Fault-tolerant Snapshot Objects in Message Passing Systems

Vijay K. Garg[1], Saptaparni Kumar[2], Lewis Tseng[2], Xiong Zheng[3]

[1]University of Texas at Austin, USA
[2]Boston College, USA
[3]Google, Inc., USA
garg@ece.utexas.edu, {kumargh; lewis.tseng}@bc.edu, xiongzheng@google.com

***Abstract*—The atomic snapshot object (ASO) can be seen as a generalization of the atomic read/write register. ASO divides the object into n segments such that each node can *update* its own segment, and instantaneously *scan* all segments of the object. ASO is a powerful data structure that has many important applications, such as update-query state machines, linearizable conflict-free replicated data types, generalized lattice agreement, and cryptocurrency as in the form of an asset transfer object.**

This paper studies ASO in asynchronous message passing systems and proposes a framework for implementing efficient fault-tolerant snapshot objects. Denote by D the maximum message delay and k the actual number of failures in an execution. Our framework derives two ASO algorithms:

- **A crash-tolerant ASO algorithm that achieves $O(\sqrt{k} \cdot D)$ time complexity for both update and scan operations, and achieves amortized constant time operations if there are $\Omega(\sqrt{k})$ operations.**
- **A Byzantine ASO algorithm that achieves $O(k \cdot D)$ time complexity for both update and scan operations, and achieves amortized constant time operations if there is no Byzantine node in a given execution.**

The framework can also be adapted to implement sequentially consistent snapshot objects (SSO) that complete scan operations locally *without* any communication, and have the same time complexity for update operations as in our ASO algorithms.

***Index Terms*—Atomic Snapshot Object, Sequential Snapshot Object, Crash Failure, Asynchrony, Byzantine Failure**

I. Introduction

The atomic snapshot object (ASO) is a concurrent object [2], [3], [6], which is partitioned into n segments, one for each node. Node i can either update the i-th segment (single-writer model), or instantaneously scan *all segments* of the object. ASO is a powerful abstraction that has a wide spectrum of applications, such as update-query state machines [23], linearizable conflict-free replicated data types (CRDT) [37], and generalized lattice agreement [21], [23]. Prior works also use ASO for solving approximate agreement [13], randomized consensus [4], [5], and implementing wait-free data structures in shared memory [5], [27]. Furthermore, ASO can be used for creating self-stabilizing memory, and detecting stable properties to debug distributed programs (other example applications can be found in [15], [36], [38]). Recently, Guerraoui et al. [26] use ASO to implement a form of cryptocurrency (or an asset transfer object). In essence, ASO simplifies the design and verification of many important distributed algorithms and concurrent data structures.

In message-passing systems, there are many algorithms for emulating atomic read/write registers in the presence of crash faults, e.g., [7], [8], [15], [32]. A simple way to implement an ASO is to first build n SWMR (single-writer/multi-reader) atomic registers, and then apply an ASO implementation designed for the shared memory model, e.g., [2], [11], [14], [30]. As pointed out by Delporte et al. [19], this simple stacking-based approach has a few drawbacks: (i) it introduces unnecessary design and engineering complication due to the stacking; and (ii) the complexity metric is often obscure. The prior shared-memory ASO algorithms (e.g., [2], [3], [6], [17]) are optimized for step complexity (i.e., the number of steps taken by each process) and number of shared memory objects used, whereas in message-passing systems, message and time complexities (measured in the maximum message delay D) are usually more important. It is not clear how shared-memory ASO algorithms behave due to the complication introduced by stacking, especially for those algorithms that use more powerful memory operations like Test&Set (e.g., [11]).

In 2018, Damien Imbs et al. [29] propose a new communication abstraction called Set-Constrained Delivery Broadcast (SCD-Broadcast), which can be used to construct an ASO. Only the best-case (or normal-case when there is no failure) complexity is presented in the paper, which takes $2D$ time for a scan operation and $4D$ time for a update operation. The lower bound result from a recent PODC paper [28] implies that $2D$ and $4D$ are *not* the worst-case complexity for scan and update, respectively. We believe that the worst-case complexity of the ASO based on the SCD-Broadcast is $O(k \cdot D)$, where k is the number of actual failures in an execution.

In 2018, Delporte et al. [19] present the first algorithm for directly implementing an ASO in asynchronous message-passing systems. In their implementation, each UPDATE operation takes constant time, and each SCAN operation takes linear time. In 2020, Attiya et al. [12] propose a store-collect object in dynamic networks with continuous churn. The object

The authors are listed alphabetically. Xiong completed the work while at University of Texas at Austin.

can be used to build an ASO, which takes linear time for both UPDATE and SCAN operations. Another approach is to apply lattice agreement algorithms in message passing systems [41], [42] to implement ASO [11], which results into logarithmic time. All the prior mentioned works tolerate *only* crash faults.

A. Main Contributions

Following the observations in [19], we investigate approaches to directly implement snapshot objects in crash-prone message-passing systems. Consider an asynchronous message-passing systems with n nodes, with up to $f < \frac{n}{2}$ crash faults (or $f < \frac{n}{3}$ Byzantine faults). We propose a framework of implementing ASO and sequentially consistent snapshot objects (SSO) [35]:

- A crash-tolerant ASO algorithm that takes $O(\sqrt{k} \cdot D)$ time for both UPDATE and SCAN operations, where $k \leq f$ is the number of actual failures in an execution. If there are $\Omega(\sqrt{k})$ operations, each operation takes amortized constant time.
- A Byzantine ASO algorithm that takes $O(k \cdot D)$ time for both UPDATE and SCAN operations, where $k \leq f$ is the number of actual failures in an execution. If there are $\Omega(\sqrt{k})$ operations and no node is Byzantine, each operation takes amortized constant time.
- Our SSO algorithms have the same time for UPDATE, and enjoys fast SCAN operations, i.e., nodes complete SCAN operations locally without any communication.
- When there is no failure in an execution, all the algorithms have constant time complexity *unconditionally*.

Our main contribution is the crash-tolerant ASO EQ-ASO, named after our novel technique – equivalence quorum – which allows nodes to check a local predicate to determine whether it is safe to complete a SCAN operation. Due to lack of space, we only present EQ-ASO with key proofs. Proof details, Byzantine ASO, and crash-tolerant and Byzantine SSO algorithms are presented in our technical report [25].

Time Complexity: Following [10], [29], [31], [34], we measure time complexity and delays in time units of a global clock, which is *only* visible to an outside viewer. No node has access to the global clock. We further assume that a message between any pair of nodes takes at most D units of time to be delivered (i.e., maximum message delay), and there is no lower bound on the message delay. D is only used for analyzing the algorithms. Nodes do not have the knowledge of D, nor can they measure it. Hence, our algorithms work correctly in the typical assumption of asynchronous systems [15], [32].[1]

One common measure of performance in asynchronous message-passing algorithms is the round complexity [15], [32], which counts the number of rounds to complete an operation. Our algorithms are *not* round-based like the ones in [7], [19]; hence, we express the time complexity of our algorithms in terms of D, which has also been applied in prior works [10], [29], [31], [34]. Observe that round complexity implies time complexity, because potentially each round takes $O(D)$ time to complete. This is also how we obtain the time complexity of prior algorithms in Table I. The comparison between previous closely related works that directly implement crash-tolerant ASO in message-passing systems [12], [19], [29] and our algorithms are presented in Table I. We are *not* aware of any algorithms that directly implement Byzantine ASO or SSO. We discuss other related work in Section IV.

[1]Reference [10] proves that consensus is also impossible in this model even with the assumption of a global clock and the time bound D.

B. Other Contributions

We identify necessary and sufficient conditions for correctly implementing ASO and SSO. Our conditions apply to both message-passing systems and shared memory systems, since we abstract away from the algorithm construction when specifying the requirements on the ordering of operations in a given execution history. Furthermore, we abstract a key component of our snapshot object framework and adapt it to solve a (one-shot) lattice agreement problem [11]. The resulting algorithm achieves $O(\sqrt{k} \cdot D)$ time complexity. It is the first early-stopping lattice agreement algorithm that we are aware of.

II. PRELIMINARIES

A. System Model

We consider an asynchronous message-passing system composed of n nodes with unique identifiers from $\{1, 2, ..., n\}$. Each node has exactly one server thread and one client thread. Client threads invoke SCAN or UPDATE operations, which can have at most one SCAN or UPDATE operation at any time, i.e., a sequential node. Server threads handle incoming messages (i.e., event-driven message handlers). At most f nodes may fail by crashing in the system. We use k, where $k \leq f$, to denote the *actual number of failures* in a given execution.

Each pair of nodes can communicate with each other by sending messages along point-to-point channels. To simplify our algorithm presentation, we further assume two nice properties of the channels. Channels are reliable and FIFO (First-In, First-Out). "Reliable" means that a message m sent by node i to node j is eventually delivered to node j if j has not already crashed. Intuitively, once the command "send m to j" is completed at node i, the network layer is responsible for delivering m to j. The delivery will occur even if node i crashes or becomes Byzantine after completing the "send" command. Such a channel can be implemented by a reliable broadcast primitive in asynchronous systems [18]. FIFO means that if message m_1 is sent before message m_2 by node i to node j, then m_1 is delivered before m_2 at node j. For our time analysis, we assume that any message is delivered within D units of time. However, the correctness proofs do not rely on the assumption of D.

B. The Snapshot Object

The snapshot object is made up of n segments (one per node), and supports two operations: UPDATE and SCAN. Node i invokes UPDATE(v) to write value v into the i-th segment of the snapshot object. We adopt the single-writer semantic, i.e., only node i can write to the i-th segment. The SCAN operation

TABLE I
CLOSELY RELATED WORK AND OUR RESULTS ON CRASH-TOLERANT SNAPSHOT OBJECTS

Reference	Time: UPDATE		Time: SCAN	
	Worst	Amortized	Worst	Amortized
[19]	$O(D)$	-	$O(n \cdot D)$	-
[12]	$O(n \cdot D)$	-	$O(n \cdot D)$	-
[29]	$O(k \cdot D)$	$O(D)$	$O(k \cdot D)$	$O(D)$
[41], [42] + [11]	$O(\log n \cdot D)$	-	$O(\log n \cdot D)$	-
EQ-ASO [this paper]	$O(\sqrt{k} \cdot D)$	$O(D)$	$O(\sqrt{k} \cdot D)$	$O(D)$
SSO-Fast-Scan [this paper]	$O(\sqrt{k} \cdot D)$	$O(D)$	$O(1)$	$O(1)$

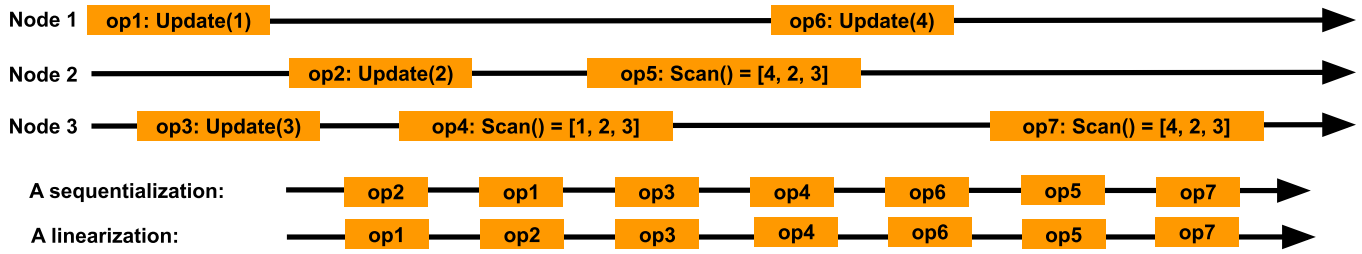

Fig. 1. (Top) An example execution history H, which is represented as a partially ordered set. (Middle) A sequentialization of H. (Bottom) A linearization of H. The left (right) edge of the box represents the invocation (response) event. The value in the parentheses of UPDATE is the value that the node intends to write to its own segment, whereas the segments returned by a SCAN is represented in the form of a vector – each entry of the vector represents the value of the corresponding segment. The only difference between the sequentialization and linearization in this example is the order of $op1$ and $op2$. In linearization, the real-time constraint needs to be enforced; hence, $op1$ has to be placed before $op2$.

allows a node to obtain an instantaneous view of the snapshot object. More concretely, the SCAN returns a vector $Snap$, where $Snap[i]$ denotes the value of the i-th segment of the object. Before formally introducing the consistency property, we present useful notions from the literature [16] to facilitate the discussion. We will use Figure 1 as a working example when presenting the notations.

A *history* is an execution of the snapshot object, which can be represented using a partially ordered set $(H, <_H)$. Here, H is the set of invocation (inv) and response ($resp$) events of the UPDATE and SCAN operations, and $<_H$ is an irreflexive transitive relation that captures the real-time "occur-before" relation of events in H. In Figure 1, the left (right) edge of the box represents the invocation (response) event. Formally, for any two events e and f, we say $e <_H f$ if e occurs before f in the execution. For two operations op_1 and op_2, we say $op_1 \rightarrow op_2$ if $resp(op_1) <_H inv(op_2)$. For example, $op1 \rightarrow op2$ in Figure 1.

A history is *sequential* if the first event is an invocation event, and every invocation event (except possibly the last) is immediately followed by the matching response event. Let $H|i$ denote the set of all events that occur at node i in H. Two histories S and H are *equivalent*, denoted as $S \simeq H$, if and only if $S|i = H|i$, for each $i \in [n]$.

Definition 1. *The sequential specification of the snapshot object consists of all sequential history S where each* SCAN *operation sc in S returns a vector $Snap$ such that*

- *$Snap[i] = v$, if* UPDATE*(v) is the most recent* UPDATE *operation by node i that appears before scan sc in S;*
- *$Snap[i] = \perp$, if no such* UPDATE *exists.*

A history is *legal* if it belongs to the sequential specification of the snapshot object. We are now ready to formally define sequentially consistent and atomic snapshot objects:

Definition 2 (SSO [35])**.**
- *A history $(H, <_H)$ of the snapshot object is sequentially consistent if there exists a legal sequential history S such that $S \simeq H$. We say that S is a sequentialization of H.*
- *An implementation of the snapshot object is sequentially consistent if each history is sequentially consistent. Such a snapshot object is a sequential snapshot object (SSO).*

Definition 3 (ASO [2])**.**
- *A history $(H, <_H)$ of the snapshot object is linearizable if there exists a legal sequential history L such that $L \simeq H$ and L preserves the real-time order of operations in H, i.e., if $op \rightarrow op'$ in H, then op is placed before op' in L. L is a linearization of H.*
- *An implementation of the snapshot object is linearizable (or atomic) if every history is linearizable. We call such a snapshot object as atomic snapshot object (ASO).*

Figure 1 illustrates an example history, along with a sequentialization and a linearization.

C. Notations

Table II presents the notations and symbols that we will use in the rest of the paper.

III. ALGORITHM EQ-ASO

We begin with a warm-up on the "one-shot ASO" problem, in which each node invokes *at most one* UPDATE operation. We discuss a key challenge for an efficient implementation, followed by an introduction of our main technique, *equivalence quorum*. Next, we present our algorithm EQ-ASO for

TABLE II
COMMONLY USED NOTATIONS

Symbol	Definition
n	number of nodes
f	upper bound on the number of faulty nodes
k	number of actual failures in an execution, $k \leq f$
D	upper bound on the message delay
$op_1 \rightarrow op_2$	op_1 occurs before op_2; that is, the response of op_1 occurs before the invocation of op_2
H	history, an execution of a snapshot represented as a partially ordered set
$U_{i,H}$	set of UPDATE's by node i in history H
$U_{i,H}^{\leq op}$	all UPDATE's in $U_{i,H}$ that occur before op and includes op
B	Base of a SCAN (Definition 4)
$\text{EQ}(V_i, i)$	A predicate used in our technique – equivalence quorum (Definition 6)
$V_i[j]$	Node i's view of the set of values (and UPDATE's) that node j has learned so far

implementing a (multi-shot or a long-lived) ASO, followed by the correctness proof and time complexity analysis. Our algorithm is correct given $n > 2f$, and achieves amortized constant time complexity if there are $\Omega(\sqrt{k})$ operations, where k is the number of crash faults in a given execution. If there is no failure, EQ-ASO achieves constant-time operations unconditionally.

A. Tight Conditions for ASO

We first present necessary and sufficient conditions for correct ASO implementations. Our conditions consider a history (a partial order on UPDATE and SCAN operations) for a given execution of an atomic snapshot object implementation $\mathbb{A}$. If the history satisfies our conditions, we provide a mechanism to identify serialization points so as to satisfy the atomicity property of $\mathbb{A}$. Hence, if all the possible executions of $\mathbb{A}$ satisfy our conditions, then $\mathbb{A}$ correctly implements ASO. We also show that the conditions are necessary by arguing that if any of the conditions is violated, then it is impossible to identify a linearization in some history of $\mathbb{A}$. Similarly, we identify tight conditions for SSO (in [25]), which also shed light on the difference between atomicity and sequential consistency. Later we show that our algorithm satisfies these conditions.

Useful Definition and Lemmas: To facilitate the discussion, we define the notion of *base* of a SCAN operation. Without loss of generality, we assume that all UPDATE operations are unique.[2] Consider a given history of the snapshot object H. Let $U_{i,H}$ denote the set of UPDATE operations by node i in history H. For any UPDATE operation $op \in U_{i,H}$, we use $U_{i,H}^{\leq op}$ to denote all UPDATE operations in $U_{i,H}$ that occurs before op and includes op. For completeness, we also introduce a special operation that writes $\perp$ to each segment at initialization, called a NULL operation. If op is an NULL operation, then $U_{i,H}^{\leq op} = \emptyset$. Since each node is assumed to be sequential, $U_{i,H}^{\leq op}$ is well defined. In Figure 1, $U_{1,H} = \{\text{UPDATE}(1), \text{UPDATE}(4)\}$, $U_{1,H}^{\leq \text{UPDATE}(1)} = \{\text{UPDATE}(1)\}$, and $U_{1,H}^{\leq \text{UPDATE}(4)} = \{\text{UPDATE}(1), \text{UPDATE}(4)\}$.

[2]This can be easily achieved by piggybacking a sequence number and a writer ID. Moreover, since there is at most one pending operation for each node, UPDATE operations originating from the same node can be ordered sequentially. That is, for any two updates originating from the same node, one can tell which update occurs earlier (by comparing the sequence number).

Definition 4 (Base of SCAN)**.** *Consider a* SCAN *operation sc that returns the vector $Snap$ in H. For each $i \in [n]$, let op_i denote the* UPDATE *operation at node i that writes the value $Snap[i]$ or a NULL operation if $Snap[i] = \perp$. The base of sc with respect to H is then defined as the set $\bigcup_{i=1}^{n} U_{i,H}^{\leq op_i}$.*

For a given history, denote by B_{sc} the base of a SCAN operation sc. For a slight abuse of terminology, we use $B_{sc}[i]$ to denote the set of UPDATE's by node i in B_{sc}. In our presentation below, B is always used to denote a base.

Definition 5 (Comparable Bases)**.** *The bases B_1 and B_2 of two* SCAN *operations are comparable if $B_1 \subseteq B_2$ or $B_2 \subseteq B_1$.*

Consider the execution in Figure 1, the base of $op5$ is $\{\text{UPDATE}(1), \text{UPDATE}(2), \text{UPDATE}(3), \text{UPDATE}(4)\}$. The base of $op4$ is $\{\text{UPDATE}(1), \text{UPDATE}(2), \text{UPDATE}(3)\}$. These two bases are comparable.

Necessary and Sufficient Conditions: We present the *tight* conditions for ASO in Theorem 1. Our conditions are inspired by the observations made in prior works studying ASO in shared memory [11], [12], [14], [19].

Theorem 1. *An implementation of the snapshot object is linearizable* ***if and only if*** *all of the following conditions are satisfied in each history H:*

(A1) *The bases of any pair of* SCAN *operations in H are comparable.*

(A2) *The base of a* SCAN *operation contains all* UPDATE *operations that precede it in H.*

(A3) *For two* SCAN *operations sc_1 and sc_2, if $sc_1 \rightarrow sc_2$ in H, then the base of sc_2 is at least the base of sc_1, i.e., $B_{sc_1} \subseteq B_{sc_2}$.*

(A4) *If an* UPDATE *op belongs to the base of a* SCAN *sc, then all* UPDATE*s that precedes op in H must belong to base of sc.*

In [25], we present the full proof for the theorem above. The following is the high-level steps that we use to construct a linearization for any given history that satisfies the above conditions, which proves the sufficiency part.

- Step I: construct a sequence L' of all SCAN operations ordered by their bases. For any two SCAN 's with the same base, they are ordered in L' by their real time order.
- Step II: construct a linearization L by inserting all UPDATE operations in L' as follows. An UPDATE operation is placed before the first SCAN operation in L' whose base contains it. For any two UPDATE operations placed between the same pair of SCAN operations, order them by their real-time order.

B. One-shot ASO: Key Challenge

Among the four conditions identified for an ASO algorithm, condition (A1) is the most non-trivial to achieve efficiently.

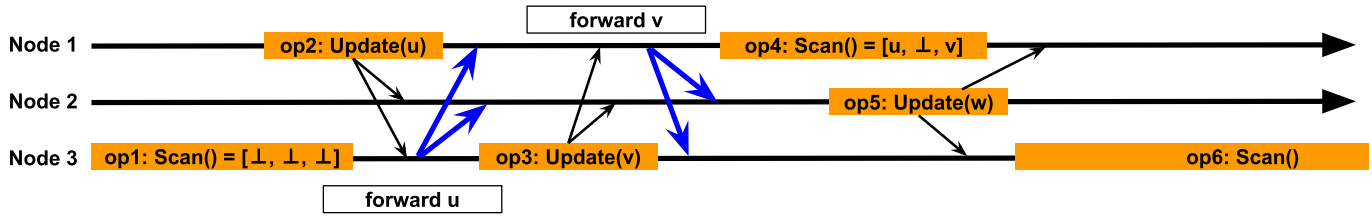

Fig. 2. An example execution history of the one-shot ASO. Black arrows denote the values sent by the UPDATE operations, whereas blue arrows denote the forwarded values. White box represents the moment when a node forwards a value. The base (Definition 4) and predicate EQ (Definition 6) for the SCAN operations are presented below:
- When $op1$ is invoked, $V_3[1] = V_3[2] = V_3[3] = \{\}$, and the base is $\{\}$.
- When $op4$ is invoked, $V_1[1] = V_1[3] = \{u, v\}$, and the base is $\{op2, op3\}$.
- When $op6$ is invoked, $V_3[1] = \{u, v\}, V_3[2] = \{w\}, V_3[3] = \{u, v, w\}$, so it cannot be returned yet. It has to wait for forwarded values (blue arrows) from either node 1 or 2 so that the predicate $\text{EQ}(V_3, 3)$ becomes satisfied.

With a proper integration, the other three conditions can be satisfied by typical techniques that ensure quorum intersection. Prior works on ASO in message-passing systems [12], [19] rely on a technique called *double-collect* to ensure (A1). Double-collect is used in many shared-memory algorithms, e.g., [2], [15], [24], and based on a simple principle: if two consecutive "collect" operations (i.e., reading values from a majority of nodes) return the same set of values, then no update has been applied concurrently. Essentially, this principle allows one to treat a double-collect as occurring instantaneously, which is important for identifying a linearization for a given execution. A major downside of double-collect is its time complexity, even in the case of one-shot ASO. Since nodes may invoke UPDATE concurrently, a naive application of double-collect incurs $O(n \cdot D)$ time for SCAN operations. This occurs when a node has to keep collecting new information to complete the double-collect, i.e., obtaining matching consecutive collects.

C. Our Main Technique: Equivalence Quorum

While double-collect is natural in shared memory, we make a key observation that in message-passing systems, a node can *proactively* notify others regarding new information. Double-collect requires nodes to query others to learn new UPDATE operations, whereas we ask each node to proactively forward values from new UPDATE operations. Consequently, a node only needs to check a *local predicate* to determine whether comparable bases can be obtained.

We formalize the approach as the *equivalence quorum*, which is inspired by the *stable vector* by Attiya et al. [9] and Mendes et al. [33]. Each node i maintains a vector of value sets V_i for storing incoming values forwarded by each node (including ones from i). If i is clear from the context, we often omit it for brevity. For node i, $V_i[1, \ldots, n]$ is a vector of size n, where for each $j \in [n]$, $V_i[j]$ is a set of values. We then define the predicate $\text{EQ}(V, i)$ that needs to be checked locally by node i.

Definition 6 (Predicate $\text{EQ}(V, i)$)**.** *$\text{EQ}(V, i)$ is true iff $\exists Q \subseteq \{1, \cdots, n\}$ s.t. $|Q| \geq n - f \wedge V[j] = V[i], \forall j \in Q$. When the predicate is true, we call Q as the **equivalence quorum** and $V[i]$ as the **equivalence set**.*

Consider an example with $n = 3$. Suppose node 1's vector is as follows: $V_1[1] = \{u, v\}$, $V_1[2] = \{\}$ and $V_1[3] = \{u, v\}$. In this case, $\text{EQ}(V_1, 1)$ holds. $\{1, 3\}$ forms a equivalence quorum, and $\{u, v\}$ is the equivalence set.

The equivalence set is a set of values, whereas the base of a SCAN is a set of UPDATE operations. Recall that we assume that all UPDATE operations are unique. Hence, each value corresponds to a unique UPDATE operation, and we can obtain a set of UPDATE operations by replacing each value with its corresponding UPDATE. Later in Section III-E, we formalize a mapping between equivalence sets and bases.

One-Shot ASO based on Equivalence Quorum: To implement a one-shot ASO, each UPDATE operation sends its value to all other nodes and waits for a quorum of acknowledgements. Upon receiving a new value from node j, node i first adds the value into $V_i[i]$ and $V_i[j]$, and then forwards the value to all other nodes. Due to our assumption of FIFO channels, $V_i[j]$ essentially represents i's view of the set of values (and UPDATE's) that node j has learned so far. A SCAN operation at node i needs to wait until the $\text{EQ}(V_i, i)$ predicate becomes true and obtains the corresponding equivalence set V^*. The SCAN operation then returns a vector $Snap$ defined as: $Snap[j] = v$ if $v \in V^*$ and v is written by node j; otherwise $Snap[j] = \perp$.

Consider an illustration in Figure 2. Black arrows denote the values sent by some UPDATE operation, whereas blue arrows denote the forwarded values. In the example, we have

- $op1$ returns immediately at node 3, as $V_3[1] = V_3[2] = V_3[3] = \{\}$. By definition, both the equivalence set and the base of $op1$ are $\{\}$.
- $op4$ returns immediately at node 1, as $V_1[1] = V_1[3] = \{u, v\}$ and $V_1[2] = \{\}$. The equivalence set is $\{u, v\}$; hence, the base of $op4$ is $\{op2, op3\}$.
- $op6$ cannot return, because $V_3[1] = \{u, v\}, V_3[2] = \{w\}$ and $V_3[3] = \{u, v, w\}$. Node 3 has to wait for more forwarded values from either node 1 or 2 so that $\text{EQ}(V_3, 3)$ becomes true. Once it is able to return, $op6$ must return $\{u, v, w\}$. This is because $V_3[3] = \{u, v, w\}$. The base is then $\{op2, op3, op5\}$.

If $op6$ returns, then bases of all three SCAN operations are comparable with each other. This is not by coincidence, and

is indeed due to the design of equivalence quorum. Observe that the predicate $\texttt{EQ}(V_i, i)$ is true if there exists a quorum of at least $n - f$ nodes (i.e., equivalence quorum) such that the set of values sent by these nodes to node i is identical to the set of values known by node i (values stored in $V_i[i]$). This design leads to the following observation:

Observation 1. *For any nodes i, j, s, $V_i[s]$ at some time t and $V_j[s]$ at some time t' are always comparable. (Time t might or might not equal to t'.)*

The observation holds because (i) $V_i[s]$ and $V_j[s]$ are the set of values received from node s; (ii) each node is sequential; and (iii) the communication channel is FIFO. Based on the observation, we show the following key lemma that can be used to show comparable bases.

Lemma 1. *When $\texttt{EQ}(V_i, i)$ and $\texttt{EQ}(V_j, j)$ are both true, $V_i[i]$ and $V_j[j]$ are comparable.*

Proof. When $\texttt{EQ}(V_i, i)$ and $\texttt{EQ}(V_j, j)$ hold, there exist quorums Q_i and Q_j of size $\geq n - f$ such that $V_i[i] = V_i[k]$ for each $k \in Q_i$ and $V_j[j] = V_j[k]$ for each $k \in Q_j$. From Observation 1 and the fact that $|Q_i \cap Q_j| \geq 1$, we have that $V_i[i]$ and $V_j[j]$ are comparable. □

A key benefit of our technique is time complexity, especially in the failure-free execution. Consider an extreme case when every message suffers delay D and no node crashes. In this case, every operation terminates in $2D$ time, even if there are concurrent operations. On the contrary, double-collect could still take roughly nD time, when one UPDATE is concurrent with exactly another UPDATE. We will defer a more thorough analysis on the time complexity to Section III-F.

D. From One-Short to Multi-Shot ASO

Now consider the general case of ASO, where nodes can invoke any number of UPDATE operations. In this case, the simple mechanism of forwarding and checking predicate no longer works, because $\texttt{EQ}(V_i, i)$ may never be satisfied when there are concurrent and infinite number of UPDATE's. One straightforward idea is to associate a tag (or a sequence number) with each value and the equivalence quorum predicate. Concretely, let $V^{\leq r}$ denote the set of values with tag $\leq r$ in V. Node i can then wait for $\texttt{EQ}(V_i^{\leq r}, i)$ to be true and return the most recent values (with respect to the tags) in the equivalence set. However, this solution does not ensure comparability, i.e., when both $\texttt{EQ}(V_i^{r}, i)$ and $\texttt{EQ}(V_j^{r'}, j)$ are true, the two equivalence sets are *not* necessarily comparable. Our technical report [25] presents such an example.

We present algorithm EQ-ASO, which is inspired by [11]. The pseudo-code is presented in Algorithm 1. Our key innovation lies in the usage of proactive forwarding and equivalence quorum. Before delving into details, we present two more techniques to integrate equivalence quorum with the rest of the algorithm.

Lattice Operation and Lattice Renewal: We use two techniques to address the comparability issue:

(T1) After $\texttt{EQ}(V_i^{\leq r}, i)$ becomes true, node i does not immediately return the corresponding equivalence set. Instead, it checks whether tag r is the largest tag that it has received so far. If yes, node i returns the equivalence set; otherwise, it continues to wait $\texttt{EQ}(V_i^{\leq r'}, i)$ to hold, where $r' > r$ is the largest tag it has received so far.

(T2) If node i keeps receiving a larger tag due to a concurrent UPDATE, its equivalence quorum predicate might never be satisfied. We use the "borrowing" technique – node i borrows an equivalence set from another node after it has already satisfied the equivalence predicate for three different tags.

Our algorithm is designed in a way that node i either obtains an equivalence set or borrows an equivalence set from some other node. We call the process of collecting values to satisfy the equivalence predicate as a *lattice operation*.[3] Recall that each value written by the UPDATE is assigned a tag. Hence, by "nodes participate in lattice operation with the same tag," we mean that these nodes try to collect values to satisfy equivalence quorum with the same tag.

To facilitate the discussion, we present the notion of "view," which represents a set of *values* that is observed by a node i. At some point of time, the view satisfies the predicate $\texttt{EQ}(V, i)$ with a certain tag (after collecting enough values with the tag). Node i then uses techniques (T1) and (T2) to determine which view is safe to return. View should not be confused with the notion of base defined in Definition 4. "View" is a set of values that is defined with respect to different operations (including lattice operations), whereas "Base" is a set of operations that is defined with respect to a SCAN operation. There indeed exists a one-to-one mapping between view and base as we will show later.

In EQ-ASO, UPDATE and SCAN operations might need to invoke multiple lattice operations. Depending on the tag observed locally, an operation may obtain a direct view (technique (T1)) or an indirect view (technique (T2)). We use the term "lattice renewal" to refer to the process of invoking lattice operations and determining the view to be returned.

Algorithm Description and Pseudocode: For readability, we adopt the thread-based and event-driven presentation following the prior work, e.g., [8], [12]. In practical implementation, a process-based framework could potentially be used to improve efficiency by using shared memory for communication. We first describe key variables, followed by the procedures and message handlers.

Variables: Each value (written by UPDATE) is associated with a timestamp of the form $\langle r, j \rangle$, where r is the tag and j is the ID of the writer who initiates the UPDATE. The exact value of the tag in a timestamp is determined in the UPDATE operation. For brevity, we often use value to denote a *value-*

[3] [25] abstracts and adapts the lattice operation into an *early-stopping* algorithm for solving the lattice agreement problem [11].

Algorithm 1 ASO: Code for node i

Local Variables: /* These variables can be accessed and modified by any thread at i. */
$V[1 \cdots n]$ ▷vector of ``views''. $V[j]$ is the set of values received from j
$maxTag$ ▷integer, largest tag ever seen via "$writeTag$", "$echoTag$" messages.
$D[1 \cdots n]$ ▷vector of views from good lattice operations.

Derived Variable:
$V^{\leq r} \leftarrow [V[1]^{\leq r}, V[2]^{\leq r}, \ldots, \ldots, V[n]^{\leq r}]$ ▷vector of ``views'' w/ tag at most r

Initialization:
1: $V \leftarrow [\emptyset, \emptyset, \cdots, \cdots, \emptyset]$
2: $D \leftarrow [\emptyset, \emptyset, \cdots, \cdots, \emptyset]$
3: $maxTag \leftarrow 0$

When UPDATE(v) **is invoked**:
4: $r \leftarrow readTag()$
5: $ts \leftarrow \langle r+1, i \rangle$
6: Send ("$value$", $\langle v, ts \rangle$) to all
7: Lattice(r) ▷Phase 0
8: $r' \leftarrow \max\{r+1, maxTag\}$
9: $updateView \leftarrow$ LatticeRenewal(r')
10: **Return** ACK

When SCAN() **is invoked**:
11: $r \leftarrow readTag()$
12: $scanView \leftarrow$ LatticeRenewal(r)
13: **Return** $extract(scanView)$

/* Helper procedures */
Procedure Lattice(r)**:**
14: $writeTag(r)$
15: **Wait until** $\texttt{EQ}(V^{\leq r}, i) = True$
/* Execute lines 16 to line 21 atomically */
16: $V^* \leftarrow$ equivalence set of satisfied $\texttt{EQ}(V^{\leq r}, i)$
17: **if** $maxTag \leq r$ **then**
18: Send ("$goodLA$", r) to all
19: **Return** $(true, V^*)$
20: **else**
21: **Return** $(false, \emptyset)$

Procedure LatticeRenewal(r):
22: **for** $phase \leftarrow 1$ to 3 **do**
23: $(status, view) \leftarrow Lattice(r)$
24: **if** $status = true$ **then**
25: **Return** $view$ ▷Direct View
26: **else if** $phase = 3$ **then**
27: **Break**
28: $r \leftarrow maxTag$
29: **Wait until** receiving ("$goodLA$", r) from some node j
30: **Return** $D[j]$ ▷Indirect View

Procedure extract(S):
31: $Snap \leftarrow [1, \cdots, n]$
32: **for** $j = 1$ to n **do**
33: $Snap[j] \leftarrow v$, where $\langle v, \langle t', j \rangle \rangle \in S$, and t' is the largest tag of j's values in S
34: **Return** $Snap$

Procedure readTag():
35: Send ("$readTag$") to all
36: **Wait until** receiving $\geq n - f$ ("$readAck$", $*$) msgs
37: **Return** largest tag contained in $readAck$ msgs

Procedure writeTag(tag):
38: Send ("$writeTag$", tag) to all
39: **Wait until** receiving $\geq n - f$ ("$writeAck$", tag) msgs

/* Event handlers: executing in background */
/* All event handlers executed atomically */
Upon receiving ("$value$", $\langle u, ts \rangle$) from node j:
40: Add $\langle u, ts \rangle$ into $V[j]$, $V[i]$
41: **if** $\langle u, ts \rangle$ has *not* been seen before **then**
42: Send ("$value$", $\langle u, ts \rangle$) to all

Upon receiving ("$writeTag$", tag) from node j:
43: **if** $tag > maxTag$ **then**
44: $maxTag \leftarrow tag$
45: Send ("$echoTag$", tag) to all
46: Send ("$writeAck$", tag) to node j

Upon receiving ("$echoTag$", tag) from node j:
47: $maxTag \leftarrow \max\{maxTag, tag\}$

Upon receiving ("$readTag$") from node j:
48: Send ("$readAck$", $maxTag$) to node j

Upon receiving ("$goodLA$", r) from node j:
49: $D[j] \leftarrow V[j]^{\leq r}$ ▷borrow node j's view

/* NOTE: All our event handlers are atomic;
hence, when receiving a ("$goodLA$", $*$)
Line 49 will be executed before Line 29
if there is a pending $LatticeRenewal()$ */

timestamp pair. Recall that for a set of values U, we use $U^{\leq r}$ to denote the subset of values in U with tag at most r.

At all time, each node i keeps track of a vector V_i of size n, which represents the vector of "views" at node i. Formally, for $j \in [n]$, $V_i[j]$ is the set of written and/or forwarded values that i has received from node j. In our design, each node i needs to forward a value it receives for the first time. In this case, we say a value is forwarded by i. Two other variables are related to V_i: (i) $V_i^{\leq r}$ is the vector of views with tag at most r, i.e., $V^{\leq r} = [V[1]^{\leq r}, V[2]^{\leq r}, \ldots, V[n]^{\leq r}]$; and (ii) $D_i[j]$ is a *particular view borrowed from node j* that can be "safely" returned. The meaning of "safe" will become clear when we discuss the lattice operation.

Each node also keeps track of a variable $maxTag$, which represents the largest tag it has ever received via "$writeTag$" messages or "$echoTag$" messages. Note that it is possible that there are some values with tag larger than $maxTag$ in V_i. We now describe the procedures.

Lattice(r): Each node uses the $Lattice(r)$ procedure to invoke lattice operation with tag r. The goal is to check whether the equivalence quorum is satisfied (technique (T1) discussed before). Consider a Lattice(r) invocation at node i. It first writes the input tag r to at least $n-f$ nodes. Then node i waits until the *equivalence quorum* predicate (Definition 6) becomes true for the first time. After that if the $maxTag$ value is strictly larger than r, then the lattice operation returns $\langle false, \emptyset \rangle$. Otherwise, it returns $\langle true, V^* \rangle$, where V^* is the equivalence set. In this case, $Lattice(r)$ is said to be a *good* lattice operation.

LatticeRenewal(r): The $LatticeRenewal(r)$ procedure also has a parameter r. The goal of this procedure is to determine which view to return. It invokes *at most* three lattice operations, as outlined in technique (T2). If some lattice operation is good, the $LatticeRenewal(r)$ returns the view obtained by the good lattice operation, i.e., *direct view*. If the first two lattice operation are *not* good, then by definition, it means that node i has observed a larger tag, i.e., condition at line 17 returns $false$. Therefore, node i initiates the next lattice operation with tag equal to $maxTag$. If the third lattice operation is also not good, then node i waits for a "$goodLA$" message from some other node j to obtain a view from j's good lattice operation. In this case, the view is called an *indirect view*.

Update(v): To write value v, node i first obtains a tag by reading from at least $n-f$ nodes. Let r denote the largest tag in the received $readAck$ messages. Then, i constructs the timestamp of value v as the $\langle r+1, i \rangle$ tuple. It sends value v with its timestamp to all nodes. Then, a lattice operation with tag r is invoked. This step is called the *phase-0* lattice operation of the UPDATE operation. After the phase-0 lattice operation, the UPDATE obtains a new tag r' and executes $LatticeRenewal(r')$. The view returned by $LatticeRenewal(r')$ is *not* used; and hence discarded. Node i returns the ACK to complete the UPDATE.

A subtle point here is that the operation executes the phase-0 lattice operation *before* invoking $LatticeRenewal(r)$. The way we devise them ensures that for each tag, there is always a *good* lattice operation. Intuitively, this operation ensures that each node is able to obtain a comparable view.

Scan(): SCAN first obtains a tag r, and executes the $LatticeRenewal(r)$, which returns a view $scanView$. Then it returns the most recent value by each node in $scanView$ by executing the $extract(scanView)$ procedure.

Message Handlers: All the handlers execute in the background *atomically*. That is, even if a node does not have a pending UPDATE or SCAN operation, it continues processing messages, and during the period that a handler is executing, no other part of the code can take step. One subtle part to note is that a node does *not* update its $maxTag$ variable when it receives a value with a larger tag from a "$value$" message. The $maxTag$ variable is only updated when a node receives a "$writeTag$" message or "$echoTag$" message. This design helps EQ-ASO achieve the desired time complexity. Especially, we rely on this design to prove that there is a good lattice operation for each tag.

E. Correctness of EQ-ASO

For correctness, we show that (i) each operation in EQ-ASO terminates, and (ii) each possible execution history of EQ-ASO satisfies conditions (A1)-(A4) presented earlier. First, we formally define important terms and concepts.

Definition 7 (Tag of UPDATE or SCAN)**.** *The tag of an* UPDATE *or* SCAN *is the tag of its last lattice operation.*

Definition 8 (Timestamp/Tag of a value)**.** *The timestamp of a value v is the $\langle r+1, i \rangle$ (tag-ID tuple) at line 5 in the* UPDATE(v) *procedure. The tag of a value is defined as the tag contained in its timestamp. For value v, we use ts_v to denote its timestamp and tag_v to denote its tag.*

Definition 9 (View)**.** *We define the views for a node and operations as below:*

- *For a node i, its view at some time t is defined as the set $V_i[i]$ at time t.*
- *For a good lattice operation with tag T ($Lattice(T)$) by node i, its view is defined as the set of values it returns.*
- *For an* UPDATE *or* SCAN *, its view is defined as the set of values returned by its $LatticeRenewal()$ call. We say that an* UPDATE *or* SCAN *operation obtains a direct view if its $LatticeRenewal()$ call returns at line 25; otherwise it obtains an indirect view.*

Termination: Later in Section III-F, we show that a lattice operation takes $O(\sqrt{k} \cdot D)$ time to complete, where $k \leq f$ is the number of actual faults in an execution. Since each lattice operation terminates, the only blocking part left in EQ-ASO is line 29 of the $LatticeRenewal()$ procedure. To show line 29 eventually returns, we claim that there exists a good lattice operation for each tag. This is primarily guaranteed by the use of the phase-0 lattice operation at line 7. Intuitively, suppose node i is the first node that writes tag r in its $LatticeRenewal()$ call, then the phase-0 lattice operation (with tag $r-1$) of node i must be a good lattice operation.

Our algorithm also ensures that the tags are non-skipping, i.e., if there is a tag r, there must also be a tag $r-1$. Thus, there must be a good lattice operation for each tag. Consequently, we can show that $LatticeRenewal()$ either obtains a direct view or borrows an indirect view. Therefore, we can show that each operation in EQ-ASO terminates.

Conditions (A1)-(A4): The full proof is presented in [25] due to lack of space. We first present a one-to-one mapping between the view of a SCAN operation and its base. For any SCAN operation sc with view $View_{sc}$, we use VB_{sc} to denote the set of UPDATE's if we replace each value in $View_{sc}$ with its corresponding UPDATE operation. We call VB_{sc} as the "view-induced base" of sc. We show in our technical report that the view-induced base is equivalent to the base as defined in Definition 4. In the sequel, we always use B_{sc} to denote the base of a SCAN operation sc and VB_{sc} to denote the view-induced base of sc. To prove condition (A1), we only need to show the following lemma which relies on our usage of lattice operation and the equivalence quorum predicate.

Lemma 2. *The views of any pair of good lattice operations are comparable.*

Proof Sketch. First observe that for any two nodes i and j and tag T, the set $V_i[s]^{\leq T}$ at time t and the set $V_j[s]^{\leq T}$ at time t' are comparable for any time t and t' and any $s \in [n]$. Note that this observation is a generalization of Observation 1 presented earlier. Now, consider any two lattice operations op_i and op_j with tag T_i and T_j. If $T_i = T_j$, then the observation and the equivalence quorum predicate together imply that their views must be comparable. Otherwise, assume without loss of generality, $T_i < T_j$. Our algorithm guarantees that the view of op_i must be a subset of the view of op_j. Intuitively, the fact that op_i does not observe T_j at line 17 implies that op_j must complete its line 14 after op_i has completed. This ensures that op_j must have received all values in the view of op_i when op_j starts line 15. □

The key to prove conditions (A2) and (A3) is to show the following lemma.

Lemma 3. *Consider any two operations op_i by node i and op_j by node j with views $View_i$ and $View_j$, respectively. If $op_i \rightarrow op_j$, then $View_i \subseteq View_j$.*

Proof Sketch. Consider any two operations op_i with tag T_i and op_j with tag T_j, respectively. Due to the way EQ-ASO reads and writes tags, we have that if $op_i \rightarrow op_j$, then $T_i \leq T_j$. Based on this observation and the property of equivalence quorum, we can argue that $View_i \subseteq View_j$. This is because for op_j to satisfy the equivalence quorum predicate, it must wait for values with tag T_j. Due to quorum intersection, at least one node must have already observed $View_i$. □

Lemma 4. *(A1)-(A4) hold for each history of* EQ-ASO.

Proof Sketch.
- (A1): This is based on (i) one-to-one mapping between view and view-induced base; (ii) the observation that view of a SCAN operation is same as the view of some good lattice operation; and (iii) Lemma 2.
- (A2): Let $op = \text{UPDATE}(v)$ be an UPDATE operation and sc be a SCAN operation such that $op \rightarrow sc$, we need to show that $op \in B_{sc}$. We claim that $v \in View_{op}$. Lemma 3 implies that $View_{op} \subseteq View_{sc}$. So, we have $v \in View_{sc}$. Thus, $op \in B_{sc}$ by the one-to-one mapping.
- (A3): Let sc_1 and sc_2 be two SCAN operations such that $sc_1 \rightarrow sc_2$, we show that $VB_{sc_1} \subseteq VB_{sc_2}$. This can be immediately obtained from Lemma 3.
- (A4): Let sc be a SCAN operation, UPDATE(u) and UPDATE(v) be any two UPDATE operations such that UPDATE(u) $\rightarrow$ UPDATE(v) and UPDATE(v) $\in B_{sc}$. We need to show that UPDATE(u) $\in B_{sc}$. Let node i be the node that executes UPDATE(v). Since UPDATE(u) $\rightarrow$ UPDATE(v), we know that before UPDATE(v) starts, value u has been received by a quorum of nodes and sent out to all. Thus, when UPDATE(v) reads tag (via a "$readTag$" message) from a quorum of nodes at line 4, node i must have received value u from at least one node due to the assumption of FIFO channel. Hence, node i must send value u to all before sending value v. Therefore, any node which receives value v must have already received value u. Thus, UPDATE(u) $\in B_{sc}$.

□

Lemma 4 and conditions (A1)-(A4) imply that EQ-ASO correctly implements ASO.

F. Time Complexity of EQ-ASO

To ensure that each lattice operation terminates in $O(\sqrt{k} \cdot D)$ time, EQ-ASO uses a simple approach: *increment tags and invoke lattice operation(s) in a way that later* UPDATE *does not prevent the progress of existing lattice operations.* Concretely, consider a lattice operation with tag T, $Lattice(T)$, that starts at time t. EQ-ASO ensures that (i) all values from UPDATE operations that start after time $t + D$ must have tag strictly greater than T; and (ii) slow writers that participate in $Lattice(T)$ after time $t + D$ do *not* introduce any new value with tag T. These two properties along with a "failure chain" argument can be used to prove that the lattice operation in EQ-ASO takes $O(\sqrt{k} \cdot D)$ time in the worst case.

Brief Analysis: We present a brief analysis below due to page constraints. The full proof is presented in our technical report [25]. We first show that each lattice operation takes $O(\sqrt{k} \cdot D)$ time.

Definition 10 (Exposed value in an interval)**.** *We say a value v is an exposed value in interval $[t, t + D]$ if some non-faulty node receives v in interval $[t, t + D]$, and no non-faulty node has received v before time t.*

The lemma below follows from Definition 10 and the observation that $2D$ is long enough for all non-faulty nodes to learn up-to-date information if there is no exposed value due to the proactive forwarding design.

Lemma 5. *[Termination Without Exposed Value] Let L be a lattice operation that starts at time t with tag T. If there is no exposed value with tag $\leq T$ in some interval $[t', t'+2D]$ where $t' \geq t$, then L terminates by time $t'+2D$.*

We introduce the notion of *failure chain* for an exposed value as below.

Definition 11 (Failure chain of an exposed value)**.** *A sequence of nodes $p_1, p_2, ..., p_m$ is said to form a failure chain of an exposed value v if (i) $p_1, p_2, \ldots, p_{m-1}$ are faulty, and p_m is correct; (ii) p_1 executes* UPDATE*(v); (iii) p_i receives value v from p_{i-1}; and (iv) For $1 \leq i < m-1$, p_i crashes while sending (v, p_i) to other nodes, i.e., p_1 crashes when executing line 6 and $p_2, ..., p_{m-2}$ crash when executing line 42.*

Definition 11 and a simple counting argument give us the following two lemmas.

Lemma 6. *Suppose value v is sent to all at time t at line 6 of* UPDATE*(v). If value v is an exposed value in some interval $[t', t'+D]$ where $t' > t$, then value v has a failure chain of length at least $\frac{t'-t}{D} + 1$.*

Lemma 7. *For any two exposed values v and u with failure chain P_v and P_u respectively. Then, the first $|P_v| - 2$ nodes in P_v and the first $|P_u| - 2$ nodes in P_u are disjoint.*

These two lemmas imply that each exposed value in some interval can be associated with a unique set of faulty nodes. Since the number of faulty nodes are bounded by k, the number of consecutive intervals of length $2D$ with exposed values can also be bounded. Along with Lemma 5, we have:

Lemma 8. *If there is $\leq k$ faulty nodes in an execution, each lattice operation takes $O(\sqrt{k} \cdot D)$ time.*

We use Lemma 8 and the following two observations to derive the worst-case time complexity of EQ-ASO, $O(\sqrt{k} \cdot D)$, and its amortized constant time complexity:

- Each UPDATE or SCAN operation invokes at most three lattice operations;
- Once a faulty node crashes, it will never introduce an exposed value by definition. In other words, it will never delay any further operation.

Amortized complexity analyzes the "average" time the algorithm takes to complete an operation given the worst possible failure and delay pattern. Due to the second observation above, the amortized time improves, as the worst-case performance for a small number of operations becomes "averaged out" over enough operations.

Since the worst-case occurs with a particular failure and message delay pattern (i.e., failure chain argument in Definition 11), we envision that it will not be the common case in practice. Note that the performance is degraded by failures in terms of k. For amortized time, k does not show up, because our algorithm achieves constant time complexity if there are at least $\sqrt{k}$ operations.

IV. Related Work

ASO and SSO can be viewed as an extension of atomic registers [8] and sequentially consistent registers [22], respectively. ASO is a well studied problem in the shared memory literature. It is first studied by Afek et al. in [2], where they propose the well known *double-collect* technique. References [15], [17], [24], [32] present ASO implementations using variations of double-collect on n single-writer multi-reader (SWMR) registers. Among these algorithms, a recent work [17] achieves the best time complexity – logarithm time complexity in both UPDATE and SCAN operations.

Attiya et al. in [11] present an algorithm which transforms any algorithm for lattice agreement to an algorithm for atomic snapshot object. Attiya et al. in [14] later present an implementation that takes $O(n \log n)$ operations on SWMR atomic registers. Inoue et al. [30] present an algorithm that requires only a linear number of read and write operations on multi-writer multi-reader (MWMR) registers. Our algorithms improved time complexity, compared to all these prior works, and we are not aware of any prior results on tight conditions for either ASO or SSO.

Lattice agreement (LA) [11] is an agreement problem closely related to ASO. Many LA algorithms have been proposed recently [20], [39]–[42], including the Byzantine LA algorithm [21]. LA is closely related to ASO. In fact, our framework can be used to solve LA and generalized LA problems with a better amortized time complexity. Our algorithms use lattice operations based on the equivalence quorum technique, instead of an existing LA algorithm to improve the amortized time complexity.

V. Conclusion

This paper presents a framework for an ASO implementation with improved time complexity. A key component of our framework can be adapted to an early-stopping algorithm for the lattice agreement problem [11]. In our technical report [25], we present the details of the Byzantine ASO algorithm, which integrates reliable broadcast [18] with our framework. The report also presents how the framework naturally supports an efficient SSO, which completes SCAN operations without any communication by returning the extracted vector from the view stored locally.

Future work includes identifying the lower bound on time complexity and investigating practical applications. It will be interesting to see whether the improvement in time complexity using our framework can be translated to the performance benefits in practical applications, such as cryptocurrency (asset transfer objects), and linearizable CRDT.

References

[1] I. Abraham et al. Optimal resilience asynchronous approximate agreement. In *Principles of Distributed Systems, 8th International Conference, OPODIS*, 2004.

[2] 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.

[3] J. H. Anderson. Composite registers. *Distributed Comput.*, 6(3):141–154, 1993.

[4] J. Aspnes. Time- and space-efficient randomized consensus. *J. Algorithms*, 14(3):414–431, 1993.

[5] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *J. Algorithms*, 11(3):441–461, 1990.

[6] J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous pram model. In *Proceedings of the second annual ACM symposium on Parallel algorithms and architectures*, 1990.

[7] H. Attiya. Efficient and robust sharing of memory in message-passing systems. *J. Alg.*, 34(1):109–127, Jan. 2000.

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

[9] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of the ACM (JACM)*, 37(3):524–548, 1990.

[10] H. Attiya, H. C. Chung, F. Ellen, S. Kumar, and J. L. Welch. Emulating a shared register in a system that never stops changing. *IEEE Trans. Parallel Distributed Syst.*, 30(3):544–559, 2019.

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

[12] H. Attiya, S. Kumari, A. Somani, and J. L. Welch. Store-collect in the presence of continuous churn with application to snapshots and lattice agreement. In SSS, 2020.

[13] H. Attiya, N. Lynch, and N. Shavit. Are wait-free algorithms fast? *Journal of the ACM (JACM)*, 41(4):725–763, 1994.

[14] H. Attiya and O. Rachman. Atomic snapshots in $\mathcal{O}(n \log n)$ operations. *SIAM Journal on Computing*, 27(2):319–340, 1998.

[15] H. Attiya and J. Welch. *Distributed computing: Fundamentals, Simulations, and Advanced topics*, volume 19. John Wiley & Sons, 2004.

[16] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems (TOCS)*, 12(2):91–122, 1994.

[17] M. A. Baig, D. Hendler, A. Milani, and C. Travers. Long-lived snapshots with polylogarithmic amortized step complexity. In PODC 2020.

[18] G. Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[19] 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.

[20] G. A. Di Luna, E. Anceaume, S. Bonomi, and L. Querzoni. Synchronous byzantine lattice agreement in $\mathcal{O}(\log f)$ rounds. *arXiv preprint arXiv:2001.02670*, 2020.

[21] G. A. Di Luna, E. Anceaume, and L. Querzoni. Byzantine generalized lattice agreement. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 674–683, 2020.

[22] N. Ekström and S. Haridi. A fault-tolerant sequentially consistent dsm with a compositional correctness proof. In *International Conference on Networked Systems*, pages 183–192. Springer, 2016.

[23] J. M. Faleiro, S. Rajamani, K. Rajan, G. Ramalingam, and K. Vaswani. Generalized lattice agreement. In PODC 2012.

[24] F. E. Fich. How hard is it to take a snapshot? In *SOFSEM* 2005

[25] V. Garg, S. Kumar, L. Tseng, and X. Zheng Amortized Constant Round Atomic Snapshot in Message-Passing Systems *arXiv preprint arXiv:2008.11837*, 2020.

[26] R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlovič, and D.-A. Seredinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*.

[27] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[28] K. Huang, Y. Huang, and H. Wei. Fine-grained analysis on fast implementations of distributed multi-writer atomic registers. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, 2020.

[29] D. Imbs, A. Mostefaoui, M. Perrin, and M. Raynal. Set-constrained delivery broadcast: Definition, abstraction power, and computability limits. In *ICDCN*, 2018.

[30] M. Inoue, T. Masuzawa, W. Chen, and N. Tokura. Linear-time snapshot using multi-writer multi-reader registers. In *International Workshop on Distributed Algorithms*, pages 130–140. Springer, 1994.

[31] S. Kumar and J. L. Welch. Byzantine-tolerant register in a system with continuous churn. *CoRR*, abs/1910.06716, 2019.

[32] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[33] H. Mendes, C. Tasson, and M. Herlihy. The topology of asynchronous byzantine colorless tasks. *arXiv preprint arXiv:1302.6224*, 2013.

[34] N. C. Nicolaou et al. ARES: adaptive, reconfigurable, erasure coded, atomic storage. In *ICDCS*, IEEE, 2019.

[35] M. Perrin, M. Petrolia, A. Mostefaoui, and C. Jard. On composition and implementation of sequential consistency. In *International Symposium on Distributed Computing*, pages 284–297. Springer, 2016.

[36] M. Raynal. *Concurrent programming: algorithms, principles, and foundations*. Springer Science & Business Media, 2012.

[37] J. Skrzypczak, F. Schintke, and T. Schütt. Linearizable state machine replication of state-based crdts without logs. *arXiv preprint arXiv:1905.08733*, 2019.

[38] G. Taubenfeld. *Synchronization algorithms and concurrent programming*. Pearson Education, 2006.

[39] X. Zheng and V. K. Garg. Byzantine lattice agreement in asynchronous systems. In OPODIS, 2020.

[40] X. Zheng and V. K. Garg. Byzantine lattice agreement in synchronous message passing systems. In DISC, 2020.

[41] X. Zheng, V. K. Garg, and J. Kaippallimalil. Linearizable replicated state machines with lattice agreement. In OPODIS, 2019.

[42] X. Zheng, C. Hu, and V. K. Garg. Lattice agreement in message passing systems. In DISC, 2018.