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

Hagit Attiya¹, Sweta Kumari¹, Archit Somani¹, and Jennifer L. Welch²

¹Department of Computer Science, Technion, Israel
{hagit, sweta, archit}@cs.technion.ac.il
²Department of Computer Science and Engineering, Texas A&M University
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. The store-collect object is not linearizable and can easily be used to implement several non-linearizable objects. Yet without much additional effort, it can also be used to implement atomic snapshot and generalized lattice agreement objects, which are linearizable. Thus the store-collect sits in a sweet spot combining algorithmic flexibility and efficiency.

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, the processes keep individual copies of the data and communicate by sending messages to keep the copies consistent. Traditional distributed computing considers a static system, with known bounds on the number of fixed computing nodes and the number of possible failures. Dynamic distributed systems allow 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. Motivating applications include those in peer-to-peer, sensor, mobile, and social networks, as well as server farms.

The usefulness of shared memory programming abstractions has been long-established for static systems (e.g., [6, 4]). 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., [10, 20, 3, 11, 7]). A couple of exceptions are [12] and [23].

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. We focus on the situation when nodes enter and leave, but the resulting network is always fully connected, which could be due to, say, an overlay network. We refer to this phenomenon as “churn”. We next discuss three main advantages of this object, showing that it combines algorithmic power and efficiency.

A churn-tolerant store-collect object can be implemented fairly easily. We adopt the system model 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 lower bound. Nodes do not know $D$ and have no local clocks, causing consensus to be
impossible to solve [7]. During any time interval of length $D$, the number of churn events that can occur is a fraction of the number of nodes in the system at the beginning of the interval. (See Section 3 for details.) Our algorithm for implementing a (non-linearizable) store-collect object is based on the (linearizable) 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 algorithm and its proof are given in Sections 4 and 5.) A by-product of our work in this paper is a revised proof of the churn management protocol that is much more accessible than that in [7].

The store-collect object specification is versatile, yielding efficient implementations for objects that are not linearizable. In Section 6.1 we provide simple algorithms to implement (non-linearizable) max-registers, abort flags, and sets using store-collect. The choice of problems and algorithms follow [23], but the algorithms inherit good efficiency properties from our store-collect implementation.

Building an atomic snapshot on top of a store-collect object is easy! We present a simple algorithm with an elegant correctness proof in Section 6.2. Our atomic snapshot algorithm is more efficient than one based on registers as the values can be collected in parallel instead of in series. In static systems, atomic snapshot objects can be used to build multi-writer registers, concurrent timestamp systems, counters, and accumulators, and to solve approximate agreement and randomized consensus (cf. [1, 4]). Aguilera [2] 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. This model was first considered in [17]. Variations of the model were proposed in [25, 19], which provided algorithms for election, mutual exclusion, consensus, collect, snapshot, and renaming.

In Section 6.3, we show that atomic snapshots yield a very simple implementation of generalized lattice agreement [16] in a dynamic system. This object supports a PROPOSE operation whose argument is a value belonging to a lattice. A PROPOSE returns 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]. It has been used to implement many objects [14, 10], including atomic snapshots [9] and conflict-free replicated data types [23, 28, 27]. Most of these papers assume static systems. A notable exception is [25], 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 [25] require that changes to the system composition eventually cease in order to ensure progress.

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.

The problem of implementing shared objects in the presence of ongoing churn and crash failures is studied in [10, 11], which consider read-write registers, and [12], 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 in [12] uses unbounded local memory at the nodes.

A popular alternative way to model churn 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., [20, 3, 22, 18, 23]). Explicit reconfiguration operations replace older configurations with newer ones. These papers all assume that reconfiguration requests eventually stop in order for liveness properties to hold.

2 The Store-Collect Problem

A shared store-collect object supports concurrent execution of 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 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 $\text{Ack}_p$, indicating that the operation has completed. For a $\text{collect}$ operation, the invocation is of the form $\text{Collect}_p$ and the response is of the form $\text{Return}_p(V)$, where $V$ is a $\text{view}$, that is, a set of client-value pairs without repetition of client ids. We use the notation $V(p)$ to indicate $v$ if $(p, v) \in V$ and $\bot$ if no pair in $V$ has $p$ as its first element.

Informally, the behavior required of a store-collect object is that each $\text{collect}$ operation should return a view containing the latest value stored by each client. However, we do not require the $\text{collect}$ operation to appear to occur instantaneously, that is, the object is not linearizable. Instead, we give a precise definition of the required behavior that is along the lines of interval linearizability [13] or the specification for regularity [24].

A sequence $\sigma$ of invocations and responses of $\text{store}$ and $\text{collect}$ operations is a $\text{schedule}$ if, for each client id $p$, the restriction of $\sigma$ 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 $\sigma$, then we say $op$ precedes $op'$ (in $\sigma$) and $op'$ follows $op$. We assume that every value written in a $\text{store}$ operation in a schedule is unique (a condition that can be achieved using sequence numbers and client ids).

A schedule $\sigma$ satisfies regularity for the store-collect problem if:

- For each $\text{collect}$ operation $cop$ in $\sigma$ that returns $V$ and every client $p$, if $V(p) = \bot$, then no $\text{store}$ operation by $p$ precedes $cop$ in $\sigma$. If $V(p) = v \neq \bot$, then there is a $\text{Store}_p(v)$ invocation that occurs in $\sigma$ before $cop$ completes and no other $\text{store}$ operation by $p$ occurs in $\sigma$ between this invocation and the invocation of $cop$.

- For every two $\text{collect}$ operations in $\sigma$, $cop_1$ which returns $V_1$ and $cop_2$ which returns $V_2$, if $cop_1$ precedes $cop_2$ in $\sigma$, then for every $(p, v_1) \in V_1$, there exists $v_2$ such that $(p, v_2) \in V_2$ where either $v_1 = v_2$ or the $\text{Store}_p(v_1)$ invocation occurs before the $\text{Store}_p(v_2)$ invocation in $\sigma$. We denote this as $V_1 \preceq V_2$.

3 System Model

We model each node $p$ as a state machine with a set of states, containing two initial states $s^i_p$ and $s^f_p$. Initial state $s^i_p$ is used if $p$ is initially in the system, whereas $s^f_p$ is used if $p$ enters the system later. The set of all nodes that are initially in the system is denoted by $S_0$. It is finite and nonempty.

State transitions are triggered by the occurrences of events. Possible triggering events are: entering the system ($\text{Enter}_p$), leaving the system ($\text{Leave}_p$), receipt of a message $m$ ($\text{Receive}_p(m)$), invocation of an operation ($\text{Collect}_p$ or $\text{Store}_p(v)$), and crashing ($\text{Crash}_p$).

A step of a node $p$ is a 5-tuple $(s', T, m, R, s)$ where $s'$ is the old state, $T$ is the triggering event, $m$ is the message to be sent, $R$ is a response ($\text{Return}_p(V)$, $\text{Ack}_p$, or $\text{Joined}_p$) or $\bot$, and $s$ is the new state. 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 $\text{Collect}_p$, $\text{Ack}_p$ is the response to $\text{Store}_p$, and $\text{Joined}_p$ is the response to $\text{Enter}_p$. If $T$ is $\text{Crash}_p$, then $m$ is $\bot$ and $R$ is $\bot$.

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 $s^i_p$, then no $\text{Enter}_p$ event occurs;
- if the old state of the first step is $s^f_p$, then the triggering event in the first step is $\text{Enter}_p$ and there is no other occurrence of $\text{Enter}_p$; and
- at most one of $\text{Crash}_p$ and $\text{Leave}_p$ 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. A node that crashes and recovers, but loses its state, can re-enter with a new id. Because nodes cannot measure time, a node that crashes and recovers, retaining its state, can be treated as if no crash occurred.

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 at time \(t\) if it entered the system (i.e., its first step has time at most \(t\)) but has not left (i.e., \(\text{Leave}_p\) 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 \(\text{Crash}_p\) occurs at or before \(t\)) is still considered to be present. 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 that are initially members. Formally, the first step of each node \(p \in S_0\) occurs at time 0 and the first step of each other node occurs after time 0.

We assume a reliable broadcast communication service that provides nodes with a mechanism to send the same message to all nodes in the system; message delivery is FIFO. 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 at time \(t\) 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. The parameters \(\alpha\) and \(\Delta\) are known to the nodes, but \(D\) is not. We assume executions satisfy two assumptions:

**Churn Assumption** For all times \(t > 0\), there are at most \(\alpha \cdot N(t)\) \text{Enter} and \text{Leave} events in \([t, t + D]\).

**Failure Fraction Assumption** For all times \(t \geq 0\), at most \(\Delta \cdot N(t)\) nodes are crashed at time \(t\).

A node \(p\) is said to be a member at time \(t\) if it has joined the system (i.e., \(p \in S_0\) or \(\text{Joined}_p\) occurs at or before \(t\)) but has not left (i.e., \(\text{Leave}_p\) does not occur at or before \(t\)). Note that, at any time \(t\), the members are a subset of the present nodes. It is possible that some members have crashed.

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 \(\text{Enter}_p\) occurs, then at least one of \(\text{Leave}_p\), \(\text{Crash}_p\), or \(\text{Joined}_p\) occurs subsequently; that is, every node that enters the system and remains active eventually joins. For every node \(p \in S_0\), \(\text{Joined}_p\) 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.
• For every node $p$, if $\text{STORE}_p(v)$ (respectively, $\text{COLLECT}_p$) occurs, then at least one of $\text{LEAVE}_p$, $\text{CRASH}_p$, or $\text{ACK}_p$ (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 $\text{Changes}$ of events concerning the nodes that have entered the system. When an $\text{ENTER}_p$ event occurs, $p$ adds $\text{enter}(p)$ to its $\text{Changes}$ set (Line 11) 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 $\text{enter}(q)$ to its $\text{Changes}$ set (Line 3 or 6). When $p$ receives an enter message from a node $q$, it replies with an enter-echo message containing its $\text{Changes}$ set, its current estimate $L\text{View}$ (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 $\text{join\_counter}$ (Line 10).

The fraction $\gamma$ 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 3). Setting $\gamma$ 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 $\text{join}(q)$ to its $\text{Changes}$ set, sets its is joined flag to true (Line 12), broadcasts a message saying that it has joined (Line 14) and outputs $\text{JOINED}_p$ (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 $\text{join}(q)$ to its $\text{Changes}$ set (Line 16 or 19). When a $\text{LEAVE}_p$ 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 $\text{leave}(q)$ to its $\text{Changes}$ set (Line 23 or 25).

Initially, node $p$’s $\text{Changes}$ set equals $\{\text{enter}(q)| q \in S_0\} \cup \{\text{join}(q)| q \in S_0\}$, if $p \in S_0$, and $\emptyset$ otherwise. Node $p$ also maintains a set of nodes that it believes are present: $\text{Present} = \{q | \text{enter}(q) \in \text{Changes} \land \text{leave}(q) \notin \text{Changes}\}$, 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 is explained below.

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 $\text{Members} = \{q | \text{join}(q) \in \text{Changes} \land \text{leave}(q) \notin \text{Changes}\}$ 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, $\{(p, v, sqno), \ldots\}$, without repetition of node ids. We use the notation $V(p) = v$ if there exists sqno such that $(p, v, sqno) \in V$, and ⊥ 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. That is, 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 $V_1, V_2 \subseteq \text{merge}(V_1, V_2)$.

Each node keeps a local copy of the current view in its $L\text{View}$ variable. In a collect operation, a client thread requests the latest value of servers’ local views using a collect-query message (Line 24). When a
server node \( p \) receives a collect-query message, it responds with its local view \( LView \) through a collect-reply message (Line 5). 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 51). Then the client waits for sufficiently many collect-reply messages before broadcasting the current value of its \( LView \) variable in a store message (Line 56). When server \( p \) receives a store message with a 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 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 merges the value it wishes to store with its local view, and broadcasts the resulting view with a new sequence number. It updates its local variable \( LView \) to reflect the new value by doing a merge (Line 59) and broadcasts a store message (Line 12). 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 50).

The fraction \( \beta \) is used to calculate the number of messages that should be received (stored in local
Algorithm 2  CCC—Client code, for node $p$.

Local Variables:
- $\text{optype}$: string, initially $\bot$ // indicates which type of operation ($\text{collect}$ or $\text{store}$), if any, is pending
- $\text{tag}$: int, initially 0 // counter to identify current operation by $p$ and match response messages received from servers
- $\text{threshold}$: int, initially 0 // number of replies/acks $p$ should receive before finishing the current phase
- $\text{counter}$: int, initially 0 // number of replies/acks received so far for the current phase

Derived Variable:
$\text{Members} = \{q | \text{join}(q) \in \text{Changes} \land \text{leave}(q) \notin \text{Changes}\}$

When $\text{Collect}_p$ occurs:
26: $\text{optype} = \text{collect}; \text{tag}++$
27: $\text{threshold} = \beta \cdot \lvert \text{Members} \rvert$
28: $\text{counter} = 0$
29: broadcast $\langle \text{collect-query}, \text{tag}, p \rangle$

When Receive$_p(\text{collect-reply}, R\text{View}, t, q)$ occurs:
30: if $(t == \text{tag}) \land (q == p)$ then
31: $L\text{View} = \text{merge}(L\text{View}, R\text{View})$
32: $\text{counter}++$
33: if $(\text{counter} \geq \text{threshold})$ then
34: $\text{threshold} = \beta \cdot \lvert \text{Members} \rvert$
35: $\text{counter} = 0$
36: broadcast $\langle \text{store}, L\text{View}, \text{tag}, p \rangle$

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

When Receive$_p(\text{store-ack}, t, q)$ occurs:
43: if $(t == \text{tag}) \land (q == p)$ then
44: $\text{counter}++$
45: if $(\text{counter} \geq \text{threshold})$ then
46: if $(\text{optype} == \text{store})$ then return $\text{ACK}$
47: else return $L\text{View}$

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

When Receive$_p(\text{store}, R\text{View}, \text{tag}, q)$ occurs:
48: $L\text{View} = \text{merge}(L\text{View}, R\text{View})$
49: if is Joined then
50: broadcast $\langle \text{store-ack}, \text{tag}, q \rangle$
51: broadcast $\langle \text{store-echo}, L\text{View} \rangle$

When Receive$_p(\text{collect-query}, \text{tag}, q)$ occurs:
52: if is Joined then
53: broadcast $\langle \text{collect-reply}, L\text{View}, \text{tag}, q \rangle$

When Receive$_p(\text{store-echo}, R\text{View})$ occurs:
54: $L\text{View} = \text{merge}(L\text{View}, R\text{View})$

variable $\text{threshold}$) based on the size of the $\text{Members}$ set, for the operation to terminate. Setting $\beta$ is also a key challenge in the algorithm as setting it too small might not return correct information from $\text{collect}$ or $\text{store}$, whereas setting it too large might not guarantee termination of the $\text{collect}$ and $\text{store}$.

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

- lines 26 through 33, the first part of a $\text{collect}$ operation,
- lines 34 through 40 and 43 through 47, the second part of a $\text{collect}$ operation called the “store-back”, or
- lines 37 through 47, the entirety of a $\text{store}$ operation.

The first kind of phase is called a $\text{collect phase}$ while the second and third kinds are called a $\text{store phase}$.

For any phase $\phi$ executed by node $p$, define $\text{view}(\phi)$ to be the value of $L\text{View}_p^t$, where $t$ is the time at the end of the phase. Since a $\text{store}$ operation consists solely of a store phase, we also apply the notation to an entire $\text{store}$ operation.
5 Proof of CCC Store-Collect Algorithm

Consider any execution of the algorithm with well-formed interactions.

Denote \( Z = [(1 – \alpha)^3 – \Delta \cdot (1 + \alpha)^3] \). The correctness of the algorithm relies on the following constraints:

\[
N_{\text{min}} \geq \frac{1}{Z + \gamma \cdot (1 – \alpha)^3 – (1 + \alpha)^3}
\]

(A)

\[
\gamma \leq \frac{Z}{(1 + \alpha)^3}
\]

(B)

\[
\beta \leq \frac{Z}{(1 + \alpha)^2}
\]

(C)

\[
\beta > \frac{(1 – Z)(1 + \alpha)^5 + (1 + \alpha)^6}{(1 – \alpha)^3(1 + \alpha)^2 + (1 – \alpha)^3}
\]

(D)

These constraints are used in Appendix A to prove the following technical claims.

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

**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 \( t_2 \). (Recall that \( Z = [(1 – \alpha)^3 – \Delta \cdot (1 + \alpha)^3] \).)

As an immediate corollary, since \( |S| \) must be at least \( N_{\text{min}} \), the lower bound on \( N_{\text{min}} \) 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 3.** 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 superscripted with \( t \) to denote the value of that variable at time \( t \); e.g., \( v_t^i \) 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’ \( \text{Changes} \) sets to the set of \( \text{ENTER} \), \( \text{JOINED} \), and \( \text{LEAVE} \) events that have actually occurred in a certain interval. We refer to these as \( \text{membership events} \). Because of the assumed initialization of the nodes in \( S_0 \), we use the convention that the set of membership events occurring in the interval \([0, 0]\) is \( \{\text{enter}(p)|p \in S_0\} \cup \{\text{join}(p)|p \in S_0\} \). The next lemma is proved in Appendix A.

**Lemma 4.** For every node \( p \) and all times \( t \) such that \( p \) is joined and active at \( t \), \( \text{Changes}_t^p \) contains all the membership events for \([0, \max\{0, t – 2D\}]\).

We can now show that every node that remains active sufficiently long will eventually join.

**Theorem 5.** Every node \( p \) that is active for at least \( 2D \) time joins by time \( t_p^e + 2D \).

**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 3 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 3 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 4 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’ ≥ t_p^e ≥ 2D, it follows that t’ - 2D ≥ 0. By Lemma 4, Changes_{q’} contains all the membership events for [0, t’ - 2D] and thus so does Changes_{p’}. As a result, Present_p’ 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’’ ≤ t’ + D, it follows from Lemma 7 part (a) that at most ((1 + α)^3 - 1) · |S| nodes enter during (t’ - 2D, t’’]. Thus join_threshold ≤ α · (1 + α)^3 · |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 8 and considering the interval (t’ - 2D, t’’]: Z · |S| ≥ γ · (1 + α)^3 · |S| ≥ join_threshold, where the first inequality holds by Constraint B.

We now show that a phase terminates as long as the invoking client node is active long enough.

**Theorem 6.** A phase invoked by a client node 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 27 or 34 or 44 of Algorithm 2.

Let S be the set of nodes present at time max{0, t - 2D} = t’. By Lemma 2, the number of those nodes that are still active at time t + D is at least Z · |S|. If t’ = 0, then S = S_0 and all these nodes are joined throughout; otherwise, by Theorem 5 all these nodes are joined by time t.

We now show that |S| ≥ |Present_p^t|/(1 + α)^2. By Lemma 4, Changes_p contains all the membership events for [0, t’], which indicates that all nodes in S are present at t’. Present_p^t is as large as possible if 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 4 part (a) implies that the maximum number of nodes that can enter is (1 + α)^2 · |S|. Thus |Present_p^t| ≤ (1 + α)^2 · |S|.

Thus the number of nodes that are joined by time t and are still active at time t + D, i.e., nodes guaranteed to respond to p, is at least Z · |S| ≥ Z · |Present_p^t|/(1 + α)^2 ≥ β · |Present_p^t| ≥ β · |Members_p^t|, where the penultimate inequality holds by Constraint C and the last inequality is true since enter(q) is added to Changes_p whenever join(q) is added. Since β · |Members_p^t| is the value to which threshold is set at time t, p succeeds in receiving the required number of collect-reply or store-ack messages by time t + 2D and the phase completes.

The next lemma states that the views of a store phase and a subsequent, non-overlapping, collect phase respect the partial order ⪯. There are 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. The proof appears in Appendix A.

**Lemma 7.** For any store phase s and any collect phase c, if s finishes before c starts, then view(s) ⪯ view(c).

**Theorem 8.** 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.

(1.1) Suppose V(p) = ⊥. We show no store operation by p precedes cop. Suppose in contradiction there is such a store operation, consisting of store phase s. By Lemma 7 view(s) ⪯ view(c). Since view(s) contains a tuple for p (with a non-⊥ value), a contradiction is achieved.
In the extreme case when \( \alpha \) it suffices to set both \( \gamma \) store-collect object. must decrease approximately linearly until reaching 0.03; in this case, it suffices to set Finally, Theorem 8 states that regularity is satisfied, implying that by a store phase, Theorem 6, which states that every phase event eventually completes as long as the invoker

**Theorem 5** states that every node that enters and remains active sufficiently long eventually joins. Since

\[ \text{Implementing Distributed Objects Despite Continuous Churn} \]

In this section we show how to implement a variety of other objects using a store-collect object. 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. A max register holds the largest value written into it; WRITEMAX(v) records a value, while READMAX() returns the largest argument of preceding WRITEMAX's (Algorithm 4). An abort flag is a Boolean flag that can only be raised from false to true; CHECK() returns true if an abort() precedes or overlaps it (Algorithm 5). A set holds all values added into it; it provides two operations ADDSET() and READSET() with the obvious properties (Algorithm 6). More details are in Appendix B.

---

**Algorithm 4** Max-Register:

When WRITEMAX\(_p\)(v) occurs:
1: \( \text{Store}_p(v) \)
2: return \( \text{ACK} \)

When READMAX\(_p\) occurs:
3: \( V = \text{Collect}_p() \)
4: return \( \max(V.\text{val}) \)

**Algorithm 5** Abort Flag:

When abort\(_p\) occurs:
1: \( \text{Store}_p(\text{true}) \)
2: return \( \text{ACK} \)

When check\(_p\) occurs:
3: \( F = \text{Collect}_p() \)
4: if \( \exists q \text{ s.t. } F(q) == \text{true} \) then
5: return \( \text{true} \)
6: else return \( \text{false} \)

**Algorithm 6** Set:

When ADDSET\(_p\)(v) occurs:
1: \( \text{LSet} = \text{LSet} \cup \{v\} \)
2: \( \text{Store}_p(\text{LSet}) \)
3: return \( \text{ACK} \)

When READSET\(_p\) occurs:
4: \( S = \text{Collect}_p() \)
5: return \( \cup S.\text{set} \)

---

\(^2\) The behavior of these objects can be formalized through interval linearizability.
6.2 Atomic Snapshots

We further demonstrate the usefulness of the store-collect algorithm, by employing it to implement atomic snapshots [5]. Like other atomic snapshot algorithms [11,15,26], it 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 \((\text{node id}, \text{value})\) pairs, without duplicate node ids.

Formally, an atomic snapshot provides two operations:

- **SCAN()**, has no arguments and returns a snapshot view.
- **UPDATE(v)**, 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, if it does not.

An implementation should be linearizable [21]. Roughly speaking, for every execution \( \alpha \), we should find a sequence of operations, lin(\( \alpha \)), which contains all completed operations in \( \alpha \) and some of the pending operations, such that: (a) lin(\( \alpha \)) is in the sequential specification of an atomic snapshot, and (b) lin(\( \alpha \)) preserves the real-time order of non-overlapping operations in \( \alpha \).

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 \((\text{node id}, \text{value})\) 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 \((p,v)\) in \( V \) with \((p,v.comp)\); when \( v.comp = \bot \), the tuple is omitted. Recall notation that for any kind of view \( V \), \( V(p) \) equals the second component of the pair whose first component is \( p \) (\( \bot \) if there is no such pair).

To execute a SCAN, Algorithm 7 increments the scan sequence number (ssqno) (Line 1) and performs a store on the shared store-collect object with all the other components unchanged, indicated by the notation. Then, the first view is collected (Line 3). In a while loop, the last collected view is saved and a new view is collected (Line 5). If the two most recently collected views are the same (Line 6), the latest collected view is returned (Line 7). 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 8). If this condition holds, the snapshot view of \( q \) is returned (Line 9); 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 10). Next, an embedded scan is performed (Line 11) and the returned view is saved in a local variable sview. Then it sets its val variable to the argument value and increments its update sequence number (Lines 12 and 13). Finally the new value, update sequence number, collected view, and set of scan sequence numbers are stored; only the node’s own scan sequence number is unchanged, indicated by the notation (Line 14).

To show linearizability of Algorithm 7 we consider any execution and specify an ordering of all the completed scans and all the updates whose store on Line 14 takes effect. The ordering takes into consideration
Algorithm 7 Atomic snapshot: code for node p.

Local Variables:

\begin{itemize}
\item \texttt{ssqno}: int, initially 0 \hspace{1em} // counts how many scans \texttt{p} has invoked so far
\item \texttt{scounts}: set of (node id, integer) pairs with no duplicate node ids; initially \emptyset
\item \texttt{val}: an element of \texttt{ValAS}, initially \bot \hspace{1em} // the argument to the most recent update invoked by \texttt{p}
\item \texttt{usqno}: int, initially 0 \hspace{1em} // number of updates \texttt{p} has invoked so far
\item \texttt{sview}: a snapshot view, initially \bot \hspace{1em} // the result of recent embedded scan by \texttt{p}
\item \texttt{V₁, V₂}: store-collect views, both initially \emptyset
\end{itemize}

\begin{algorithmic}[1]
\Function{scan\textsubscript{p}()}{occurs:}
\State \texttt{ssqno}++
\State \texttt{store\textsubscript{p}}((\_, \_, \texttt{ssqno}, \_, \_))
\State \texttt{V₁} = \texttt{collect\textsubscript{p}()} \Comment{\texttt{collect\textsubscript{p}()} is \texttt{CC} on \texttt{V₁} \texttt{view\textsubscript{p}}}
\While{true}
\State \texttt{V₂} = \texttt{V₁} \hspace{1em} \texttt{V₁} = \texttt{collect\textsubscript{p}()}
\If{(\texttt{V₁} == \texttt{V₂})}
\State \texttt{return \texttt{V₁}, val} \hspace{1em} // direct scan
\EndIf
\State \texttt{if} \exists \texttt{p} such that (\texttt{p}, \texttt{ssqno}) \texttt{∈} \texttt{V₁(q).scounts} \texttt{then} \texttt{5: return Ack}
\State \texttt{return \texttt{V₁(q).sview}} \hspace{1em} // borrowed scan
\EndWhile
\EndFunction
\Function{update\textsubscript{p}(v)}{occurs:}
\State \texttt{scount} = \texttt{collect\textsubscript{p}()}.\texttt{ssqno}
\State \texttt{sv} = \texttt{scan\textsubscript{p}()} \hspace{1em} // embedded scan
\State \texttt{val} = \texttt{v}
\State \texttt{usqno}++
\State \texttt{store\textsubscript{p}}((\texttt{val}, \texttt{usqno}, \_, \texttt{sview}, \texttt{scounts}))
\EndFunction
\end{algorithmic}

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 show that the ordering is consistent with the order of non-overlapping operations in the execution and that the ordering satisfies the sequential specification of the atomic snapshot object.

We first show that direct scans are comparable in the \(\leq\) order.

Lemma 9. If a direct scan by node \(p\) returns \(V₁\) and a direct scan by node \(q\) returns \(V₂\), then either \(V₁ \leq V₂\) or \(V₂ \leq V₁\).

Proof. Let \(\texttt{cop}_p^1\) and \(\texttt{cop}_p^2\) be the last two collects of \(p\) (both returning \(V₁\)), and \(\texttt{cop}_q^1\) and \(\texttt{cop}_q^2\) be the last two collects of \(q\) (both returning \(V₂\)). We have that either \(\texttt{cop}_p^1\) completes before \(\texttt{cop}_q^2\) starts or \(\texttt{cop}_q^1\) completes before \(\texttt{cop}_p^2\) starts. In the former case, by the regularity of store-collect, \(V₁ \leq V₂\), while in the latter case, \(V₂ \leq V₁\) (for the same reason).

Consider all direct scans in the order they complete and place them by the comparability order. If a direct scan returning snapshot view \(V₁\) precedes another direct scan returning snapshot view \(V₂\), then the regularity of store-collect ensures \(V₁ \leq V₂\). 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 \(\texttt{sop}_p\) by node \(p\) borrows the snapshot view in \(V₁(q)\), then there is an update \(\texttt{uop}_q\) by \(q\) that writes this view (via a store). We call the embedded scan of \(\texttt{uop}_q\) the scan from which \(\texttt{sop}_p\) borrows.

Lemma 10. If a scan \(\texttt{sop}_p\) by node \(p\) borrows from a scan \(\texttt{sop}_q\) by node \(q\), then \(\texttt{sop}_q\) starts after \(\texttt{sop}_p\) starts and completes before \(\texttt{sop}_p\) completes.

Proof. Let \(\texttt{uop}_q\) be the update in which \(\texttt{sop}_q\) is embedded. Since \(\texttt{sop}_p\) borrows the snapshot view of \(\texttt{sop}_q\), its \(\texttt{ssqno}\) appears in \texttt{scounts} of \(q\)'s value in the view collected in Line 5. The properties of store-collect imply that the collect of \(\texttt{uop}_q\) (Line 10) does not complete before the store of \(p\) (Line 2) starts. Hence, \(\texttt{sop}_q\) (called in Line 11) starts after \(\texttt{sop}_p\) starts. Furthermore, since the collect of \(p\) returns the snapshot view stored after \(\texttt{sop}_q\) completes (Line 13), it follow that \(\texttt{sop}_q\) completes before \(\texttt{sop}_p\) completes.

For every borrowed scan \(\texttt{sop}_1\), there exists a chain of scans \(\texttt{sop}_2, \texttt{sop}_3, \ldots, \texttt{sop}_k\) such that \(\texttt{sop}_i\) borrows from \(\texttt{sop}_{i+1}, 1 \leq i < k\), and \(\texttt{sop}_k\) is a direct scan, which we refer to as the direct scan from which \(\texttt{sop}_1\) 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 10 inductively, sop \_k starts after sop \_l starts and completes before sop \_m 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 14) 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 an atomic snapshot object.

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 include 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 shows that the real-time order of non-overlapping updates is preserved.

**Lemma 11.** 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 \( V = V(w) \). If sop' = sop, then \( V = V(w) \) since sop is a direct scan. Otherwise, \( V = V(w) \) because \( V(w) \) is returned to the enclosing scan, assigned to sview, and then stored (cf. Lines 11 and 14). 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 14 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 \( V = V(w) \), 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 11, 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. We say that an update uop \_p of node p interferes with the double collect, and with sop \_q, if V \_1(p).usqno \( \neq \) V \_2(p).usqno. It is immediate to see that in this case, 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 12.** Algorithm 7 is a linearizable implementation of an atomic snapshot object.
Algorithm 8 Generalized lattice agreement: code for node $p$.

When $\text{Propose}_p(v)$ occurs:
1: $\text{val} = \text{val} \sqcup v$ \hfill // track previous inputs of $p$
2: $\text{Update}_p(\text{val})$
3: $\text{sview} = \text{Scan}_p()$
4: return $\sqcup \text{sview}.\text{val}$

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 $\text{Propose}$ operation with a lattice input value, and gets back a lattice output value. The input to $p$’s $i$th $\text{Propose}$ is denoted $v^p_i$ and the response is $w^p_i$. The following conditions are required:

Validity Every response value $w^p_i$ is the join of some values proposed before this response, including $v^p_i$, and all values returned to any node before the invocation of $\text{Propose}$.

Consistency Any two values $w^p_i$ and $w^q_j$ are comparable.

This definition follows [23], and is a direct extension of one-shot lattice agreement [9]. The version of generalized lattice agreement studied in [16] 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 ($\text{val}$). A $\text{Propose}$ operation is simply an $\text{Update}$ of a lattice value which is the join of all the node’s previous inputs, followed by a $\text{Scan}$ returning the analogous values for all nodes, whose join is the output of $\text{Propose}$.

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 $\text{Propose}$. Since $\text{Propose}$ includes one $\text{Update}$ and one $\text{Scan}$, it terminates as long as 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 described some simple implementations of non-linearizable objects (max register, abort flag, and set) using store-collect. We also presented an algorithm for an atomic snapshot object and 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. 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 the same counterexample as that given in [7] for their register algorithm applies to our algorithm. This behavior is in contrast to the algorithms in [20, 3], 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.

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 $\text{Changes}$ sets.

More generally, we would like to understand the relationships between modeling churn via explicit versus implicit reconfiguration.
References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[2] M. K. Aguilera. A pleasant stroll through the land of infinitely many creatures. *SIGACT News*, 35(2):36–59, 2004.

[3] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7, 2011.

[4] J. Aspnes. Notes on theory of distributed systems. http://www.cs.yale.edu/homes/aspnes/classes/465/notes.pdf. Accessed: 2020-02-14.

[5] J. Aspnes, H. Attiya, and K. Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1):1–24, 2012.

[6] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, Jan. 1995.

[7] H. Attiya, H. C. Chung, F. Ellen, S. Kumar, and J. L. Welch. Emulating a shared register in an asynchronous system that never stops changing. *IEEE Transactions on Parallel and Distributed Systems*, 30(3):544–559, 2018.

[8] H. Attiya, A. Fouren, and E. Gafni. An adaptive collect algorithm with applications. *Dist. Comp.*, 15(2):87–96, Apr. 2002.

[9] H. Attiya, M. Herlihy, and O. Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, 1995.

[10] R. Baldoni, S. Bonomi, A.-M. Kermarrec, and M. Raynal. Implementing a register in a dynamic distributed system. In *IEEE International Conference on Distributed Computing Systems*, pages 639–647, 2009.

[11] R. Baldoni, S. Bonomi, and M. Raynal. Implementing a regular register in an eventually synchronous distributed system prone to continuous churn. *IEEE Transactions on Parallel and Distributed Systems*, 23(1):102–109, 2012.

[12] R. Baldoni, S. Bonomi, and M. Raynal. Implementing set objects in dynamic distributed systems. *Journal of Computer and System Sciences*, 82(5):654–689, 2016.

[13] A. Castañeda, S. Rajsbaum, and M. Raynal. Unifying concurrent objects and distributed tasks: Interval-linearizability. *J. ACM*, 65(6):1–42, 2018.

[14] S. Chordia, S. Rajamani, K. Rajan, G. Ramalingam, and K. Vaswani. Asynchronous resilient linearizability. In *DISC*, pages 164–178, 2013.

[15] C. Delporte-Gallet, H. Fauconnier, S. Rajsbaum, and M. Raynal. Implementing snapshot objects on top of crash-prone asynchronous message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(9):2033–2045, 2018.

[16] J. M. Faleiro, S. Rajamani, K. Rajan, G. Ramalingam, and K. Vaswani. Generalized lattice agreement. In *PODC*, pages 125–134, 2012.

[17] E. Gafni and E. Koutsoupias. On uniform protocols. http://www.cs.ucla.edu/~eli/eli.html/main.ps.

[18] E. Gafni and D. Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *DISC*, volume 9363 of *Lecture Notes in Computer Science*, pages 140–153. Springer, 2015.
A Additional Proofs for Algorithm CCC

The next lemma bounds how much the system size can grow during $i$ intervals of length $D$ each, as a function of the original system size.

**Lemma 1** (Repeated). 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

$$
\begin{align*}
&\leq \alpha \cdot N(t) + ((1 + \alpha)^i - 1) \cdot N(t + D) \\
&\leq \alpha \cdot N(t) + ((1 + \alpha)^i - 1) \cdot (1 + \alpha) \cdot N(t) \quad \text{by (ii)} \\
&= \alpha \cdot N(t) + (1 + \alpha)^{i+1} \cdot N(t) - (1 + \alpha) \cdot N(t) \\
&= ((1 + \alpha)^{i+1} - 1) \cdot N(t).
\end{align*}
$$
To show (b) for $i + 1$:

\[
N(t + (i + 1) \cdot D) \leq (1 + \alpha)^i \cdot N(t + D) \quad \text{by the inductive hypothesis for (b)}
\]

\[
\leq (1 + \alpha)^i \cdot (1 + \alpha) \cdot N(t) \quad \text{by (ii)}
\]

\[
= (1 + \alpha)^{i+1} \cdot N(t).
\]

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 13.** For all $\alpha$, $0 < \alpha < 0.06$, 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 $\ell$ be the exact number of nodes that leave in $(t, t + D]$. The number of nodes that leave in the entire interval is:

\[
\leq \ell + (1 - (1 - \alpha)^i) \cdot N(t + D) \quad \text{by the inductive hypothesis}
\]

\[
\leq \ell + (1 - (1 - \alpha)^i) \cdot [(1 + \alpha) \cdot N(t) - 2\ell]
\]

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

\[
\leq (1 - (1 - \alpha)^i) \cdot (1 + \alpha) \cdot N(t) + (2(1 - \alpha)^i - 1)\ell
\]

\[
\leq (1 - (1 - \alpha)^i) \cdot (1 + \alpha) \cdot N(t) + (2(1 - \alpha)^i - 1) \cdot \alpha \cdot N(t)
\]

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 $\alpha$ and $i$ in the premise of the lemma. This expression equals $(1 - (1 - \alpha)^{i+1}) \cdot N(t)$.

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 2 (repeated).** 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 $Z = \left[(1 - \alpha)^3 - \Delta \cdot (1 + \alpha)^3\right]$.)

**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 13, 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 $\Delta$ 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

\[
|S| - (1 - (1 - \alpha)^3) \cdot |S| - \Delta \cdot (1 + \alpha)^3 \cdot |S| = \left[(1 - \alpha)^3 - \Delta \cdot (1 + \alpha)^3\right] \cdot |S|,
\]

\[
\square
\]
The next lemmas describe how a node’s Changes set relates to prior membership events. Lemma 15 states that a node that has been present in the system sufficiently long (at least $2D$ time), has all the information up until $D$ time in the past. Lemma 4 states that a joined node, no matter how recently it entered the system, has all the information up until 2$D$ time in the past. The later parts of the correctness proof only use Lemma 4 but its proof relies on Lemma 15. The proof of Lemma 4 relies on Lemma 16, 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 15.

The proofs of Lemmas 15 and 4 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.

**Observation 14.** For every node $p$ and all times $t \geq t_p^e + D$, if $p$ is active at time $t$, then $\text{Changes}_p^q$ contains all the membership events for $[t_p^e, t - D]$.

**Lemma 15.** For every node $p$ and all times $t \geq t_p^e + 2D$ such that $p$ is active at $t$, $\text{Changes}_p^q$ contains all the 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 $p$.

**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 14 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 $t \geq t_p^e + 2D$. By Corollary 5, 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 $\text{Changes}_p^q$ contains all the 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 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 membership events for $[0, t' - D]$.

2. Suppose some node $r$ enters, joins, or leaves in $[t' - D, t_p^e]$. 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 14 gives the result for $[t_p^e, t - D]$.

**Lemma 16.** 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 $t' \leq t$. If $\text{Changes}_p^q$ contains all the 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 the desired $q'$ is an element of $S$.

We now calculate a lower 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 at time $t''$ immediately after $p$ receives the enter-echo response from $q$ (cf. Line 29 of Algorithm 1). By the premise of the lemma, $\text{Changes}_p^q$ contains all the membership events for $[0, \max\{0, t' - 2D\}]$. Thus when $p$ receives the enter-echo response from $q$ at time
$t'' \leq t' + D$, its Present variable contains, at a minimum, all the nodes in $S$ minus the maximum number of them that could have left during $[\max\{0, t' - 2D\}, t' + D]$. By Lemma 13 the maximum number of nodes that can leave in this interval is $(1 - (1 - \alpha)^3) \cdot |S|$. So we have that join_threshold $\geq \gamma \cdot (1 - \alpha)^3 \cdot |S|$.

We next 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 minimum value of join_threshold and subtract the maximum number of enter-echo responses that could come from nodes not in $S$, i.e., nodes that enter after $\max\{0, t' - 2D\}$ and up to $t''$, which is $((1 + \alpha)^3 - 1) \cdot |S|$ by Lemma 11 part (a). We then subtract the maximum number of nodes in $S$ that do not survive until $t''$, which by Lemma 12 is at most $(1 - Z) \cdot |S|$. The result is 

$$\gamma \cdot (1 - \alpha)^3 \cdot |S| - ((1 + \alpha)^3 - 1) \cdot |S| - (1 - Z) \cdot |S| = [Z + \gamma \cdot (1 - \alpha)^3 - (1 + \alpha)^3] \cdot |S|.$$ 

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 $[\max\{0, t' - 2D\}, t'']$.

**Lemma 4 (repeated).** For every node $p$ and all times $t$ such that $p$ is joined and active at $t$, Changes$^t_p$ contains all the membership events for $[0, \max\{0, t - 2D\}]$.

**Proof.** The proof is by induction on the order in which nodes join. In particular, we show 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 $t \leq 2D$, we just need to show that Changes$^t_p$ contains all the membership events for $[0, 0]$, which is true by the assumed initialization of nodes in $S_0$. When $t > 2D$, Observation 14 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.

When $t - t^e_p \geq 2D$, Lemma 14 gives the result. So we suppose $t - t^e_p < 2D$. If $t \leq 2D$, then all that’s required is for Changes$^t_p$ to include all the 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 membership events for $[0, 0]$ in its Changes set when it sent the enter-echo. Thus $p$ receives all the membership events for $[0, 0]$ before it joins. For rest of the proof, assume $t > 2D$.

We will show that Changes$^t_p$ contains all the membership events for $[0, t - 2D]$ in two steps: one for $[\max\{0, t' - 2D\}]$ 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$^{t'}_q$ contains all the membership events for $[0, \max\{0, t' - 2D\}]$, and thus so does Changes$^t_p$.

2. By Lemma 16 $p$ 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 $[\max\{0, t' - 2D\}, t - 2D]$. 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 $t - t' \leq t - t^e_p < 2D$, 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 $t$.

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

**Observation 17.** For any store phase $s$ that starts at time $t_s$ and calls broadcast (Line 32 of Algorithm 3), and any node $p$ that is active throughout $[t_s, t]$ where $t \geq t_s + D$, view$(s) \leq LV view^t_p$. 19
The next lemma is the analog of Lemma 15: a node that has been active for at least $2D$ time “knows about” store phases that started up to $D$ in the past.

**Lemma 18.** If node $p$ is active at any time $t \geq t^e_p + 2D$, then $\text{view}(s) \preceq \text{LView}^p_t$ for every store phase $s$ that starts at or before $t - D$ and calls broadcast (Line 42 of Algorithm 3).

**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 17.

**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 $t \geq t^e_p + 2D$ when $p$ is active. Let $s$ be a store phase that starts at $t_s \leq t - D$. If $t_s \geq t^e_p$, the claim holds by Observation 17.

Suppose $t_s < t^e_p$. By Corollary 3, there is at least one node $q$ that is active throughout $[\max\{0, t^e_p - 2D\}, t^e_p + D]$. Since $t \geq t^e_p + 2D$, $p$ receives $q$’s enter-echo response by time $t$. Since views and sequence numbers are included in enter-echo messages, $\text{LView}^q_t \preceq \text{LView}^p_t$, where $t'$ is the time when $q$ receives $p$’s enter message.

Case 1: $t_s < \max\{0, t^e_p - D\}$. We show that the inductive hypothesis applies for node $q$, time $t'$, and store phase $s$. Thus $\text{view}(s) \preceq \text{LView}^q_{t'}$, and by transitivity, $\text{view}(s) \preceq \text{LView}^p_t$. 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: $t_s \geq \max\{0, t^e_p - D\}$. The store message sent during $s$ is guaranteed to arrive at $q$ either before $t^e_p$ or at or after $t^e_p$. In the former case, $q$’s enter-echo response, which $p$ receives by $t^e_p + 2D \leq t$, contains a view $V$ such that $\text{view}(s) \preceq V$. In the latter case, $q$’s store-echo message contains a view $V$ with $\text{view}(s) \preceq V$ and $p$ receives this message by $t_s + 2D < t^e_p + 2D \leq t$. In both situations, $\text{view}(s) \preceq \text{LView}^p_t$.

The next lemma is the analog of Lemma 4: a node that is joined “knows about” store phases that started up to $2D$ in the past.

**Lemma 19.** If node $p$ is joined and active at any time $t$, then $\text{view}(s) \preceq \text{LView}^p_{t'}$ for every store phase $s$ that starts at or before $t - 2D$ and calls broadcast (Line 42 of Algorithm 3).

**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 17.

**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 $t_s \leq t - 2D$. If $t \geq t^e_p + 2D$, the claim follows from Lemma 18.

Suppose $t < t^e_p + 2D$. For every store phase that starts at or after $t^e_p$, the claim follows from Observation 17.

Consider any store phase that starts at some time $t_s < t^e_p$. 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 $\text{view}(s) \preceq \text{LView}^q_{t'}$, and by transitivity, $\text{view}(s) \preceq \text{LView}^p_{t'}$. 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 4 implies that $\text{Changes}^q_{t'}$ contains all the membership events for $[0, \max\{0, t' - 2D\}]$. Thus Lemma 10 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 $t^e_p$ or at or after $t^e_p$. In the former case, the enter-echo message from $q'$ is sent to $p$ contains a view $V$ with $\text{view}(s) \preceq V$; this message is received by $p$ before it joins. In the latter case, the store-echo message from $q'$ is sent to $p$ contains a view $V$ with $\text{view}(s) \preceq V$; this message is received by $p$ by $t_s + 2D < t^e_p + 2D \leq t$. In both situations, $\text{view}(s) \preceq \text{LView}^p_{t'}$. 

The next lemma gives a lower bound on the size of a node’s $\text{Members}$ set as a function of the size of the system $3D$ time in the past.
Lemma 20. For every node \( p \) and every time \( t \) at which \( p \) is joined and active, \(|Members^t_p| \geq (1 - \alpha)^3 \cdot N(\max\{0, t - 3D\})\).

Proof. Let \( S \) be the set of nodes that are present at time \( \max\{0, t - 3D\} \), so \(|S| = N(\max\{0, t - 3D\})\). By Lemma 4, \( p \)'s Changes set contains, at time \( t \), \( \text{enter}(q) \) for every \( q \in S \) and \( \text{join}(q) \) for every \( q \in S \) that is joined by time \( t \). To minimize \(|Members^t_p|\), we consider the case when no nodes join and the maximum number of nodes leave during \([\max\{0, t - 3D\}, t]\), and all the leave messages reach \( p \) by \( t \). By Lemma 18, the resulting size of \( p \)'s Present set is at least \((1 - \alpha)^3 \cdot |S|\). If \( t - 3D > 0 \), then by Theorem 5 all the remaining nodes in \( S \) are joined by \( t - D \) and have notified \( p \) by \( t \); if \( t - 3D \leq 0 \), then \( \max\{0, t - 3D\} = 0 \) and \( S = S_0 \), all of which are joined from the beginning. Thus at time \( t \), \( p \)'s Members set is equal to its Present set, implying \(|Members^t_p| \geq (1 - \alpha)^3 \cdot |S|\).

**Lemma 21** (repeated). For any store phase \( s \) and any collect phase \( c \), if \( s \) finishes before \( c \) starts, then \( \text{view}(s) \preceq \text{view}(c) \).

Proof. Let \( p_1 \) be the client node that executes \( s \) and \( t_\alpha \) the start time of \( s \). Let \( p_2 \) be the client node that invokes \( c \) and \( t_\alpha \) 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 14 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 22 of Algorithm 2).

Case I: \( t_\alpha - t_\alpha \geq 2D \). 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 \( t \geq t_\alpha \). By the assumption of the case, \( t - t_\alpha \geq 2D \). Thus by Lemma 19, \( \text{view}(s) \preceq \text{LView}^t_q \). Since \( p_2 \) receives an enter-echo message from \( q \) containing \( \text{LView}^t_q \) before completing \( c \), it follows that \( \text{view}(s) \preceq \text{view}(c) \).

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

- Let \( J \) be the set of all nodes that are joined and active at some time in \([t_\alpha, t_\alpha + D] \). 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 \( t_\alpha \). 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 \( t^* = \max\{0, t_\alpha - 2D\} \).

First we calculate an upper bound on \(|J| \). Since it takes at most \( 2D \) time to join after entering by Theorem 5, every node in \( J \) is either present at \( t^* \) or enters during \([t^*, t_\alpha + D] \). By Lemma 1(b),

\[ |J| \leq (1 + \alpha)^3 \cdot N(t^*) \]

Next we calculate a lower bound on \(|Q_c| \).

\[ |Q_c| = \beta \cdot |Members^t_{p_2}| \]

\[ \geq \beta \cdot (1 - \alpha)^3 \cdot N(\max\{0, t_\alpha - 3D\}) \] 

\[ \geq \beta \cdot (1 - \alpha)^3 \cdot (1 + \alpha)^{-1} \cdot N(t^*) \]

We now calculate a lower bound on \(|K| \). By Lemma 2 at most \((1 - Z) \cdot N(t_\alpha) \) nodes crash or fail during \([t_\alpha, t_\alpha + D] \), since the length of the interval is at most \( 3D \). In the worst case, all the nodes that crash or fail
are in \( Q_s \).

\[
|K| \geq |Q_s| - (1 - Z) \cdot N(t_s)
\]

\[
= \beta \cdot |Members_{ps_1}| - (1 - Z) \cdot N(t_s)
\]

by the code

\[
\geq \beta \cdot (1 - \alpha)^3 \cdot N(\max\{0, t_s - 3D\}) - (1 - Z) \cdot N(t_s)
\]

by Lemma 20

\[
\geq \beta \cdot (1 - \alpha)^3 \cdot (1 + \alpha)^{-3} \cdot N(t) - (1 - Z) \cdot N(t_s)
\]

by Lemma 20

\[
\geq \beta \cdot (1 - \alpha)^3 \cdot (1 + \alpha)^{-3} \cdot N(t^*) - (1 - Z) \cdot (1 + \alpha)^2 \cdot N(t^*)
\]

by Lemma 20(b) since \( 0 < t_s - t < 2D \) and \( 1 - Z > 0 \)

\[
= \left[ \beta \cdot (1 - \alpha)^3 \cdot (1 + \alpha)^{-3} - (1 - Z) \cdot (1 + \alpha)^2 \right] \cdot N(t^*)
\]

Finally, we show \( |Q_c| + |K| > |J| \).

\[
|Q_c| + |K| \geq \left[ \beta \cdot (1 - \alpha)^3 \cdot (1 + \alpha)^{-1} + \beta \cdot (1 - \alpha)^3 \cdot (1 + \alpha)^{-3} - (1 - Z) \cdot (1 + \alpha)^2 \right] \cdot N(t^*)
\]

\[
> (1 + \alpha)^3 \cdot N(t^*) \text{ by Constraint 10}
\]

\[
\geq |J|.
\]

\[\square\]

**B  More Details for the Non-Linearizable Objects**

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

- **WRITE_MAX(v)** takes a value \( v \) as an argument and returns \( \text{ACK} \).
- **READ_MAX()** has no arguments and returns a value.

Its sequential specification consists of all sequences of WRITE_MAX and READ_MAX operations in which each READ_MAX returns the largest argument of all preceding WRITE_MAX’s, if any exists, and 0, if there is none.

Algorithm 4 uses a single store-collect object, holding a single value \( \text{val} \) for each node, and a local variable \( V \) for each node, holding a view. WRITE_MAX stores the new value (Line 1) and returns \( \text{ACK} \) (Line 2). READ_MAX collects a view (Line 3) and returns the maximum value stored in it (Line 4).

The main correctness properties achieved are as follows. If any value is written before the end of the collect by a READ_MAX, then by the regularity property of store-collect, the READ_MAX returns the maximum value of all the values written before it. If the start of the store by a WRITE_MAX follows a READ_MAX, then the READ_MAX does not consider the store value.

**Abort flag**: a Boolean flag that can only be raised from \( \text{false} \) to \( \text{true} \) [23]; provides two operations:

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

Its sequential specification consists of all sequences of ABORT and CHECK operations in which each CHECK returns \( \text{true} \) if an ABORT precedes it, and otherwise returns \( \text{false} \).

Algorithm 5 follows [23]. It uses a single store-collect object, holding a single flag \( \text{flag} \) for each node, and a local variable \( F \) for each node, holding a view. ABORT stores true (Line 1) and returns \( \text{ACK} \) (Line 2). CHECK
collects the flags (Line 3). If any of the flags is true then CHECK returns true (Line 4). Otherwise, returns false (Line 5).

The main correctness property achieved is the following. If an ABORT completes before a CHECK starts, then in particular, its store raises the flag before the end of the collect by CHECK. Hence, the CHECK returns true by the regularity property for store-collect. Otherwise, CHECK returns false.

Set: contains all values added into it [23]; 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.

Its sequential specification consists of all sequences of ADDSET and READSET operations in which READSET returns the set of all values added by preceding ADDSET operations.

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 1), stores it (Line 2), and returns ACK (Line 3). READSET collects the set of values (Line 4) and returns the union of all the sets of values (Line 5).

The main correctness property achieved is the following. If an ADDSET stores a value v before the end of a collect by a READSET, then the READSET returns a set of values that includes v by the regularity property for store-collect. Otherwise, the READSET’s return value does not include v.