins, the scan of op' is linearized before the scan of *op*. Therefore the update of op'' is linearized before the scan of *op*. Thus, v'' is joined into the value returned by *op*.

To show consistency, consider two views V_1 and V_2 used to produce two return values of PROPOSE. By the atomic snapshot properties, the sets of values in these two views are comparable (by containment), and hence their joins are comparable in the lattice.

Clearly, the algorithm terminates within $O(N)$ collects and stores, where N is the maximum number of nodes concurrently active during the execution of PROPOSE. Since PROPOSE includes one UPDATE and one SCAN, it terminates if the node does not crash or leave.

7. Conclusion

We have advocated for the usefulness of the store-collect object as a powerful, flexible, and efficient primitive for implementing a variety of shared objects in dynamic systems with continuous churn. We presented a simple churn-tolerant implementation of store-collect in which the store operation completes within one round trip and the collect operation completes within two. We presented an algorithm for atomic snapshots and another one for generalized lattice agreement using atomic snapshot. The good performance of the underlying store-collect carries over to the latter two problems, since the values can be collected in parallel rather than in series. We also described some simple implementations of non-linearizable objects (max register, abort flag, and set) using store-collect. This assortment of applications highlights the ability to choose whether we want to pay the price of linearizability or settle for the weaker "regularity" condition of store-collect.

If the level of churn is too great, our store-collect algorithm is not guaranteed to preserve the safety property; that is, a collect might miss the value written by a previous store, essentially by the same counter-example as that given in [7]. This

behavior is in contrast to the algorithms in [2,20,24], which never violate the safety property but only ensure progress once reconfigurations cease. In future work, we would like to either improve our algorithm to avoid this behavior or prove that any algorithm that tolerates ongoing churn is subject to such bad behavior.

Our correctness proof for our store-collect algorithm requires that the parameters defining the churn rate and failure fraction satisfy certain conditions. These conditions imply that even in the absence of churn the failure fraction tolerable by our algorithm is smaller than in the static case (namely, less than one-third versus less than one-half). Some degradation is unavoidable when allowing for the possibility of churn, since an argument from [7] can be adapted to show that when implementing store-collect in a system with churn rate α, the fraction of failures must be less than $1/(\alpha + 2)$. It would be nice to find less restrictive constraints on the parameters, either through a better analysis or a modified algorithm, or to show that they are necessary.

Another desirable modification to the store-collect algorithm would be reducing the size of the messages and the amount of local storage by garbage-collecting the *Changes* sets. In the same vein, we would like to know if modifying the atomic snapshot specification to remove from returned views entries of nodes that have left, as is done in [27], can lead to a more space-efficient algorithm.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We thank Luis Pantin for helpful comments.

References

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, Nir Shavit, Atomic snapshots of shared memory, J. ACM 40 (4) (1993) 873–890.
[2] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, Alexander Shraer, Dynamic atomic storage without consensus, J. ACM 58 (2) (2011) 7.
[3] Marcos Kawazoe Aguilera, A pleasant stroll through the land of infinitely many creatures, SIGACT News 35 (2) (2004) 36–59.
[4] James Aspnes, Notes on theory of distributed systems, http://www.cs.yale.edu/homes/aspnes/classes/465/notes.pdf. (Accessed 8 August 2020).
[5] James Aspnes, Hagit Attiya, Keren Censor-Hillel, Polylogarithmic concurrent data structures from monotone circuits, J. ACM 59 (1) (2012) 1–24.
[6] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, Sharing memory robustly in message-passing systems, J. ACM 42 (1) (January 1995) 124–142.
[7] Hagit Attiya, HyunChul Chung, Faith Ellen, Saptaparni Kumar, Jennifer Welch, Emulating a shared register in an asynchronous system that never stops changing, IEEE Trans. Parallel Distrib. Syst. 30 (3) (2018) 544–559.
[8] Hagit Attiya, Arie Fouren, Eli Gafni, An adaptive collect algorithm with applications, Distrib. Comput. 15 (2) (2002) 87–96.
[9] Hagit Attiya, Maurice Herlihy, Ophir Rachman, Atomic snapshots using lattice agreement, Distrib. Comput. 8 (3) (1995) 121–132.
[10] Hagit Attiya, Sweta Kumari, Archit Somani, Jennifer L. Welch, Store-collect in the presence of continuous churn with application to snapshots and lattice agreement, in: 22nd International Symposium on Stabilization, Safety, and Security of Distributed Systems, in: Lecture Notes in Computer Science, vol. 12514, Springer, 2020, pp. 1–15.
[11] Roberto Baldoni, Silvia Bonomi, A.-M. Kermarrec, Michel Raynal, Implementing a register in a dynamic distributed system, in: 29th IEEE International Conference on Distributed Computing Systems, 2009, pp. 639–647.
[12] Roberto Baldoni, Silvia Bonomi, Michel Raynal, Implementing a regular register in an eventually synchronous distributed system prone to continuous churn, IEEE Trans. Parallel Distrib. Syst. 23 (1) (2012) 102–109.
[13] Roberto Baldoni, Silvia Bonomi, Michel Raynal, Implementing set objects in dynamic distributed systems, J. Comput. Syst. Sci. 82 (5) (2016) 654–689.
[14] Armando Castañeda, Sergio Rajsbaum, Michel Raynal, Unifying concurrent objects and distributed tasks: interval-linearizability, J. ACM 65 (6) (2018) 1–42.
[15] Sagar Chordia, Sriram Rajamani, Kaushik Rajan, Ganesan Ramalingam, Kapil Vaswani, Asynchronous resilient linearizability, in: 27th International Symposium on Distributed Computing, in: Lecture Notes in Computer Science, vol. 8205, Springer, 2013, pp. 164–178.
[16] Carole Delporte-Gallet, Hugues Fauconnier, Sergio Rajsbaum, Michel Raynal, Implementing snapshot objects on top of crash-prone asynchronous message-passing systems, IEEE Trans. Parallel Distrib. Syst. 29 (9) (2018) 2033–2045.
[17] Jose M. Faleiro, Sriram Rajamani, Kaushik Rajan, G. Ramalingam, Kapil Vaswani, Generalized lattice agreement, in: Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing, 2012, pp. 125–134.
[18] Eli Gafni, Dahlia Malkhi, Elastic configuration maintenance via a parsimonious speculating snapshot solution, in: 29th International Symposium on Distributed Computing, in: Lecture Notes in Computer Science, vol. 9363, Springer, 2015, pp. 140–153.
[19] Eli Gafni, Michael Merritt, Gadi Taubenfeld, The concurrency hierarchy, and algorithms for unbounded concurrency, in: Proceedings of the 2001 ACM Symposium on Principles of Distributed Computing, 2001, pp. 161–169.
[20] Seth Gilbert, Nancy A. Lynch, Alexander A. Shvartsman, Rambo: a robust, reconfigurable atomic memory service for dynamic networks, Distrib. Comput. 23 (4) (2010) 225–272.
[21] Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, Dynamic byzantine reliable broadcast, [technical report]. CoRR, arXiv:2001.06271 [cs.DC], 2020.
[22] Maurice P. Herlihy, Jeannette M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Trans. Program. Lang. Syst. 12 (3) (1990) 463–492.
[23] Leander Jehl, Roman Vitenberg, Hein Meling, Smartmerge: a new approach to reconfiguration for atomic storage, in: 29th International Symposium on Distributed Computing, in: Lecture Notes in Computer Science, vol. 9363, Springer, 2015, pp. 154–169.
[24] Petr Kuznetsov, Thibault Rieutord, Sara Tucci-Piergiovanni, Reconfigurable lattice agreement and applications, in: 23rd International Conference on Principles of Distributed Systems, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 153, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 31.
[25] Leslie Lamport, On interprocess communication. Part I: basic formalism, Distrib. Comput. 1 (2) (1986) 77–85.

[26] Michael Merritt, Gadi Taubenfeld, Computing with infinitely many processes, in: 14th International Symposium on Distributed Computing, in: Lecture Notes in Computer Science, vol. 1914, Springer, 2000, pp. 164–178.
[27] Alexander Spiegelman, Idit Keidar, Dynamic atomic snapshots, in: 20th International Conference on Principles of Distributed Systems, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 70, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 33.
[28] Xiong Zheng, Vijay Garg, John Kaippallimalil, Linearizable replicated state machines with lattice agreement, in: 23rd International Conference on Principles of Distributed Systems, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 153, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 29.
[29] Xiong Zheng, Changyong Hu, Vijay Garg, Lattice agreement in message passing systems, in: 32nd International Symposium on Distributed Computing, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 121, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 41.