Store-Collect in the Presence of Continuous Churn with Application to Snapshots and Lattice Agreement

Hagit Attiya[1(✉)], Sweta Kumari[1], Archit Somani[1], and Jennifer L. Welch[2]

[1] Department of Computer Science, Technion, Haifa, Israel
{hagit,sweta,archit}@cs.technion.ac.il
[2] Department of Computer Science and Engineering, Texas A&M University, College Station, TX, USA
welch@cse.tamu.edu

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.

Keywords: Store-collect object · Dynamic message-passing systems · Churn · Crash resilience · Atomic snapshots · Generalized lattice agreement

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 techniques. 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 usefulness of shared memory programming abstractions has been long established for static systems (e.g., [4,5]), which have known bounds on the number of fixed computing nodes and the number of possible failures. This success has inspired work

Supported by ISF grant 380/18 and NSF grant 1816922; full paper in [9].

S. Devismes and N. Mittal (Eds.): SSS 2020, LNCS 12514, pp. 1–15, 2020.
https://doi.org/10.1007/978-3-030-64348-5_1

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., [2,6,10,11,18]). We discuss a couple of exceptions [12,21] below.

In this paper, we promote the *store-collect* shared object [7] (defined in Sect. 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 [6], 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 [6]. 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 Sect. 3 for model details.)

Our algorithm for implementing a churn-tolerant store-collect object is based on the read-write register algorithm in [6]. 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 [19]. In contrast to our single-round-trip store operation, the write operation in the algorithm of [6] 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 [6], consequently making it easier to build on the results. (See Sect. 4.)

Building an atomic snapshot on top of a store-collect object is easy! We present a simple algorithm with an elegant correctness proof (Sect. 5.1). One may be tempted to implement an atomic snapshot in our model by plugging churn-tolerant registers (e.g., [6]) 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. A

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 knowing when to borrow a scan 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 (Sect. 5.2) how a churn-tolerant atomic snapshot object can be used to provide a churn-tolerant generalized lattice agreement object [15]. 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 [8]. Generalized lattice agreement has been used to implement many objects [13,15], most notably, *conflict-free replicated data types* [21,24,25].

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 is hidden from higher layers by our store-collect implementation. As evidence, we observe in [9] 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 [21] 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 in [14]. 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 in [17,22], which provided algorithms for election, mutual exclusion, consensus, collect, snapshot, and renaming. Spiegelman and Keidar [23] present atomic snapshot 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 can be done in [23].

The problem of implementing shared objects in the presence of *ongoing* churn and crash failures in *message-passing systems* is studied in [10,11], for read-write registers, and [12], for 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 in [12] 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., [2,16,18,20,21]). Explicit reconfiguration operations replace older configurations with newer ones. The assumptions made in [2,16,18,20,21] are incomparable with those in [6] 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. [8, 13,15,24,25]. A notable exception is [21], which considers dynamic systems subject to changes in the composition due to reconfiguration. This 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 in [21] require that changes to the system composition eventually cease in order to ensure progress.

2 The Store-Collect Problem

A shared *store-collect object* [7] 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.

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).

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 for every $\langle p, v_1\rangle \in V_1$, there exists v_2 such that $\langle p, v_2\rangle \in V_2$ where either $v_1 = v_2$ or the $\text{STORE}_p(v_1)$ invocation occurs before the $\text{STORE}_p(v_2)$ invocation in σ. We denote this as $V_1 \preceq V_2$.

3 Overview of System Model

The events that can occur at a node p are entering the system (ENTER_p), leaving the system (LEAVE_p), crashing (CRASH_p), receiving a message m ($\text{RECEIVE}_p(m)$), and invoking an operation (COLLECT_p or $\text{STORE}_p(v)$). The occurrence of an event at node p results in changes to p's local state; optionally, a message to be broadcast; and optionally, a response, which is $\text{RETURN}_p(V)$ for a collect, ACK_p for a store, and JOINED_p for the enter. An *execution* is a collection of sequences of events, one sequence for each node, that satisfies certain conditions. The key points are the following.

A node enters, leaves, and crashes at most once. A node does nothing before it enters and after it crashes or leaves.

We assume that a nonnegative real number is associated with each event in an execution, which is the time when the event occurs. A node is *present* at time t if it entered but did not leave before time t; a crashed node is considered to be present. $N(t)$ is the number of nodes present at time t. There is a constant N_{min} such that $N(t) \geq N_{min}$ for all $t \geq 0$. A node is *active* at time t if it is present and not crashed at time t.

At time 0, the system consists of a finite nonempty set of nodes, S_0, that are considered by definition to be active. Initially nodes in S_0 have knowledge about all the nodes in S_0; every other node, which enters after time 0, has no initial knowledge about any node other than itself. A node is *joined* (or a *member*) at time t if it is present (has not left) and either is in S_0 or has experienced JOINED_p.

A broadcast service reliably delivers each message sent by node p at time t to each node q that is active throughout $[t, t + D]$, where $D > 0$ is a maximum message delay unknown to the nodes, if p's next event is not CRASH_p. If p's next event is to crash or if q enters, leaves, or crashes during the interval, there is no guarantee whether q receives the message. This is a weaker broadcast specification than in [6], which assumed broadcasts were atomic with respect to crashes. Messages from the same sender are received in the order they are sent. Every message that is received has delay in $(0, D]$.

Let $\alpha > 0$ and $0 < \Delta < 1$ be real numbers that denote the *churn rate* and *failure fraction*, respectively. The parameters α and Δ are known to the nodes. For all times $t \geq 0$, there are at most $\alpha \cdot N(t)$ enter and leave events in $[t, t + D]$ (*churn assumption*), and at most $\Delta \cdot N(t)$ nodes are crashed at time t (*failure fraction assumption*).

An algorithm is a *correct implementation of a store-collect object* if in every execution: *(1)* Every node that enters the system after time 0 and remains active eventually joins, and no joined node p experiences JOINED_p. *(2)* Every store or collect operation invoked at a node that remains active eventually completes. *(3)* The schedule resulting

from restricting 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.

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 picks the latest value in each view. That is, given two views V_1 and V_2, $merge(V_1, V_2)$ is 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 $V_1, V_2 \preceq merge(V_1, V_2)$.

A node p tracks the composition of the system with a set *Changes* of events concerning the nodes that have entered the system. 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. The code for managing these sets (Algorithm 1) is the same as [6] 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 handles *collect* and *store* operations (Algorithm 2) and its server thread (Algorithm 3) responds 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 it considers as members, i.e., nodes that have joined but not left.

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 29). When a server node p receives a **collect-query** message, it responds with its local view (*LView*) through a **collect-reply** message (Line 53) 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 31). Then the client waits for sufficiently many **collect-reply** messages before broadcasting the current value of its *LView* variable in a **store** message (Line 36). When server p receives a **store** message with a view *RView*, it merges *RView* with its local *LView* (Line 50) and, if p is joined, it broadcasts **store-ack** (Line 48). The client waits for sufficiently many **store-ack** messages before returning *LView* to complete the *collect* (Line 47); this threshold is recalculated in Line 34 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 39) and broadcasts a **store** message (Line 42). When server p receives a **store** message with view *RView*, it merges *RView* with its local *LView* (Line 48) and, if p is joined, it broadcasts **store-ack** (Line 50). The client waits for sufficiently many **store-ack** messages before completing the *store* (Line 46).

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)* $\in$ Changes$\wedge$ *leave(q)* $\notin$ *Changes*$\}$

When ENTER$_p$ **occurs:**
1: **add** *enter(p)* **to** *Changes*
2: **broadcast** $\langle$**enter,** $p\rangle$

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

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

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

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

When LEAVE$_p$ **occurs:**
21: **broadcast** $\langle$**leave,** $p\rangle$
22: **halt**

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

When RECEIVE$_p\langle$**leave-echo,** $q\rangle$ **occurs:**
25: **add** *leave(q)* **to** *Changes*

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: *(1)* lines 26 through 33, the first part of a *collect* operation, *(2)* lines 34 through 36 and 43 through 47, the second part of a *collect* operation called the "store-back", or *(3)* lines 37 through 46, 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*(φ) 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.

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:**
26: *optype* $=$ *collect; tag*++
27: *threshold* $= \beta \cdot |Members|$
28: *counter* $= 0$
29: **broadcast** $\langle$**collect-query,** *tag*, $p\rangle$

When $\text{RECEIVE}_p\langle$**collect-reply,** *RView, t, q*$\rangle$ **occurs:**
30: **if** $(t == tag) \wedge (q == p)$ **then**
31: *LView* **=** *merge(LView, RView)*
32: *counter*++
33: **if** (*counter* $\geq$ *threshold*) **then**
34: $threshold = \beta \cdot |Members|$
35: *counter* $= 0$
36: **broadcast** $\langle$**store,** *LView, tag, p*$\rangle$

When $\text{STORE}_p(v)$ **occurs:**
37: *optype* $=$ *store; tag*++
38: *sqno*++
39: *LView* $=$ *merge(LView,*$\{\langle p, v, sqno\rangle\}$)
40: *threshold* $= \beta \cdot |Members|$
41: *counter* $= 0$
42: **broadcast** $\langle$**store,** *LView, tag, p*$\rangle$

When $\text{RECEIVE}_p\langle$**store-ack,** *t*, $q\rangle$ **occurs:**
43: **if** $(t == tag) \wedge (q == p)$ **then**
44: *counter*++
45: **if** (*counter* $\geq$ *threshold*) **then**
46: **if** (*optype*$==$*store*) **then return** ACK
47: **else return** *LView*

Algorithm 3. CCC—Server code, for node p.

When $\text{RECEIVE}_p\langle$**store,** *RView,tag*, $q\rangle$ **occurs**
48: *LView* $=$ *merge(LView, RView)*
49: **if** *is_joined* **then**
50: **broadcast** $\langle$**store-ack,** *tag*, $q\rangle$
51: **broadcast** $\langle$**store-echo,** *LView*$\rangle$

When $\text{RECEIVE}_p\langle$**collect-query,** *tag*, $q\rangle$ **occurs:**
52: **if** *is_joined* **then**
53: **broadcast** $\langle$**collect-reply,** *LView, tag*, $q\rangle$

When $\text{RECEIVE}_p\langle$**store-echo,** *RView*$\rangle$ **occurs:**
54: *LView* $=$ *merge(LView, RView)*

To prove the correctness of the algorithm, consider any execution of the algorithm. The correctness of the algorithm relies on the following constraints ($Z = \left[(1-\alpha)^3 - \Delta \cdot (1+\alpha)^3\right]$, which is the fraction of nodes that survive 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. The following technical claims hold:

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) $N(t + i \cdot D) \leq (1+\alpha)^i \cdot N(t)$.*

Lemma 2. *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.*

Below, 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 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 lemma holds:

Lemma 3. *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 $[0, \max\{0, t - 2D\}]$.*

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

Theorem 1. *Every node p that enters at some time t and is active for at least $2D$ time joins by time $t + 2D$.*

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

Proof sketch. Consider a phase invoked by node p at time t. Let S be the set of nodes present at time $\max\{0, t - 2D\}$. Lemma 2 and Theorem 1 imply that at least $Z \cdot |S|$ of them are joined by time t and active at time $t + D$, and thus respond to p's message. We argue that $Z \cdot |S|$ is at least as large as the value of *threshold* computed by p in Line 27 or 34 or 40 of Algorithm 2. We first show that $|S| \geq |Present_p^t|/(1+\alpha)^2$, by Lemmas 3 and 1(a). Then we note that Constraint C implies that $Z/(1+\alpha)^2 \geq \beta$, and so $Z \cdot |S| \geq \beta \cdot |Present_p^t|$. Since $|Present_p^t| \geq |Members_p^t|$, the definition of *threshold* gives the result. □

The next lemma shows that the view of a store phase is smaller, in the partial order $\preceq$, than the view of a subsequent, non-overlapping, collect phase.

Lemma 4. *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).*

Theorem 3. *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 4, $\mathit{view}(s) \preceq \mathit{view}(c)$. Hence, $\mathit{view}(s)$ contains a tuple for p with a non-$\perp$ value, which is a contradiction.

Therefore, $V(p) = v \neq \perp$. We show that a $\text{STORE}_p(v)$ 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 $\text{STORE}_p(v)$ invocation.

Now suppose for the sake of contradiction that the $\text{STORE}_p(v)$ 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 $\mathit{view}(sop) \preceq \mathit{view}(sop')$. By Lemma 4, $\mathit{view}(sop') \preceq \mathit{view}(c) = V$. 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 4, $\mathit{view}(s) \preceq \mathit{view}(c)$. Regularity holds since $\mathit{view}(s) = V_1$ and $\mathit{view}(c) = V_2$, implying that $V_1 \preceq V_2$. □

By Theorem 1, 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 2, every operation eventually completes as long as the invoker remains active. Finally, Theorem 3 ensures regularity.

Corollary 1. CCC *is a correct implementation of a store-collect object, in which each* Store *or* Collect *completes within a constant number of communication rounds.*

5 Implementing Distributed Objects Despite Continuous Churn

We show implementations of two objects using store-collect. Three additional objects are discussed in [9]. For all applications, we assume that the conditions for store-collect termination hold, which guarantees termination of the operations.

5.1 Atomic Snapshots

Like other atomic snapshot algorithms [1,14,23], 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. 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.

An *atomic snapshot* 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* [19]. Roughly speaking, for every execution α, we should find a sequence of operations, containing all completed operations in α and some of the pending operations, which is in the sequential specification of an atomic snapshot, and 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 $\langle p, v.comp\rangle$; when $v.comp = \perp$, the tuple is omitted. 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).

To execute a SCAN, Algorithm 4 increments the scan sequence number (*ssqno*) (Line 55) 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 57). In a while loop, the last collected view is saved and a new view is collected (Line 59). If the two most recently collected views are equal (Line 60), the latest collected view is returned (Line 61). 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 62). If this condition holds, the snapshot view of q is returned (Line 63); 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 64). Next, the value of an embedded scan is saved in a local variable *sview* (Line 65). Then it sets its *val* variable to the argument value and increments its update sequence number (Lines 66 and 67). 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 68).

To prove linearizability, we consider an execution and specify an ordering of all the completed scans and all the updates whose store on Line 68 takes effect. 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 direct scans are comparable in the $\preceq$ order.

Algorithm 4. 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 $\text{SCAN}_p()$ **occurs:**
55: *ssqno*++
56: $\textbf{STORE}_p(\langle -, -, ssqno, -, - \rangle)$
57: $V_1 = \textbf{COLLECT}_p()$
58: **while** *true* **do**
59: $V_2 = V_1$; $V_1 = \textbf{COLLECT}_p()$
60: **if** $(V_1 == V_2)$ **then**
61: **return** $V_1.val$ **// direct scan**
62: **if** $\exists q$ *such that* $\langle p, ssqno \rangle \in V_1(q).scounts$ **then**
63: **return** $V_1(q).sview$ **// borrowed scan**

When $\textbf{UPDATE}_p(v)$ **occurs:**
64: $scounts = \textbf{COLLECT}_p().ssqno$
65: $sview = \textbf{SCAN}_p()$ **// embedded scan**
66: $val = v$
67: *usqno*++
68: $\textbf{STORE}_p(\langle val, usqno, -, sview, scounts \rangle)$
69: **return** ACK

Lemma 5. *If a direct scan by node p returns V_1 and a direct scan by node q returns V_2, then either $V_1 \preceq V_2$ or $V_2 \preceq V_1$.*

Proof. Let cop_p^1 and cop_p^2 be the last two collects of p (both returning V_1), and cop_q^1 and cop_q^2 be the last two collects of q (both returning V_2). We have that either cop_p^1 completes before cop_q^2 starts or cop_q^1 completes before cop_p^2 starts. In the former case, by the regularity of store-collect, $V_1 \preceq V_2$, while in the latter case, $V_2 \preceq V_1$. □

Consider all direct scans in the order they complete and place them by the comparability order. If a direct scan returning snapshot view V_1 precedes another direct scan returning snapshot view V_2, then the regularity of store-collect ensures $V_1 \preceq V_2$. 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 6. *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 59. The properties of store-collect imply that the collect of uop_q (Line 64) does not complete before the store of p (Line 56) starts. Hence, sop_q (called in Line 65) starts after sop_p starts. Furthermore, since the collect of p returns the snapshot view stored after sop_q completes (Line 68), 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 6 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 68) 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 7. *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.*

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 7, $V(q)$ 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. Let V_1 and V_2 be two collect views returned by consecutive collects *cop*$_1$ and *cop*$_2$, within a scan *sop*$_q$ by node q. If this double collect is not successful, then $V_1 \neq V_2$. If $V_1(p).usqno \neq V_2(p).usqno$, then it is immediate that for some update *uop*$_p$ of node p, either *uop*$_p$'s *scounts* includes the scan sequence number of *sop*$_q$, or *uop*$_p$ starts before *sop*$_q$ starts. Let t be the time that *sop*$_q$ starts, and note that at most $N(t)$ updates are pending at time t. Further note that if *uop*$_p$'s *scounts* includes the scan sequence number of *sop*$_q$, then *sop*$_q$ can borrow the scan view of *uop*$_p$'s embedded scan. This implies that *sop*$_q$ has at most $N(t)$ unsuccessful double collects before it can borrow a scan view, and therefore it executes at most $O(N(t))$ collects. Hence, UPDATE executes at most $O(N(t))$ collects and stores. Putting the pieces together, we have:

Theorem 4. *Algorithm 4 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.*

5.2 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: *(1)* 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 (*validity*). *(2)* Any two values w_i^p and w_j^q are comparable (*consistency*). This definition is a direct extension of *one-shot* lattice agreement [8], following [21]. The version studied in [15] is weaker and lacks real-time guarantees across nodes.

Our algorithm uses an atomic snapshot object, in which each node stores a single lattice value (*val*). 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.

Validity and consistency are immediate from atomic snapshot properties. 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.

6 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. 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 [6]. This behavior is in contrast to the algorithms in [2,18,21], 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 [6] 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 [23], can lead to a more space-efficient algorithm.

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. J. ACM **40**(4), 873–890 (1993)
2. Aguilera, M.K., Keidar, I., Malkhi, D., Shraer, A.: Dynamic atomic storage without consensus. J. ACM **58**(2), 7:1–7:32 (2011)
3. Aguilera, M.K.: A pleasant stroll through the land of infinitely many creatures. SIGACT News **35**(2), 36–59 (2004)
4. Aspnes, J.: Notes on theory of distributed systems. http://www.cs.yale.edu/homes/aspnes/classes/465/notes.pdf. Accessed 8 Aug 2020
5. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM **42**(1), 124–142 (1995)
6. Attiya, H., Chung, H., Ellen, F., Kumar, S., Welch, J.: Emulating a shared register in an asynchronous system that never stops changing. TPDS **30**(3), 544–559 (2018)
7. Attiya, H., Fouren, A., Gafni, E.: An adaptive collect algorithm with applications. Dist. Comp. **15**(2), 87–96 (2002)
8. Attiya, H., Herlihy, M., Rachman, O.: Atomic snapshots using lattice agreement. Dist. Comp. **8**(3), 121–132 (1995)
9. Attiya, H., Kumari, S., Somani, A., Welch, J.L.: Store-collect in the presence of continuous churn with application to snapshots and lattice agreement. CoRR abs/2003.07787 (2020). https://arxiv.org/abs/2003.07787
10. Baldoni, R., Bonomi, S., Kermarrec, A.M., Raynal, M.: Implementing a register in a dynamic distributed system. In: ICDCS, pp. 639–647 (2009)
11. Baldoni, R., Bonomi, S., Raynal, M.: Implementing a regular register in an eventually synchronous distributed system prone to continuous churn. TPDS **23**(1), 102–109 (2012)
12. Baldoni, R., Bonomi, S., Raynal, M.: Implementing set objects in dynamic distributed systems. J. Comput. Syst. Sci. **82**(5), 654–689 (2016)
13. Chordia, S., Rajamani, S., Rajan, K., Ramalingam, G., Vaswani, K.: Asynchronous resilient linearizability. In: DISC pp. 164–178 (2013)
14. Delporte-Gallet, C., Fauconnier, H., Rajsbaum, S., Raynal, M.: Implementing snapshot objects on top of crash-prone asynchronous message-passing systems. TPDS **29**(9), 2033–2045 (2018)
15. Faleiro, J.M., Rajamani, S., Rajan, K., Ramalingam, G., Vaswani, K.: Generalized lattice agreement. In: PODC, pp. 125–134 (2012)
16. Gafni, E., Malkhi, D.: Elastic configuration maintenance via a parsimonious speculating snapshot solution. In: DISC, pp. 140–153 (2015)
17. Gafni, E., Merritt, M., Taubenfeld, G.: The concurrency hierarchy, and algorithms for unbounded concurrency. In: PODC, pp. 161–169 (2001)
18. Gilbert, S., Lynch, N.A., Shvartsman, A.A.: RAMBO: a robust, reconfigurable atomic memory service for dynamic networks. Dist. Comp. **23**(4), 225–272 (2010)
19. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. Trans. Prog. Lang. Sys. **12**(3), 463–492 (1990)
20. Jehl, L., Vitenberg, R., Meling, H.: SmartMerge: a new approach to reconfiguration for atomic storage. In: DISC, pp. 154–169 (2015)
21. Kuznetsov, P., Rieutord, T., Tucci-Piergiovanni, S.: Reconfigurable lattice agreement and applications. In: OPODIS, pp. 31:1–31:17 (2019)
22. Merritt, M., Taubenfeld, G.: Computing with infinitely many processes. In: DISC, pp. 164–178 (2000)
23. Spiegelman, A., Keidar, I.: Dynamic atomic snapshots. In: OPODIS (2016)
24. Zheng, X., Garg, V., Kaippallimalil, J.: Linearizable replicated state machines with lattice agreement. In: OPODIS (2019)
25. Zheng, X., Hu, C., Garg, V.: Lattice agreement in message passing systems. In: DISC (2018)