IMPROVEMENTS FOR STORE-COLLECT AND ATOMIC SNAPSHOT OBJECTS UNDER CONTINUOUS CHURN

An Undergraduate Research Scholars Thesis

by

LUIS PANTIN MAYAUDON

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Jennifer L. Welch

May 2021

Majors: Computer Science
Applied Mathematical Sciences

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Luis Pantin Mayaudon, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

Page

ABSTRACT

Improvements for Store-Collect and Atomic Snapshot Objects under Continuous Churn

Luis Pantin Mayaudon
Department of Computer Science and Engineering
Texas A&M University

Research Faculty Advisor: Dr. Jennifer L. Welch
Department of Computer Science and Engineering
Texas A&M University

The field of distributed computing has given rise to many algorithms to share data among nodes in a network. This work focuses on the store-collect and the atomic snapshot objects in an asynchronous, crash-prone message-passing dynamic system with nodes continuously entering and leaving the system. We assume that the maximum number of nodes that enter, leave or crash during some time interval is proportional to the size of the system.

A store-collect object is a distributed object that allows nodes to store data in the system in a variable that can be read by all nodes, but only modified by the node that stored it. This is achieved through two basic operations: the store operation, which stores information into the network, and collect, which collects a copy of all the information stored by every node in the network at the beginning of the time interval in which the operation is active.

The atomic snapshot object is quite similar. It provides two operations, scan and update, that behave in a very similar fashion to the collect and store operations given by the store-collect object; however the atomic snapshot object must satisfy the linearizability condition, which means that it is always possible to arrange all the operations performed into an ordered sequence even if there are operations that occur simultaneously.

This work improves upon the store-collect and atomic snapshot implementations given in At-

tiya et al [SSS, 2020]. We developed a method for quantifying the churn of a network subject to certain assumptions. This new method allows us to prove the correctness of the store-collect algorithm under less restrictive conditions than those found in the original proof of Attiya et al. Additionally, we developed an improved implementation of the atomic snapshot object based on a store-collect object that requires fewer messages to complete a scan or an update operation.

DEDICATION

To my family, who is always supporting me from far away.

ACKNOWLEDGMENTS

Contributors

I would like to thank my faculty advisor, Dr. Welch for her guidance and support throughout the course of this research.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

Funding Sources

This work received no external funding.

1. INTRODUCTION

Distributed computing is an important subfield of computer science. Distributed systems can come in many forms, with the most typical example being a set of computers connected via the internet. A distributed system is composed of distinct entities that communicate with each other to achieve a common goal, unlike a classical centralized system where all the work is performed by a central entity.

One classical problem studied in distributed computing is shared memory. Several processes want to read and write information to the system so that the other processes in the system can access this information to perform their work. However, shared memory is often hard to implement if the system is not well-behaved. In this paper we will be working with a system where processes can enter, leave, and crash at any time, with certain restrictions. The difficulty of working with such environments is the primary motivation behind the two algorithms that we will study in this paper.

In this paper we will be looking at two different distributed objects that provide shared memory, each implemented using message passing. The first object is called *store-collect* [1, 2], and the second object is called *atomic snapshot* [3]. These objects act as building blocks for implementing more complex algorithms by managing the changes and crashed of the system.

Store-collect is a distributed object that supports two basic operations: store and collect. When a node performs a store operation, it writes a value into the system. If one value has already been written by this node, the new value will overwrite the already existing one. When a node performs a collect, it will fetch all the values stored by the other nodes in the system and return a set of (node, value) pairs. We guarantee that if a store operation finishes before a collect operation starts, then the collect will return the value of the store operation or a more recent value. Likewise, if a collect finishes before another collect starts, then the latter view will contain the same values as the former collect, or more recent values. However, we cannot say much about operations that happen concurrently.

Since store-collect object offer little information about concurrent operations, we will consider the atomic snapshot object, which offers stronger restrictions. The atomic snapshot object is similar to the store-collect object, it supports two operations: update and scan, which behave nearly identically to the store and collect operations. However, atomic snapshot objects guarantee linearizability [4]. Informally, an algorithm is linearizable if every sequence of events can be put in a total order such that the time order of non-overlapping operations is respected, and each operation behaves according to the sequential specification of the algorithm.

Chapter 2 contains the work done on the proof of correctness of store-collect. This includes the background theory of the parametrization method based on bitonic sequences, and the application of the method to derive new formulas for the constrains. Chapter 3 contains the new algorithm for atomic snapshots, including a proof of the correctness and linearizability of the implementation. Chapter 4 contains the conclusion of this work. It contains a high level discussion of the results from the previous two chapters.

2. ON THE CONSTRAINTS OF STORE-COLLECT OBJECTS

2.1 Background

2.1.1 Motivation

In the study of distributed algorithms, there are two main paradigms for communication between processes: shared memory and message passing. While message passing is more flexible than shared memory, it is often more difficult to use, especially if we work on systems that constantly change, or where crashes can occur.

The store-collect object [2] offers a solution to these difficulties. It acts as an intermediate layer that provides a simple, yet flexible form of shared memory that is able to work in dynamic and crash-prone systems. We still need to impose some limitations on the rate at which the system composition changes and the ratio of crashed nodes with respect to the size of the system.

The original proof of correctness of the store-collect algorithm in [2] presents these limitations as four inequalities that must be satisfied to ensure correctness. Experimental work done by [5] on the closely related CCReg algorithm [6] suggests that CCReg might be correct for a wider variety of cases than the proof guarantees. Given the similarities between CCReg and the store-collect algorithm, this suggests that the constraints for store-collect objects might also be too conservative. By introducing new mathematical tools into the analysis, we show that the constraints on the churn rate and the failure fraction can be relaxed without significant changes to the structure of the original proof.

The store-collect object supports two concurrent operations, store and collect, that can be performed by a set of nodes. Informally speaking, nodes can share data with the other nodes by *storing* a copy of the data into the store-collect object, then other nodes can access the data of all the other nodes by *collecting* the most recent values from store-collect. Each node can store one variable into the system or update the variable by calling store. Then, they can retrieve the most recent values for each variable by calling collect. We guarantee that after a node finishes a store

or a collect operation, every collect operation that is started afterward will contain the values of the operation or more recent values. However, we do not guarantee that the operations behave instantaneously, so the resulting execution is not always linearizable.

2.1.2 Model Description

The store collect object is implemented using a message passing system. The system consists of a set of clients or nodes with unique identifiers that communicate with each other via messages. The messages are assumed to be sent instantaneously, but take non-zero amount of time to reach their destination. The maximum time to reach the destination is bounded by a fixed constant D that is not known by the nodes. Additionally, the messages are processed on a first-in first out fashion where all the messages from a node p to a node q are processed in the order that they are sent rather than the order that they are received.

The system allows for nodes to enter and leave the system at any time with two restrictions. The size of the system at any time cannot fall below a positive constant $N_{\min}$, which represents the minimum system size. Additionally, we restrict the number of nodes that can enter or leave the system during a time window of length D. If $N(t)$ is the size of the system at time t, then at most $\alpha N(t)$ nodes can enter or leave during $[t, t + D]$, where α is a fixed constant known as the churn rate of the system.

The system also allows nodes to crash, in which case they become unable to receive or respond to any messages; additionally, we assume that crashed nodes do not recover from their crash. A node can crash while sending a message, in which case there is no guarantee that the receiver will get the message. We assume that at any given time t, no more than $\Delta N(t)$ nodes can be in a crashed state, where $N(t)$ is the total number of nodes in the system at time t (including crashed nodes) and Δ is a constant known as the churn the failure fraction of the system.

2.1.3 Algorithm Description

We will now give an informal overview of the store-collect algorithm. The full implementation of the algorithm can be found in the appendix. To track the composition of the system, a node p

maintains a set *Changes* of events concerning the nodes that have entered the system, a set *LView* (local view) containing the values that each node has stored in the system, and an *is_joined* flag that marks whether the node has joined the system. The joining process has two parts. The node first *enters* the system, and then sends a request to join the system. Once enough joined nodes have replied, the node *joins* the system. A node that has entered is considered part of the system, it can send and receive messages, but only nodes that have joined can perform store or collect operations.

When a node enters the system, it adds *enter(p)* to its *Changes* set and broadcasts an **enter** message informing other nodes about its entrance and requesting information about previous events. When a node receives an **enter** message, it updates its *Changes* set and replies with an **enter-echo** message with a copy of *LView* so that the process becomes aware of previous events in the system, and the value of its *is_joined* variable. Once a node receives enough **enter-echo** responses from joined nodes, it joins the system by setting its *is_joined* variable, adding the event *join(p)* to its *Changes* set, and broadcasting a **join** message to inform the other nodes.

To calculate the number of needed responses, the node waits until it receives its first response from a joined node. Then, it updates its *Changes* set with the information received and sets a variable *join_threshold* to equal the number of active nodes (as seen by the node) multiplied by a constant γ. The value of γ is constrained by the churn rate and failure fraction of the system in question. If the value of γ is too high, the algorithm might not terminate, while if the value is too small, the algorithm might not propagate information correctly.

Each node keeps a local copy of the current view in its *LView* variable. The local view consists of a set of triples, $\{\langle p, v, sqno\rangle, \ldots\}$, where p is the (unique) node id, v is the value stored by the node (or $\perp$ if no value has been stored), and *sqno* is the sequence number associated with that value. The sequence number allows us to merge two views by picking the latest value stored by each node according to the highest *sqno*.

In a *collect operation*, a node requests the latest values by sending a **collect-query** message. When a joined node p receives a **collect-query** message, it responds with its local view (*LView*) through a **collect-reply** message. 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. Then the client waits for sufficiently many **collect-reply** messages before broadcasting the current value of its *LView* variable in a **store** message.

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

The threshold for collect and store operations is calculated in a similar fashion as the threshold for joining the system. The number of responses required by an operation is equal to the number of joined nodes (as seen by the node) multiplied by a constant β, which is also constrained by the churn of the system. Setting β is a key challenge in the algorithm as setting it too small might not return correct information from *collect* or *store*, whereas setting it too large might not guarantee termination of the *collect* and *store*.

We will now formally state the system assumptions for this problem. If a sequence of events respects these assumptions, we say that the sequence is *valid*.

Churn Assumption For all times $t > 0$, there are at most $\alpha \cdot N(t)$ ENTER and LEAVE events in $[t, t + D]$.

Minimum System Size For all times $t \geq 0$, $N(t) \geq N_{min}$.

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

2.1.4 Proof Outline

The original proof of the algorithm defines four constraints that bound the values that one can choose for γ and β depending on the values of α, Δ, and $N_{\min}$ for the system. If there is no possible value for γ or β, then the proof does not guarantee correctness. We will revisit the lemmas where these constraints are applied, and use our new parametrization method to derive a new formula for each one of the constraints that will satisfy the role that they play in the original proof.

During the proof, we will make use of the following results from [2]. The original name of each result is given enclosed in parentheses. Some of these results depend on the lemmas that we are improving upon, however we only make use of results that were available at that step of the proof in [2], which protects us from circular arguments.

We define active membership events as the membership events (enter or leave) where the node experiencing the event did not crash during the event, and thus, any messages that the node sent in during the event are guaranteed to be sent.

Lemma 1. *(Lemma 4) For every node p and all times $t \geq t_p^e + 2D$ such that p is active at t, LView$_p^t$ contains all the active membership events for $[0, t - D]$.*

Lemma 2. *(Theorem 3) Every node p that enters at some time t and is active for at least $2D$ time joins by time $t + 2D$.*

Lemma 3. *For every node p and all times t such that p is joined and active at t, Changes$_p^t$ contains all the active membership events for $[0, \max\{0, t - 2D\}]$.*

2.2 Bitonic Sequences

We say that a sequence of events is **bitonic** if every enter event happens before every leave event. In other words, the system grows in size until reaching its maximum size, and then shrinks to its final size. If the system starts with size N_0, then reaches a maximum size of $N_{\max}$, and it finishes with size N_f, then the number of nodes that entered is $N_{\max} - N_0$, and the number of nodes that left is $N_{\max} - N_f$.

We can parametrize the churn behavior of such systems with the help of the functions $F(x, y) = (1+\alpha)^x(1-\alpha)^y$ and $M(x) = (1+\alpha)^x$. Note that M is an increasing function, while F is increasing with respect to the variable x, but decreasing with respect to y. As we will see in the following lemma, these functions are related to the final and maximum size of the system with x representing the number of nodes that enter and y representing the number that leave.

Lemma 4. *Let E be a valid bitonic sequence of events on some interval $[0, i \cdot D]$ for integer i with initial system size N_0. There exist numbers $x \in [0, i]$ and $y \in [0, i - x]$ such that the system size at*

the end of the interval of length is $N_0F(x, y)$, and the maximum system size during the interval is $N_0M(x)$.

Proof. Base case ($i = 1$): Let e be the number of nodes that enter during the interval, l be the number of nodes that leave, and N_f be the final system size. Then, $N_f = N_0 + e - l$.

By the churn assumption, the number of enter events is at most αN_0, hence

$$N_0 \leq N_0 + e \leq N_0(1 + \alpha)$$

By the Intermediate Value Theorem, there exists an $x \in [0, 1]$ such that $N_{\max} = N_0 + e = N_0(1 + \alpha)^x = N_0M(x)$. We will now find a lower bound for N_f.

$$\begin{aligned}
N_f &= N_0 + e - l = N_0 + e + (e - e) - l \\
&= N_0 + 2e - (e + l) \geq N_0 + 2e - \alpha N_0 \quad \text{By the churn assumption} \\
&= N_0(1 - \alpha) + 2(N_0(1 + \alpha)^x - N_0) = 2N_0(1 + \alpha)^x - N_0(1 + \alpha) \\
&= N_0(1 + \alpha)^x[2 - (1 + \alpha)^{1-x}] \\
&\geq N_0(1 + \alpha)^x(1 - \alpha)^{1-x}
\end{aligned}$$

To justify the last inequality, we can just show that $f(z) = \frac{2-(1+\alpha)^z}{(1-\alpha)^z} \geq 1$ for $z \in [0, 1]$. Notice that $f(0) = f(1) = 1$. Furthermore $f''(z) \leq 0$ on $[0, 1]$, so any critical point of f must be a local maximum. Hence, the minimum of f is attained at the bounds of the interval, and thus $f(z) \geq 1$ for $z \in [0, 1]$. Therefore, the last inequality holds.

This means that

$$N_0(1 + \alpha)^x \geq N_f \geq N_0(1 + \alpha)^x(1 - \alpha)^{1-x}$$

And by the Intermediate Value Theorem, there exists a $y \in [0, 1 - x]$ such that $N_f = N_0F(x, y)$.

Induction ($i > 1$): Split the interval into a subinterval of length $(i - 1) \cdot D$ followed by another subinterval of length D. Notice that subintervals of bitonic sequences are also bitonic. Let N_1 be the size of the system at the end of the first subinterval, and let N_f be the system size

at the end of the second subinterval. By the inductive hypothesis, there exist $x_1 \in [0, i-1]$ and $y_1 \in [0, (i-1) - x_1]$ such that $N_1 = N_0 F(x_1, y_1)$, and the maximum size achieved during the first subinterval is $N_0 M(x_1)$. Similarly, there exist $x_2 \in [0, 1]$ and $y_2 \in [0, 1 - x_2]$ such that the maximum size achieved during the second subinterval is $N_1 M(x_2)$ and $N_f = N_1 F(x_2, y_2)$. Let $x = x_1 + x_2$ and $y = y_1 + y_2$. It follows that $x \in [0, i]$ and $y \in [0, i - x]$, and that

$$N_f = N_1(1+\alpha)^{x_2}(1-\alpha)^{y_2} = N_0(1+\alpha)^{x_1+x_2}(1-\alpha)^{y_1+y_2} = N_0 F(x, y)$$

We consider two cases for the maximum size of the system: the first subinterval either contains leaving nodes or it does not. If it contains leaving nodes, then the second subinterval can only contain nodes that leave, forcing x_2 to be zero since the maximum system size of the subinterval is N_1. This means that the maximum size of the whole interval is achieved on the first subinterval. Hence the maximum size of the system is $N_0 M(x_1) = N_0 M(x)$.

Otherwise, the first subinterval only contains entering nodes, which forces y_1 to be zero since N_1 is the maximum system size on the first subinterval, and the maximum is achieved on the second subinterval. Then, the maximum system size is

$$N_1 M(x_2) = N_0(1+\alpha)^{x_1}(1+\alpha)^{x_2} = N_0 M(x)$$

□

The following lemma gives us a method of converting any valid sequence of events into a bitonic sequence with the same number of nodes leaving, entering and crashing.

Lemma 5. *Let E be a valid sequence of events on some interval $[0, i \cdot D]$ for integer i. Suppose that there is a leave event l and an enter event e such that l happens before e. The event sequence E' resulting from swapping the times of the two events is also a valid sequence.*

Proof. Let $N(t)$ be the number of active nodes at time t for E and $N'(t)$ be the number of active nodes at time t for E'. Suppose that l happened at time t_l and e happened at time t_e. If $t < t_l$, then

the swapped events have not happened, so $N(t) = N'(t)$. Likewise if $t > t_e$ then the swapped events have both happened, and changing the order the events happen does not change the overall number of nodes, so $N(t) = N'(t)$. If $t_l \leq t \leq t_e$, then $N'(t) = N(t) + 2$ because l has not happened for E', but e did. Hence $N'(t) \geq N(t)$ for all t.

This means that at any time t, the size of the system is $N'(t) \geq N(t) \geq N_{\min}$. Therefore, the minimum size assumption is respected. For any interval $[t, t + D]$ we have that at most $\alpha N(t)$ events happened for E, and because the swap does not change the number of events on any interval, the number of events in the interval for E' is at most $\alpha N(t) \leq \alpha N'(t)$, so the churn assumption is respected. By the same argument, $\Delta N(t) \leq \Delta N'(t)$ so the failure fraction assumption is also preserved. If a node crashes, the crash will still be its last operation because we do not change the time when crashes happen. Therefore E' is a valid event sequence. □

We can use the two lemmas to parametrize the churn of any event sequence

Theorem 1. *Let E be a valid sequence of events on a time interval of length $i \cdot D$ for integer i. Let the initial size of the system be N_0. There exist real numbers $x \in [0, i]$ and $y \in [0, i - x]$ such that the number of nodes that enter is $N_0(M(x) - 1)$, the number of nodes that leave is $N_0(M(x) - F(x, y))$, and the total number of crashed nodes at the end of the interval is at most $\Delta N_0 F(x, y)$.*

Proof. We can convert E into a bitonic sequence E' by repeatedly apply Lemma 5. This transformation preserves the number of enter, leave, and crash events. By Lemma 4, there exist real numbers $x \in [0, i]$ and $y \in [0, i - x]$ such that the final system size is given by $N_0 F(x, y)$, and the maximum system size on the interval is $N_0 M(x)$. Because E' is bitonic, the number of nodes that enter is $N_{\max} - N_0 = N_0(M(x) - 1)$, and the number of nodes that leave is $N_{\max} - N_f = N_0(M(x) - F(x, y))$. Finally, the crash assumption tells us that at most $\Delta N_f = \Delta F(x, y)$ nodes can be crashed at the end of the interval. □

2.3 Applying the Parametrization Method

The study of Bitonic sequences has granted us a powerful tool to analyze the behavior of the algorithm. Now, we turn our attention towards the proof of correctness found in [2]. We will reproduce the results involving the four constraints and show that they hold under less restrictive conditions. First, we state a lemma that will be used to simplify several mathematical expressions.

Lemma 6. *Let* $f(x) = \frac{ax+b}{cx+d}$ *be a function on a closed interval* $[\alpha, \beta]$ *with* $-d/c \notin [\alpha, \beta]$. *If* $ad - bc \geq 0$ *then the maximum value for* f *is attained at* $x = \beta$ *and the minimum at* $x = \alpha$. *If* $ad - bc \leq 0$ *then the maximum value for* f *is attained at* $x = \alpha$ *and the minimum at* $x = \beta$.

Proof. Since $-d/c \notin [\alpha, \beta]$, f is differentiable on $[\alpha, \beta]$. From the definition of f we have that:

$$f'(x) = \frac{a(cx+d) - (ax+b)c}{(cx+d)^2} = \frac{ad - bc}{(cx+d)^2}$$

This means that if $ad - bc \geq 0$ then $f'(x) \geq 0$, so f would be increasing, thus the maximum will be attained at $x = \beta$ and the minimum at $x = \alpha$. The case when $ad - bc \leq 0$ is similar but with f being a decreasing function. □

The following corollary follows directly from the previous lemma.

Corollary 1. *Let* $f(x) = \frac{ax}{cx+d}$ *be a function on a closed interval* $[\alpha, \beta]$ *with* $-d/c \notin [\alpha, \beta]$. *If* $ad \geq 0$ *then the maximum value for* f *is attained at* $x = \beta$ *and the minimum at* $x = \alpha$. *If* $ad \leq 0$ *then the maximum value for* f *is attained at* $x = \alpha$ *and the minimum at* $x = \beta$.

2.3.1 Improvements for Constraint A

Constraint A has two roles in the proof of correctness. The first one is to ensure that at least one node survives an interval of length $3D$ (Lemma 3 in [2]). The second one is to ensure that when a node joins the system, it must receive an enter-echo response from a node active during $[\max\{0, t' - 2D\}, t' + D]$ where t' is the time when the node received its first enter-echo from a joined node (Lemma 5 in [2]). We will focus on the second application and show that the resulting bound is sufficient to prove the first application.

The proof of the second application involves a node p who receives a response from a node q sent at time t' and received at time t'' that contains all active membership events up to time $\max\{0, t' - 2D\}$. We will divide the interval $[\max\{0, t' - 2D\}, t' + D]$ into three subintervals. The first one is $[\max\{0, t' - 2D\}, t']$, the second one is $[t', t'']$, and the third one is $[t'', t' + D]$. Let N_0 be the size of the system at time $\max\{0, t' - 2D\}$, let e_i be the number of nodes that enter during the ith subinterval, and l_i be the number of nodes that leave during the ith subinterval. Finally, let c be the number of nodes that are crashed at time $t' + D$.

Our objective is to show that at least one node that replied to p is active during $[\max\{0, t' - 2D\}, t' + D]$. To do so, we first calculate the minimum number of responses to p that come from nodes active at t' and then subtract the maximum number of nodes that could leave or crash during $[\max\{0, t' - 2D\}, t' + D]$.

The minimum value for the join threshold is based on p's Present set at time t''. Since p is aware of all active membership events for $[0, \max\{0, t' - 2D\}]$, its Present set contains at least all the nodes in N_0 minus those who crashed while sending their enter message, and those that left between $\max\{0, t' - 2D\}$ and t''. Thus, the join threshold is at least $\gamma(N_0 - l_1 - l_2 - c)$.

Next, we subtract the number of nodes not in N_0 that could reply to the message, which are those who join during $[\max\{0, t' - 2D\}, t' + D]$. This leaves us with at least $\gamma(N_0 - l - c) - e_1 - e_2$ responses that come from nodes active at $\max\{0, t' - 2D\}$.

Finally, we subtract the nodes that could leave or crash during $[\max\{0, t' - 2D\}, t' + D]$. This number is at most $l + c$. Therefore the number of nodes that reply to p's message and are active during $[\max\{0, t' - 2D\}, t' + D]$ is at least $\gamma(N_0 - l - c) - e - l - c$. Since we want this quantity to be at least one, we must satisfy the following inequality:

$$\gamma(N_0 - l - c) - e - l - c \geq 1 \qquad \text{(Ineq. 1)}$$

Solving for γ gives:

$$\gamma \geq \frac{1 + e + l + c}{N_0 - l - c}$$

Our goal is to find a number K such that $K \geq$ RHS for any event sequence, then letting $\gamma \geq K$ will ensure that Ineq. 1 holds. We use Theorem 1 to bound the RHS by

$$
\begin{aligned}
\text{RHS} &\leq \sup_{\substack{x\in[0,3]\\ y\in[0,3-x]}} \frac{1+N_0[(M(x)-1)+(M(x)-F(x,y))+\Delta F(x,y)]}{N_0[1-(M(x)-F(x,y))-\Delta F(x,y)]} \\
&= \sup_{\substack{x\in[0,3]\\ y\in[0,3-x]}} \frac{N_0^{-1}+2M(x)-(1-\Delta)F(x,y)-1}{1-M(x)+(1-\Delta)F(x,y))} \\
&= \sup_{x\in[0,3]} \frac{N_0^{-1}+2M(x)-(1-\Delta)F(x,3-x)-1}{1-M(x)+(1-\Delta)F(x,3-x))} \quad \text{Because } F(x,y) \text{ is decreasing for } y \\
&= \sup_{x\in[0,3]} \frac{N_0^{-1}+2M(x)-(1-\Delta)F(x,3-x)-1}{1-M(x)+(1-\Delta)F(x,3-x))} \\
&\leq \sup_{x\in[0,3]} \frac{N_{\min}^{-1}+2M(x)-(1-\Delta)F(x,3-x)-1}{1-M(x)+(1-\Delta)F(x,3-x))}
\end{aligned}
$$

To show that this bound implies that the number of nodes that do not crash or leave during an interval of length $3D$ is at least 1, we go back to Ineq. 1. Since $\gamma < 1$ and $e \geq 0$, we have that:

$$
\begin{aligned}
&\gamma(N_0 - l - c) - e - l - c \geq 1 \\
&\gamma N_0 - e - (l+c)(1+\gamma) \geq 1 \\
&\gamma N_0 - (l+c) \geq 1
\end{aligned}
$$

Which proves the claim. Therefore, letting

$$
\gamma \leq \sup_{x\in[0,3]} \frac{N_{\min}^{-1}+2M(x)-(1-\Delta)F(x,3-x)-1}{1-M(x)+(1-\Delta)F(x,3-x))}
$$

is enough to prove the lemmas where Constraint A is applied.

2.3.2 Improvements for Constraint B

The role of Constraint B is to ensure that there are enough joined nodes to answer the enter message of a node p^* that enters at some time $t_e > 2D$ (Theorem 3 in [2]). The first joined node that p hears from, which we will call q, receives the enter message at time t', and p receives q's

reply at time t''.

Let N_0 be the size of the system at time $t' - 2D$, let e, l be the number of nodes that enter and leave respectively during the interval $[t' - 2D, t' + D]$ and let c be the number of nodes that crash before $t' + D$. We can obtain an upper bound on the number of responses that p expects by taking $\gamma(N_0 + e)$ because q knows about all active membership events prior to $t' - 2D$ (due to Lemma 1) and $t'' < t' + D$ so p's Present set cannot contain more than $N_0 + e$ nodes. There are $N_0 - l - c$ nodes that are active during $[t' - 2D, t' + D]$. Any node that is active during $[t' - 2D, t' + D]$ will also be active during $[t_e, t_e + D]$ and hence, it is guaranteed to reply to p's message. This means that if the following inequality holds for any valid event sequence, then p is guaranteed to receive enough responses.

$$\gamma(N_0 + e) \leq N_0 - l - c \qquad \text{(Ineq. 2)}$$

Solving for γ gives:

$$\gamma \leq \frac{N_0 - l - c}{N_0 + e}$$

Our goal is to find a number K such that $K \leq$ RHS for any event sequence, then letting $\gamma \leq K$ will ensure that Ineq. 2 holds, We use Theorem 1 to bound the RHS by

$$\text{RHS} \geq \inf_{\substack{x\in[0,3] \\ y\in[0,3-x]}} \frac{1 - (M(x) - F(x,y)) - \Delta F(x,y)}{M(x)} = \inf_{\substack{x\in[0,3] \\ y\in[0,3-x]}} \frac{1 - M(x) + (1-\Delta)F(x,y)}{M(x)}$$

$$= \inf_{x\in[0,3]} \frac{1 - M(x) + (1-\Delta)F(x,3-x)}{M(x)} \quad \text{Because } F(x,y) \text{ is decreasing for } y$$

Therefore, it is enough to let $\gamma \leq \inf_{x\in[0,3]} \frac{1-M(x)+(1-\Delta)F(x,3-x)}{M(x)}$ to ensure that Ineq. 2 holds, and thus replace the original Constraint B.

2.3.3 Improvements for Constraint C

In this case, we want to ensure that if a node p starts a phase at time t, there will be enough joined nodes to respond to its request (Theorem 4 in [2]).

Let N_0 be the size of the system at time $t' = \max\{0, t - 2D\}$, let e_1, l_1 be the number of

nodes that enter and leave respectively during $[t', t]$, let l_2 be the number of nodes that leave during $[t, t+D]$, and let c be the number of nodes that crash before $t+D$. The size of p's Members set is at most equal to the size of p's Present set. By Lemma reflem:joined-changes, we know that p is aware of all active membership events up to time t', so at time t, the maximum size of p's Present set is at time t is $N_0 + e_1$. Hence, p expects at most $\gamma(N_0 + e_1)$ responses.

There are $N_0 - l_1 - l_2 - c$ nodes that are active during $[t', t+D]$. If $t - t' = 2D$ then by Lemma 2, they will join by time t. If $t - t' < 2D$ then $t' = 0$, and by assumption all the initial nodes have already joined. Additionally, every node that is active during $[t', t+D]$ will be active during $[t, t+D]$. This ensures that they will respond to p's message. This means that the following inequality for β guarantees that p will receive enough responses

$$\beta(N_0 + e_1) \leq N_0 - l_1 - l_2 - c \qquad \text{(Ineq. 3)}$$

Solving for β gives

$$\beta \leq \frac{N_0 - l_1 - l_2 - c}{N_0 + e_1}$$

Our goal is the same as in Constraint B: to find a number K such that $K \leq$ RHS for any event sequence, then letting $\beta \leq K$ will ensure that Ineq. 3 holds. We use Theorem 1 on $[t', t]$ and $[t, t+D]$ separately.

$$\begin{aligned}
\text{RHS} &\geq \inf_{\substack{x_1\in[0,2];y_1\in[0,2-x]\\ x_2\in[0,1];y_2\in[0,1-x_2]}} \frac{1-(M(x_1)-F(x_1,y_1))-F(x_1,y_1)(M(x_2)-F(x_2,y_2))-\Delta F(x_1,y_1)F(x_2,y_2)}{M(x_1)} \\
&= \inf_{\substack{x_1\in[0,2];y_1\in[0,2-x]\\ x_2\in[0,1];y_2\in[0,1-x_2]}} \frac{1-M(x_1)+F(x_1,y_1)(1-M(x_2)+(1-\Delta)F(x_2,y_2))}{M(x_1)} \\
&= \inf_{\substack{x_1\in[0,2];y_1\in[0,2-x]\\ x_2\in[0,1]}} \frac{1-M(x_1)+F(x_1,y_1)(1-M(x_2)+(1-\Delta)F(x_2,1-x_2))}{M(x_1)} \\
&= \inf_{\substack{x_1\in[0,2]\\ x_2\in[0,1]}} \frac{1-M(x_1)+F(x_1,2-x_1)(1-M(x_2)+(1-\Delta)F(x_2,1-x_2))}{M(x_1)}
\end{aligned}$$

Resulting in the following bound for β

$$\beta \leq \inf_{\substack{x_1 \in [0,2] \\ x_2 \in [0,1]}} \frac{1 - M(x_1) + F(x_1, 2 - x_1)(1 - M(x_2) + (1 - \Delta)F(x_2, 1 - x_2))}{M(x_1)}$$

2.3.4 Improvements for Constraint D

The role of Constraint D is to ensure that if a store operation begins at time t_s and finishes before some collect operation that starts at time t_c such that $t_c - t_s < 2D$, then the store operation will reflect the store operation on its output (Lemmas 9 and 10 in [2]). In this proof, we will assume that $\Delta \leq (1 + \alpha)/2$. It has been proven that it is impossible to have $\Delta > (1 + \alpha)/2$ for a read-write register. Given the similarities between the specification of a read-write register and the specification of the store-collect object, this is likely the case as well for store-collect objects.

Let Q_s be the set of nodes that replied to the store operation and stay active until $t_s + 3D$, let Q_c be the set of nodes that replied to the collect operation, and let J be the set of nodes active at some time during $[t_c, t_c + D]$. To show that the collect reflects the store operation, we will show that one of the nodes that responded to the collect also responded to the store. In other words $Q_s \cap Q_c \neq \emptyset$, which we will prove by showing that $|Q_s| + |Q_c| > |J|$ since Q_s and Q_c are both subsets of J.

Let N_0 be the number of nodes at time $\max\{0, t_s - 2D\}$, l_1, e_1 be the number of nodes that leave or enter during $[\max\{0, t_s - 2D\}, t_s]$, l_2, e_2 be the number of nodes that leave or enter on $[t_s, t_s + 2D]$, l_3, e_3 be the number of nodes that leave or enter on $[t_s + 2D, t_s + 3D]$, c_1 be the number of nodes that crash before t_s, c_2 be the number of nodes that crash before $t_s + 2D$, and c_3 the number of nodes that crash before $t_s + 3D$.

We first calculate a lower bound for $|Q_s|$. By a similar argument to the one used in Constraint A, at least $\beta(N_0 - l_1 - c_1)$ nodes reply to the store message, hence at least $\beta(N_0 - l_1 - c_1) - l_2 - l_3 - (c_3 - c_1)$ nodes reply to the store and remain active until $t_s + 3D$. Next, we give a lower bound for $|Q_c|$. Since $t_c \leq t_s + 2D$, at least $\beta(N_0 - l_1 - l_2 - c_2)$ nodes reply to the store operation.

Finally, we give an upper bound for $|J|$. If a node is active during $[t_c, t_c + D]$ then it must be active at time t_s, or enter at some point in $[t_s, t_s + 3D]$ because $t_s \leq t_c \leq t_s + 2D$. Hence

$|J| \leq N_0 + e_1 - l_1 - c_1 + e_2 + e_3$. Putting it all together gives

$$\beta(N_0 - l_1 - c_1) + \beta(N_0 - l_1 - l_2 - c_2) - l_2 - l_3 - (c_3 - c_1) > N_0 + e_1 - l_1 - c_1 + e_2 + e_3 \quad \text{(Ineq. 4)}$$

Solving for β gives:

$$\beta > \frac{N_0 + e_1 - l_1 - c_1 + e_2 + l_2 + l_3 + e_3 + (c_3 - c_1)}{2(N_0 - l_1) - c_1 - l_2 - c_2}$$

We will find a number K such that $K >$ RHS for any event sequence, then letting $\beta > K$ will ensure that Ineq. 4 holds. Since the third interval has length D, we use the churn assumption to bound above the quantity $l_3 + e_3$.

$$\begin{aligned}
\text{RHS} &= \frac{N_0 + e_1 - l_1 - c_1 + e_2 + l_2 + (l_3 + e_3) + (c_3 - c_1)}{2(N_0 - l_1) - c_1 - l_2 - c_2} \\
&\leq \frac{N_0 + e_1 - l_1 + e_2 + l_2 + \alpha(N_0 + e_1 - l_1 + e_2 - l_2) + c_3 - 2c_1}{2(N_0 - l_1) - c_1 - l_2 - c_2} \quad \text{By the churn assumption} \\
&= \frac{(1+\alpha)(N_0 + e_1 - l_1) + (1+\alpha)e_2 + (1-\alpha)l_2 + c_3 - 2c_1}{2(N_0 - l_1) - c_1 - l_2 - c_2}
\end{aligned}$$

Now, we can apply Theorem 1 with one extra addition. Since Theorem 1 only gives us a maximum value for the number of nodes that can crash, we will add a new parameter $z \in [0, 1]$ that we multiply to this value to represent the exact number of nodes that crash for the case of c_1. Given

the length of the formula, we will abbreviate $F(x_i, y_i)$ as F_i and $M(x_i)$ as M_i for integers i.

$$\leq \sup_{\substack{x_1\in[0,2];y_1\in[0,2-x]\\ x_2\in[0,2];y_2\in[0,2-x_2]\\ x_3\in[0,1];y_3\in[0,1-x_2]\\ z\in[0,1]}} \frac{(1+\alpha)F_1 + F_1((1+\alpha)(M_2-1)+(1-\alpha)(M_2-F_2)) + \Delta F_1F_2F_3 - 2z\Delta F_1}{2(1-(M_1-F_1)) - \Delta zF_1 - F_1(M_2-F_2) - \Delta F_1F_2}$$

$$= \sup_{\substack{x_1\in[0,2];y_1\in[0,2-x]\\ x_2\in[0,2];y_2\in[0,2-x_2]\\ x_3\in[0,1];y_3\in[0,1-x_2]\\ z\in[0,1]}} \frac{F_1[1+\alpha+(1+\alpha)(M_2-1)+(1-\alpha)(M_2-F_2)+\Delta F_2F_3-2z\Delta]}{2(1-M_1)+F_1[2-z\Delta-M_2+(1-\Delta)F_2]}$$

$$= \sup_{\substack{x_1\in[0,2];y_1\in[0,2-x]\\ x_2\in[0,2];y_2\in[0,2-x_2]\\ z\in[0,1]}} \frac{F_1[(1+\alpha)M_2+(1-\alpha)(M_2-F_2)-2z\Delta+\Delta F_2(1+\alpha)]}{2(1-M_1)+F_1[2-z\Delta-M_2+(1-\Delta)F_2]} \quad \text{Because } F_3 \leq (1+\alpha)$$

Next, we apply Corollary 1 with

$$x = F_1$$

$$a = (1+\alpha)M_2 + (1-\alpha)(M_2-F_2) - 2z\Delta + \Delta F_2(1+\alpha)$$

$$c = 2 - z\Delta - M_2 + (1-\Delta)F_2$$

$$d = 2(1-M_1)$$

Our assumption that $\Delta < (1+\alpha)/2$ ensures that a is positive because the only negative term is $-2z\Delta$ and $(1+\alpha)M_2 \geq 1+\alpha > 2\Delta \geq 2z\Delta$. Since d is negative, $a * d \leq 0$, and Corollary 1 tells us that the function is maximized when F_1 is minimized (assuming all the other terms are constant). Since the term M_1 is also dependent on the value of x_1, we can only safely minimize F_1 by maximizing y_1, in other words, $y_1 = 2-x_1$, and hence $F_1 = F(x_1, 2-x_1) = M(x_1)(1-\alpha)^{2-x_1}$.

We then we get:

$$= \sup_{\substack{x_1\in[0,2];\\ x_2\in[0,2];y_2\in[0,2-x_2]\\ z\in[0,1]}} \frac{M_1(1-\alpha)^{2-x_1}[(1+\alpha)M_2+(1-\alpha)(M_2-F_2)-2z\Delta+\Delta F_2(1+\alpha)]}{2(1-M_1)+M_1(1-\alpha)^{2-x_1}[2-z\Delta-M_2+(1-\Delta)F_2]}$$

$$= \sup_{\substack{x_1\in[0,2];\\ x_2\in[0,2];y_2\in[0,2-x_2]\\ z\in[0,1]}} \frac{M_1(1-\alpha)^{2-x_1}[(1+\alpha)M_2+(1-\alpha)(M_2-F_2)-2z\Delta+\Delta F_2(1+\alpha)]}{2+M_1[-2+(1-\alpha)^{2-x_1}[2-z\Delta-M_2+(1-\Delta)F_2]]}$$

We once again apply Corollary 1, this time with

$$x = M_1$$

$$a = (1-\alpha)^{2-x_1}[(1+\alpha)M_2+(1-\alpha)(M_2-F_2)-2z\Delta+\Delta F_2(1+\alpha)]$$

$$c = -2+(1-\alpha)^{2-x_1}[2-z\Delta-M_2+(1-\Delta)F_2]$$

$$d = 2$$

Since a and d are positive, the expression is maximized when M_1 is maximized. Hence $x_1 = 2$, and thus $M_1 = (1+\alpha)^2$.We now have

$$= \sup_{\substack{x_2\in[0,2];y_2\in[0,2-x_2]\\ z\in[0,1]}} \frac{(1+\alpha)^2[(1+\alpha)M_2+(1-\alpha)(M_2-F_2)-2z\Delta+\Delta F_2(1+\alpha)]}{2+(1+\alpha)^2[-2+2-z\Delta-M_2+(1-\Delta)F_2]}$$

$$= \sup_{\substack{x_2\in[0,2];y_2\in[0,2-x_2]\\ z\in[0,1]}} \frac{(1+\alpha)^2[(1+\alpha)M_2+(1-\alpha)(M_2-F_2)-2z\Delta+\Delta F_2(1+\alpha)]}{2+(1+\alpha)^2[-z\Delta-M_2+(1-\Delta)F_2]}$$

$$= \sup_{\substack{x_2\in[0,2];y_2\in[0,2-x_2]\\ z\in[0,1]}} \frac{(1+\alpha)^2[2M_2+F_2[\Delta(1+\alpha)-(1-\alpha)]-2z\Delta]}{2+(1+\alpha)^2[-z\Delta-M_2+(1-\Delta)F_2]}$$

$$= \sup_{\substack{x_2\in[0,2];y_2\in[0,2-x_2]\\ z\in[0,1]}} \frac{2M_2+F_2[\Delta(1+\alpha)-(1-\alpha)]-2z\Delta}{2(1+\alpha)^{-2}-z\Delta-M_2+(1-\Delta)F_2}$$

Which is our final value for K. Hence, it suffices to set

$$\beta > \sup_{\substack{x_2 \in [0,2]; y_2 \in [0, 2-x_2] \\ z \in [0,1]}} \frac{2M_2 + F_2[\Delta(1+\alpha) - (1-\alpha)] - 2z\Delta}{2(1+\alpha)^{-2} - z\Delta - M_2 + (1-\Delta)F_2}$$

to ensure that Ineq. 4 holds.

2.4 Discussion of Results

The new replacements for the constraints result in significant improvements in all cases, except for constraint A, where the new formula is only better than the original when α is large. To compare the two sets of constraints, we calculated the maximum value of Δ that is permissible for a given value of α. Constraints A and B provide a lower and upper bound for γ, while Constraints C and D provide an upper and lower bound for β. We can find the maximum permissible value by using the bisection method to find the largest number such that the constraint intervals for γ and β are both non-empty. The figure below shows a comparison of three different constraint sets with $N_{\min} = 2$: the original constraints from [2], the new constraints, and the new constraints but with the old version of constraint A.

Part of the reason why constraint A was more difficult to improve than the others is that the method we used requires us to set a reference point and divide the time interval into blocks of length D, which is not able to properly manage events that do not occur at an integer multiple of D with respect to the reference point.

Future work should focus on formulating more careful versions of the churn inequalities, with a special focus on constraint A. Additionally, the parametrization might be modified to obtain better bounds, or to work under different model assumptions.

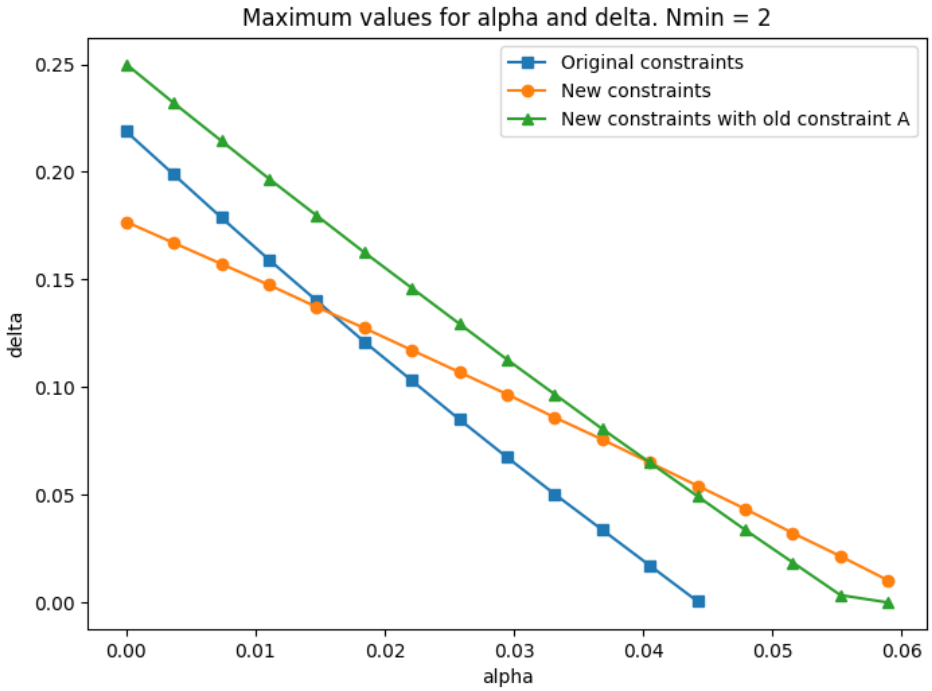

Figure 2.1: Comparison of the maximum values of α and Δ for different sets of constraints.

3. AN EFFICIENT ATOMIC SNAPSHOT IMPLEMENTATION

3.1 Background

Atomic snapshots [3] are distributed objects that are closely related to store-collect objects. They support two basic operations, update and scan, which are direct equivalents of the store and collect operations of store-collect. However, atomic snapshots guarantee that any execution must be linearizable.

The linearizability of the atomic snapshot object makes it a useful building block for more complex algorithms. Atomic snapshots have been used to implement algorithms for elections, mutual exclusion, consensus, collect, snapshot, and renaming on dynamic systems [7, 8, 9].

A simple implementation of atomic snapshots based on store-collect can be found in [2, 1]. By adapting optimizations from [10], in addition to new ideas from our own work, we can give a more efficient implementation of atomic snapshots using store-collect.

The two main improvements of this version of the algorithm over the version found in [2] are the removal of a collect operation inside the update function, and reducing the number of collects required for a store operation in the best case from 2 to 1. In order to remove the collect operation inside update, we modified the conditions of a *borrowed scan*. We argue that this change in the borrow system does not result in any significant penalty, if any, to the time to complete a scan operation.

3.2 Algorithm Description

Formally, an *atomic snapshot* provides two operations: SCAN(), which has no arguments and returns a snapshot view, and UPDATE(v), which takes a value v from some set Val_{AS} as an argument and returns ACK. Its sequential specification consists of all sequences of updates and scans where the snapshot view returned by a SCAN contains the value of the last UPDATE performed by each node p, if such an UPDATE exists. We do not include in the view the nodes who have performed no update at the time.

Unlike store-collect, an implementation of an atomic snapshot should be *linearizable* [4]. This means that for any execution α, there must exist an order for the operations in α that respects the real time ordering of non-overlapping operations and the sequential specification for an atomic snapshot.

For the scan function, the algorithm performs repeated collects until finding a pair of collects with the same values (not counting uninitialized entries). Such a scan is called a **direct scan**. Additionally, a scan may 'borrow' the view returned by another direct scan and return that view without having to wait for two equal collects. Such a scan is called a **borrowed scan**. An improvement upon the implementation found in [2] is that the algorithm can now use the result from its last collect, allowing the algorithm to finish a scan with only one collect in the best case.

For the update function, each update call executes a scan, which we call an **embedded scan**, to allow the node to synchronize, followed by a store that adds the new value into the system. We managed to reduce the amount of work done compare to the original implementation, which uses a collect in addition to the scan.

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

Our algorithm to implement an atomic snapshot uses an unnamed store-collect object. The values stored in the store-collect object contain additional information besides the stored elements. The values we store are taken from the following set ($\mathcal{P}$ indicates the power set of its argument):

$$Val_{SC} = Val_{AS} \times \mathbb{N} \times \mathbb{N} \times \mathcal{P}(\Pi \times Val_{AS}) \times \mathcal{P}(\Pi \times \mathbb{N}) \times \mathbb{Z}_2$$

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

Algorithm 1 Atomic snapshot: code for node p.

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

When $\text{SCAN}_p()$ occurs:
1: *ssqno*++
2: $\text{STORE}_p(\langle -, -, ssqno, -, -, - \rangle)$
3: **if** (*currV* $== \emptyset$) **then**
4: *currV* $= \text{COLLECT}_p()$;
5: **while** *true* **do**
6: *oldV* $=$ *currV*
7: *currV* $= \text{COLLECT}_p()$
8: **if** (r(*currV*).*usqno*$= r$(*oldV*).*usqno*) **then**
9: *isDirect* = **true**;
10: return r(*currV*).*val* // direct scan
11: **if** $\exists q$ s.t. $\langle p, ssqno \rangle \in$ *currV*(q).*scounts*
12: **and** *currV*(q).*isDirect* $==$ *true* **then**
13: *isDirect* = **false**;
return *currV*(q).*sview* // borrowed scan

When $\text{UPDATE}_p(v)$ occurs:
14: *sview* $= \text{SCAN}_p()$ // embedded scan
15: *scounts* $=$ *currV*.*ssqno*
16: *val* $= v$
17: *usqno*++
18: $\text{STORE}_p(\langle$ *val*, *usqno*, -, *sview*, *scounts*, *isDirect*$\rangle)$
19: return ACK

fifth component (*scounts*) holds a set of counts of how many scans have been done by the other nodes, as observed by p. The sixth component (*isDirect*) holds a boolean that is true if and only if the embedded scan in the last update performed by this node was a direct scan. We denote the components of an element v in Val_{SC} respectively as: v.*val*, v.*usqno*, v.*ssqno*, v.*sview*, v.*scounts*, and v.*isDirect*.

For any store-collect view V, we define V.*comp* as the result of replacing each pair $\langle p, v \rangle$ with $\langle p, v.comp \rangle$ where *comp* is one of the components of v. For any view V and node p, we define $V(p)$ as the second component of the pair whose first component is p ($\perp$ if there is no such pair), and $r(V)$ as the subset of V containing only the pairs where the value is not null.

To execute a SCAN, Algorithm 1 increments the scan sequence number (*ssqno*) (Line 1) and stores it in the embedded store-collect object with all the other components unchanged (which we denote with the $-$ notation). Then, if no view is stored in the variable *currV*, a view is collected (Line 3). In a while loop, the last collected view is saved and a new view is collected (Line 7). If

the two most recently collected views have the same non-null entries (Line 8), the variable *isDirect* is set to **true** and the latest collected view is returned (Lines 8 and 9). 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 the node's *ssqno* in a direct scan by checking the *scounts* and *isDirect* components (Line 11). If this condition holds, the variable *isDirect* is set to **false** (Line 12), and the snapshot view of q is returned (Line 13); we call this a *borrowed scan*.

An UPDATE first saves the value of an embedded scan in a local variable *sview* (Line 14). Then it obtains all scan sequence numbers from the last collected view (which was updated by the aforementioned scan), and assigns them to a local variable *scounts* (Line 15).

Then it sets its *val* variable to the argument value and increments its update sequence number (Lines 16 and 17). Finally the new value, update sequence number, collected view, and set of scan sequence numbers are stored; the node's own scan sequence number is unchanged (Line 18).

3.3 Proof of Correctness

This proof is based on the proof from [2] with some modifications. Most of the steps in the proof are very similar to the original, except that Lemma 8 (Lemma 12 in the original) and the proof of termination have been modified to match the new conditions for borrowing scans.

To prove linearizability, we consider an execution and define an ordering of all the completed scans and all the updates whose store on Line 18 takes effect. The ordering must preserve the time order of non-overlapping operations and respect the sequential specification for the atomic snapshot object.

We start by defining the following order for snapshot views: let W_1 be the snapshot view returned by a direct scan based on the collect view V_1 (cf. Line 10) and W_2 be the snapshot view returned by a direct scan based on the collect view V_2 (cf. Line 10). We define $W_1 \preceq W_2$ if for every $\langle p, v\rangle \in W_1$, there exists $\langle p, v'\rangle \in W_2$ where the *usqno* associated with v in V_1 is less than or equal to the *usqno* associated with v' in V_2.

Lemma 7. *If a direct scan by node p returns W_1 and a direct scan by node q returns W_2, then either $W_1 \preceq W_2$ or $W_2 \preceq W_1$.*

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

Case 1: cop_p^2 starts before cop_q^2 starts. This means that cop_p^1 finishes before cop_q^2 starts. Let $\langle p, v\rangle \in W_1$; then $\langle p, v\rangle \in V_1'$ with $v.usqno > 0$. By the regularity of store-collect, $V_1' \preceq V_2$. Thus, there is an entry $\langle p, v'\rangle \in V_2$ such that $v = v'$ or a more recent value. Since the *usqno* variable takes on increasing values, $0 < v.usqno \leq v'.usqno$. Thus, $\langle p, v'\rangle \in W_2$ and therefore $W_1 \preceq W_2$.

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

If a direct scan returning snapshot view W_1, obtained from collect view V_1, completes before another direct scan starts, which returns snapshot view W_2, obtained from collect view V_2, then the regularity of store-collect ensures $V_1 \preceq V_2$, and thus $W_1 \preceq W_2$. Hence, the ordering $\preceq$ preserves the real-time order of non-overlapping direct scans.

We now extend the rule $\preceq$ for ordering scans to also include borrowed scans. Consider all scans that borrow from a given direct scan *sop*. Place the borrowed scans immediately after *sop* in the order in which they complete. The following lemma implies that any borrowed scan overlaps with the scan it borrows from. Hence, the real-time order of any two non-overlapping scans, at least one of which is borrowed, is preserved by our ordering rule since direct scans have already been shown to be ordered properly.

Lemma 8. *If a scan sop_p by node p borrows from a direct scan sop_q by node q, then sop_q completes before sop_p completes and sop_p starts before sop_q completes. Hence the two scans overlap.*

Proof. Let uop_q be the update in which sop_q is embedded. Since the collect of p must wait until q stores the updated *ssqno* for p (Line 18), sop_q completes before sop_p completes.

Furthermore, since sop_p borrows the snapshot view of sop_q, its *ssqno* appears in q's *scounts* variable and hence, p's store (Line 2) starts before q's store (Line 18). Hence, sop_q starts before sop_p completes. □

Finally, we consider all updates in the order their stores (Line 18) start. Place each update, say *uop* by node p with argument v, immediately before the first scan whose returned view includes $\langle p, v\rangle$, or a later value. Note that every scan after this scan contains $\langle p, v\rangle$, or a later value. If there is no such scan, then place *uop* at the end of the ordering. It is clear that this rule for placing updates preserves the sequential specification of atomic snapshots. Additionally, it preserves the real-time order between non-overlapping updates and scans because if a scan completes before an update starts, then the view returned by the scan 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. The next lemma deals with non-overlapping updates.

Lemma 9. *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 collect, cop_1 of node p, and let cop_2 be the collect preceding cop_1.

We now show that $V = r(W).val$. If $sop' = sop$, then $V = r(W).val$ by Line 10, since *sop* is a direct scan. Otherwise, $V = r(W).val$ because $r(W).val$ is returned by the scan *sop* borrows from, assigned to *sview*, and then stored (cf. Lines 14 and 18) and returned by *sop* as V.

Since V includes the value of uop_p, so does W. It follows that the last two stores of uop_p start before cop_2 completes and thus before cop_1 starts. Since uop_q precedes uop_p, the store of uop_q at Line 18 completes before either store of uop_p starts. Thus the store of uop_q completes before cop_1 starts, and by the store-collect property, the view W returned by cop_1 must include the value of uop_q or a later update by q. Since $V = r(W).val$, the same is true for V. □

Consider an update uop_p, by node p, that follows an update uop_q, by node q, in the execution. If uop_p is placed at the end of the (current) ordering because there is no scan that observes its value or a later update by p, then it is ordered after uop_q. If uop_p is placed before a scan, then the same

must be true of uop_q. By construction, the next scan after uop_p in the ordering, call it *sop*, returns view V with $V(p)$ equal to the value of uop_p or a later update by p. By Lemma 9, $V(q)$ 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. Assume for contradiction that there is a scan operation sop_q by node q that does not terminate. Let t be the time the initial store operation of the scan (Line 1) finishes. At this time there are at most $N(t)$ updates pending. Notice that the only operations that can affect the update sequence number are updates that terminate since the *usqno* is modified at the end of the update operation (Line 18).

Let t' be the time all terminating updates pending at time t finish. If there are no terminating updates after time t', then the collect will terminate after the next pair of collects by q. Otherwise, there must exist a node p that successfully performs an update. Let sop_p be the embedded scan of such update. If sop_p is direct, then q can borrow its scan because sop_p started after time t. If sop_p is a borrowed scan, the update it borrows from must have started after time t, and hence be borrowed by q, because if it started before t, then it must have finished before time t', but sop_p starts after t' so the two don't overlap, contradicting Lemma 8. Hence sop_q terminates. Putting the pieces together, we have:

Theorem 2. *Algorithm 1 is a linearizable implementation of an atomic snapshot object.*

3.4 Discussion of Results

This improved algorithm retains the simplicity of the original while providing a more efficient implementation. Our proof of termination does not provide an upper bound in the number of communication rounds in a SCAN or an UPDATE operation, however we presume that it is at most linear in the number of nodes present in the system when the operation starts based on the original proof of termination. It might be possible to adapt the original proof of termination to the new conditions for borrowing, and show that the bound is still linear in the number of nodes in the system at the start of the operation.

The store-collect object simplified the work required to implement the atomic snapshot object since we can now use shared memory and ignore any aspects related to the churn of the system or crashing nodes. However, this layer of abstraction introduces redundant or unnecessary operations.

For example, each collect operation performs an embedded store to enure that any subsequent collects will return a view with more recent values for all nodes, but this property is not necessary for this algorithm (we only need sequencial consistency between stores and collects, not collects and collects).

Another example is that the variable *currV* is only updated when a node performs a scan, but *currV* could also use the the view of the embedded store-collect object at that time, which is more updated and thus increases the chances of performing a successful scan with only one call to collect.

A potential direction for future work is to improve the performance of this algorithm by removing the intermediary layer and work directly using messages.

4. CONCLUSION

In the first part of the thesis, we successfully developed and applied our new method for analyzing the behavior of a distributed system that experiences ongoing churn. This method provides a new way of reasoning about the proof of correctness of the algorithm in [1], leading to less restrictive constraints on the system parameters. To prove a certain property, we express this property using inequalities involving the churn events of the system; then, we parametrize those inequalities and apply mathematical optimization to find the desired constraint.

Future work may concentrate on developing an alternative method of parametrizing the churn of the system, which could lead to further improvements on the constraints without substantial modification of the setup. Alternatively, it may concentrate on using the same parametrization method, but with a more careful construction of the churn inequalities, particularly for constraint A, which was the only constraint that was not a total improvement compared to the original constraint.

In the second part of the thesis, we developed a more efficient implementation of the atomic snapshot object with the store-collect algorithm while retaining the simplicity of the algorithm found in [1]. Future work focusing on atomic snapshot objects may try to improve the efficiency of the presented algorithm by removing the layer of abstraction provided by store-collect. Alternatively, future work may concentrate on finding other algorithms beyond atomic snapshot that could be implemented using store-collect.

REFERENCES

[1] 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 *International Symposium on Stabilizing, Safety, and Security of Distributed Systems*. Springer, 2020, pp. 1–15.

[2] ——, "Store-collect in the presence of continuous churn with application to snapshots and lattice agreement," *arXiv preprint arXiv:2003.07787*, 2020.

[3] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic snapshots of shared memory," *J. ACM*, vol. 40, no. 4, pp. 873–890, 1993.

[4] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *Trans. Prog. Lang. Sys.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[5] G. L. Hu, "Average case analysis of a shared register emulation algorithm," B.S. Thesis, Texas A&M University, 2019.

[6] 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.*, vol. 30, no. 3, pp. 544–559, 2019. [Online]. Available: https://doi.org/10.1109/TPDS.2018.2867479

[7] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer, "Dynamic atomic storage without consensus," *J. ACM*, vol. 58, no. 2, pp. 7:1–7:32, 2011.

[8] E. Gafni, M. Merritt, and G. Taubenfeld, "The concurrency hierarchy, and algorithms for unbounded concurrency," in *PODC*, 2001, pp. 161–169.

[9] M. Merritt and G. Taubenfeld, "Computing with infinitely many processes," in *DISC*, 2000, pp. 164–178.

[10] C. Delporte-Gallet, H. Fauconnier, S. Rajsbaum, and M. Raynal, "Implementing snapshot objects on top of crash-prone asynchronous message-passing systems," *TPDS*, vol. 29, no. 9, pp. 2033–2045, 2018.

APPENDIX: Implementation of Store-Collect

Algorithm 2 CCC—Common code managing churn, for node p.

Local Variables:

LView: set of (node id, value, sequence number) triples, initially $\emptyset$ // local view

is_joined: Boolean, initially false // true iff p has joined the system

join_threshold: int, initially 0 // number of enter-echo messages needed for joining

join_counter: int, initially 0 // number of enter-echo messages received so far

Changes: set of *enter*(q), *leave*(q), and *join*(q) // active membership events known to p
initially $\{$*enter*$(q)|q \in S_0\} \cup \{$*join*$(q)|q \in S_0\}$ if $p \in S_0$, and $\emptyset$ otherwise

Derived Variable:

Present $= \{q \mid$ *enter(q)* $\in$ Changes$\wedge$ *leave(q)* $\notin$ *Changes*$\}$

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

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

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

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

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

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

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

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

Algorithm 3 CCC—Client code, for node p.

Local Variables:

optype: string, initially $\perp$ // indicates which type of operation (*collect* or *store*) is pending
tag: int, initially 0 // counter to identify currently pending operation by p
threshold: int, initially 0 // number of replies/acks needed for current phase
counter: int, initially 0 // number of replies/acks received so far for current phase
sqno: int, initially 0 // sequence number for values stored by p

Derived Variable:

Members $= \{q |$ *join(q)* $\in$ *Changes* $\wedge$ *leave*$(q) \notin$ *Changes*$\}$

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

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

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

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

Algorithm 4 CCC—Server code, for node p.

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

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

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