AVERAGE CASE ANALYSIS OF A SHARED REGISTER EMULATION ALGORITHM

An Undergraduate Research Scholars Thesis

by

GERALD HU

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

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Jennifer Welch

May 2019

Major: Computer Science and Engineering

TABLE OF CONTENTS

ABSTRACT

Average Case Analysis of a Shared Register Emulation Algorithm

Gerald Hu
Department of Computer Science and Engineering
Texas A&M University

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

Distributed algorithms are important for managing systems with multiple networked components, where each component operates independently but coordinates to achieve a common goal. Previous theoretical research has produced numerous distributed algorithms but primarily uses mathematical proofs to yield theoretical results, such as worst-case runtime complexity. However, less research has been done on how these algorithms behave in practice, such as average-case runtime complexity.

This paper will describe the empirical behavior of the distributed algorithm CCReg [1] in a realistic environment, using the language DistAlgo[2] to implement said algorithm. CCReg emulates a shared read/write register using a message-passing system. In particular, CCReg allows the underlying message-passing system to experience continuous changes to the set of components present, and tolerates crash failures of components.

When the rate of component change and the fraction of crash failures are bounded, CCReg is proven to work correctly. The original paper specifies bounds for both that are guaranteed to work, and gives proof for those bounds. However, these bounds are

restrictive and do not allow for much component change or many crash failures.

Thus the goal of our implementation is to determine if CCReg's theoretical bounds can be relaxed in practice. We focus on CCReg's safety and liveness conditions: the algorithm eventually terminates, and a consistency condition called linearizability is maintained. We use a general method developed by Gibbons and Korach [3] for determining if any ordering of operations satisfying linearizability exists.

We find that, for executions where operations are randomly invoked, the algorithm does not exhibit any adverse behaviors. Each execution we tested terminates in finite time and has an order of operations satisfying linearizability. We discuss these findings, as well as future approaches and methodology for testing the theoretical boundaries in practice.

DEDICATION

To everyone who helped me get this far, and to everyone who believed in me when I could not.

ACKNOWLEDGMENTS

First of all, I would like to acknowledge my faculty advisor Dr. Jennifer Welch, who has always been a positive influence on my life, and whose continuous generosity and kindness is inspiring and humbling. She has always been accommodating, and has always listened to my troubles, no matter how busy her own schedule was. Whenever we discussed issues in the projects, roadblocks in the implementations, points of confusion, or what steps to take next, I always came away inspired and ready to take on the project. She is a continual source of inspiration and support in my life. She sparked my interest in theoretical computer science, and I think it's because of her that I want to continue pursuing research in academia. I cannot thank her enough for giving me the opportunity to do research in her lab, under her mentorship.

I would also like to thank Saptaparni Kumar for being a source of guidance and support, through long nights of work and periods of self-doubt. She was always there to answer questions about the churn algorithm, or about other topics in distributed computing, or about the merits of research and academia in general. In particular her clarifications about the sliding window and her sharp ideas about adversarial executions were quite helpful, even if I could not execute on all of them. Her energy and drive for research was infectious, and I found myself driven as well.

Last, I would like to thank Michael Earl and Jordan Lamkin. It has been a privilege to know you, both inside of class and outside, talking about anything from computing to whatever. It has always been a pleasure to keep your company.

To everyone here, and more I did not mention: thank you so much for your support. I wish you the best in all of your future endeavors in computing, no matter what those may be.

NOMENCLATURE

CCReg	Continuous Churn Register algorithm
α	Churn fraction
Δ	Failure fraction
γ	Join bound
β	Read/write bound

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

1. INTRODUCTION AND LITERATURE REVIEW

Distributed computer systems are an important part of modern computing. These systems are composed of distributed devices (as opposed to one centralized device), working together to achieve some common goal. Distributed systems can take many forms, such as networks of embedded sensors, geographically distributed database storage, and users of the internet accessing a shared document. They can be composed of static devices such as servers, or mobile devices such as smart phones, and combinations thereof.

A large portion of distributed computing research focuses on theoretical analysis, using mathematical and formal methods to specify an algorithm's behavior and prove theoretical boundaries. This is often beneficial as careful specification allows algorithms to be abstracted from hardware implementations, and allows the results to be relevant to multiple situations. Theoretical analysis also yields useful conclusions, such as impossibility of certain results.

Theoretical analysis methods are powerful, but have some blind spots. In particular, analytical methods can produce upper bounds and lower bounds, but moving those closer together to form a tightly-defined bound is difficult. Sometimes, these bounds can be too pessimistic or overly restrictive. In addition, theoretical analysis can be challenged by uncertainty and nondeterminism in the model. Asynchronous messages can take arbitrary amounts of time to reach their destination, arbitrary nodes can crash or fail. With so much uncertainty, many analyses can only assume a worst case scenario every time.

This motivates the primary direction of this research: exploring the gap between the theoretical bounds, drawing conclusions from empirical methods and real-world behavior. We have selected a distributed algorithm for study, and have implemented this algorithm in the programming language DistAlgo[2]. DistAlgo is a high-level language which allows expression and implementation of distributed algorithms. Its high-level expressiveness allows for human readability while still being implementable and unambiguous.

The algorithm selected is CCReg [1], which simulates a shared read/write register on top of a message-passing system with churn. We shall explain what a shared read/write register is, why we wish to simulate it with a message-passing system, the conditions we would like this simulation to have, what churn is, and why churn makes these conditions harder.

There are two primary mechanisms for communication between systems: shared memory objects, and message-passing systems. Shared memory can be thought of as a single central object that multiple devices can access concurrently. Message-passing systems are those where devices concurrently communicate in a point-to-point way. Designing algorithms for message-passing systems can be difficult, especially when devices in those systems are subject to failures, and one may want to work with a shared-memory model instead. However, in some contexts, a "true" shared-memory model can be impossible to implement. This motivates the idea of *emulating* a shared memory model on top of message passing. An emulation provides the illusion of having shared memory, so a user or designer may run an algorithm designed for shared memory where a shared memory system does not truly exist. These emulated models replicate the memory state across multiple devices, and accessing or changing the "shared" memory state requires inter-device communication. [4] An illustration of this concept can be seen in Figure 1.1.

We would like for every device to have a similar view of what happened to the (simulated) shared memory object. Even though operations on the shared memory object can happen concurrently, we would like some "view" of the system where all operations happen in some sequence, without any concurrency. This way, every device has the same "view" of the system. Each device can order the operations the same way, and every device agrees on what the state of the shared object should be. In this way, the shared

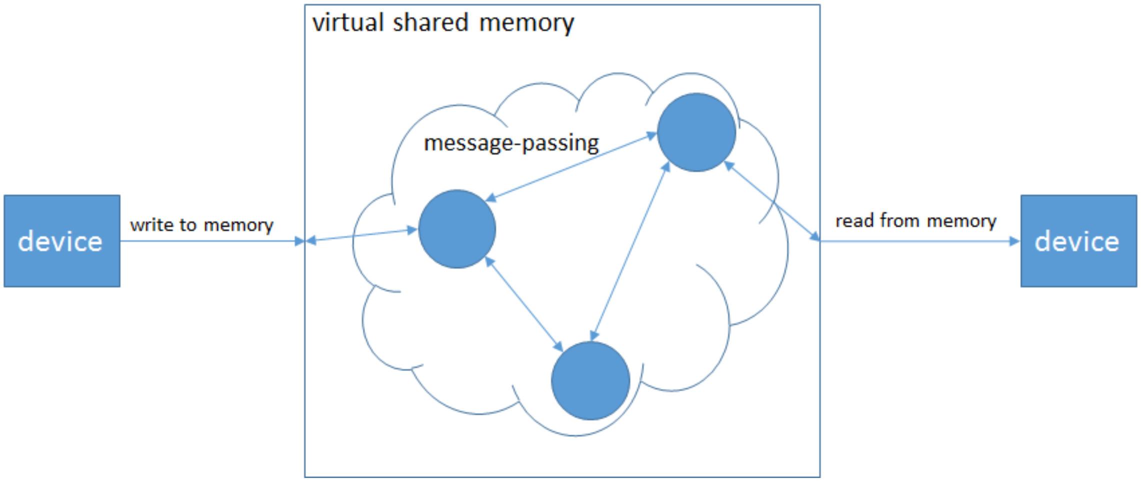

Figure 1.1: A graphic demonstrating shared memory emulations. Apps believe they are reading and writing to shared memory. However, the shared memory is actually being emulated by a message-passing system. The message passing system provides a consistent view of what's stored in the shared memory, and an interface for apps to access the shared memory.

memory emulation is consistent across all participants, providing the view that only a single object exists (instead of multiple copies distributed across the system). The desired sequence (and its formal properties) are called *consistency conditions*.

Enforcing consistency is critical to making the simulation work, but it is not easy. Some devices could be out of sync with others, having a different idea of the operations that have happened across the distributed system, or a memory state that differs from the memory that other devices have. Some devices could invoke an operation, but operations could take time to finish, and overlap with other pending operations. If a device does not hear from the other devices in a timely manner, it may not hear about recent changes made on the shared memory object.

The simulation approach is made more difficult in dynamic systems, where devices are entering and leaving the system unpredictably. These dynamic systems are said to have "churn". See Figure 1.2 for a sketch of nodes entering and exiting the system.

When a device leaves or fails, it may take replicated copies of memory or recent changes to the memory with it. If only some of the devices were informed of the change, this could cause disagreements on the overall history of the shared memory object.

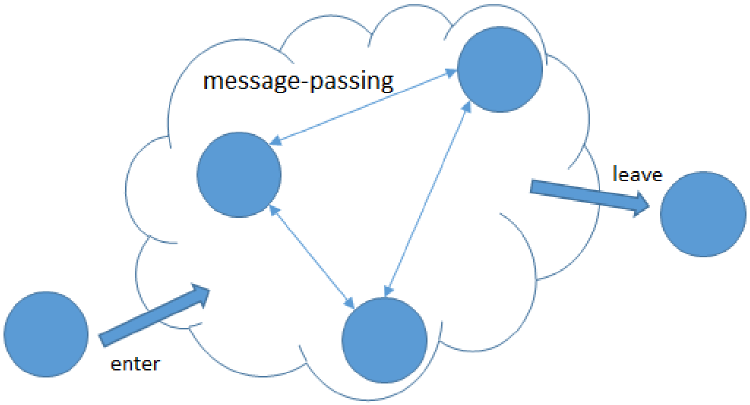

Figure 1.2: A graphic demonstrating churn. Devices may enter and leave the system continuously.

CCReg provides an algorithm to emulate a shared read/write register which can work in these dynamic systems. CCReg is not the only algorithm to do so; there are other algorithms which emulate shared read/write registers in dynamic systems [5] [6] [7], each with different assumptions about failure models, message delays, whether the churn is continuous or eventually disappears, and other factors.

The CCReg paper provides proofs that the algorithm eventually finishes all operations (a property called *liveness*), and that consistency is maintained through every operation. In particular, CCReg implements a particular consistency condition called *linearizability*. CCReg guarantees liveness and linearizability, so long as the churn rate and failure fraction are within a limited range.

We would like to see what happens when the system is pushed beyond the bounds on churn rate and failure fraction. The theoretical statement does not guarantee that the algorithm should succeed when the bounds are exceeded. And if the algorithm fails in extreme cases, it suggests that the theoretical analysis is close to the algorithm's true limits. On the other hand, if the algorithm maintains liveness and linearizability, it suggests that the analysis could be improved, and that the true theoretical limits have yet to be discovered.

2. SHARED REGISTER EMULATION ALGORITHM

2.1 Model Description

The CCReg algorithm from [1] simulates the interface and functionality of a shared-memory system by using a message-passing system. We now describe the model of the message-passing system CCReg operates in.

The message-passing system consists of many nodes, where each node is an abstract representation of a device. Each node has a unique ID, and can have one of several states. It can be Present, indicating that it's in the system and propagating messages. It can also be Joined, indicating that it's in the system, and also has permission to modify the simulated shared memory. Every node that is Joined is also Present, but not every Present node is Joined.

The shared memory we try to simulate has exactly one variable. Each node keeps a local variable *val*, which has the last known state of the shared variable.

Messages between nodes can be delayed by some amount of time, from 0 to D. This allows us to assume that messages will eventually reach their destination, and aren't lost. D is a parameter of the system, but is not known to the individual nodes. We assume that all received messages were sent by some node, and messages from the same node are received in the order which they were sent.

Nodes can join and leave the system using subroutines of the CCReg algorithm. However, there are limits on how many joins and leaves can occur. We assume that in any window of time $[t, t + D]$, the number of joins and leaves is at most $\alpha \cdot N(t)$, where $N(t)$ is the total number of nodes in the system at time t, and α is a model parameter.

CCReg also tolerates a limited number of crashes. A node crashing is distinct from a node leaving; crashed nodes do not announce their departure to other nodes in the system, but fail silently. At any time t, the number of crashed nodes is at most $\Delta \cdot N(t)$, where Δ is a model parameter. $N(t)$ is the total number of nodes in the system at time t (as above); $N(t)$ includes the crashed nodes in the total.

2.2 Algorithm Description

The algorithm has two primary mechanisms: one for dealing with churn, and one for managing reads/writes to the shared variable.

The churn mechanism is necessary to track the composition of the system (who is present, Joined, left, etc) because the composition keeps changing. As such, each node p_i keeps track of the system changes it's seen, and stores the changes in a local variable $Changes$. When another node p_j requests to Join the system or leave the system, it broadcasts this request to all other nodes; node p_i will receive the message and add it to its local $Changes$.

The join procedure works roughly as follows. When node p_i enters the system, it broadcasts that it would like to Join and asks other nodes what the system composition is. Each node that hears p_i's request will reply, and send p_i their local $Changes$. p_i will estimate how many nodes need to respond to p_i's request, based on the $Changes$ variables it has received. Once p_i has received enough messages, the join procedure is complete. (Here, "enough" is defined as $\gamma\%$ of the believed system size, where γ is an algorithm parameter.)

The leave procedure is simpler. If a node p_i wishes to leave, it will broadcast its departure, and other nodes will add p_i's departure to their $Changes$ variables.

Nodes use a similar (though not identical) procedure for handling both reads and writes to the shared variable. First, a node p_i broadcasts to other nodes, asking for information about the shared variable. p_i waits for $\beta\%$ of Joined nodes to respond, where β is an algorithm parameter. After receiving enough information, p_i updates its local state, and broadcasts the change. Then p_i waits for $\beta\%$ of Joined nodes to accept the change, and change their local states as well.

2.3 Algorithm Properties

The original churn paper gives assumptions about the shared memory system, and bounds on the parameters α, β, Δ, and γ. Should these assumptions and bounds hold, CCReg is guaranteed to satisfy a set of safety and liveness properties. These are the properties we would like to examine when the theoretical bounds are violated.

Liveness properties. If a node invokes an operation, and the node does not leave or crash, then that operation eventually completes, and the node receives a proper response to the operation. That is:

- every node that enters the system and does not leave or crash eventually completes the Join procedure

- If a node begins a read or a write, and the node does not leave or crash, eventually its read or write will finish. For a read, the node receives the value read from the shared memory emulation. For a write, the node receives an acknowledgment that the write completed.

Safety properties. CCReg provides linearizability [8]. Any execution of the CCReg algorithm consists of a combination of reads, writes, enters, leaves, or crashes that satisfy the bounds α and Δ. For any such execution, there is a way to order all completed reads and writes, and some subset of uncompleted writes, such that the following are true:

- Every read returns the value of the latest preceding write

- If an operation op_1 finishes before some other operation op_2 begins, then op_1 is ordered before op_2. This is true regardless of which node began op_1 or op_2.

In short, when the parameter bounds and assumptions are maintained, the algorithm should complete joins, complete read and write operations, and choose the right values for the reads.

3. METHODOLOGY

3.1 Implementation in DistAlgo

When implementing the algorithm in DistAlgo, the core functionality of the algorithm was not changed, but some additions to the supporting structure had to be made. Some additions were to accommodate the limitations of the DistAlgo language. Others were made to introduce randomness, necessary for simulating the many potential executions, and introducing asynchronous message delays.

The first major addition is the Scheduler object. The Scheduler has two responsibilities: it simulates node-to-node communication, and triggers events across the system.

In the original paper the nodes communicate directly with each other. However, the way DistAlgo implements its processes does not allow this to happen. In DistAlgo, when starting a simulation, each instance of the Node class is created as a separate process in memory. Inter-process communication is possible, but the sending process must know the recipient process's ID. But each node is randomly assigned a process ID by the operating system, so there is no way for a node to know apriori what the ID of another node will be, nor is there any way to fix a specific ID.

Our resolution uses the Scheduler class. The Scheduler class's setup() method is responsible for creating all Node instances, so the Scheduler can directly access the process IDs of all Node instances. With this, the scheduler can listen for an outbound message from one node, and forward the message to all the other nodes. A diagram of the Scheduler's interactions with the system can be seen in Figure 3.1.

Because the Scheduler has access to all the nodes, it can also trigger, coordinate, and record events across the system. It can send a message to any node, telling that node to begin some operation. Before choosing an operation, the scheduler checks which operations are permitted, checking against system model parameters α and Δ. A new node can enter the system if, after entering the system, the churn constraint is not violated. A node is allowed to crash if, after crashing, the failure rate constraint is not violated. A node is allowed to leave if, after leaving, both the churn rate and failure rate constraints are not violated. (The failure rate constraint is necessary as it is calculated from current system size; if the current system size is reduced, then the fraction of crashed nodes grows.) Finally, reads and writes are permitted, but are only allowed on nodes that haven't crashed, and on nodes that are not in the middle of a previous read or write. (If no such nodes meet these qualifications, then reads and writes are disallowed.)

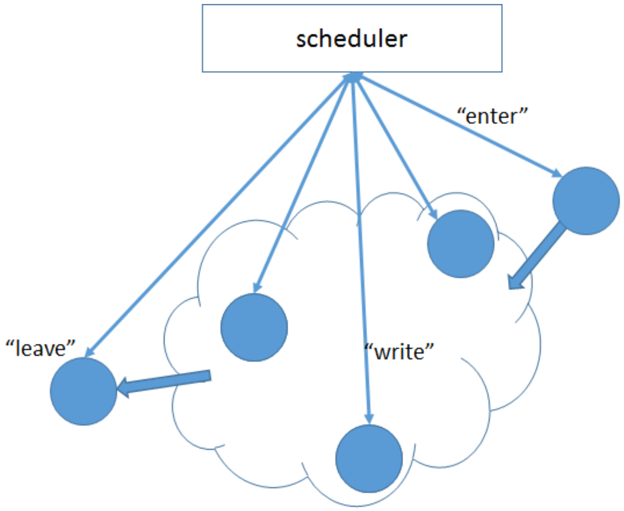

Figure 3.1: The scheduler facilitates interaction with the nodes. It simulates node-to-node communication, and simulates external client devices invoking read/write operations on the shared memory.

The second major addition is adding randomness to parts of the implementation. The scheduler randomly chooses which operation to perform, which node should invoke it, and how long to wait before choosing the next operation. This is to create a better simulation of "average" case behavior.

The nodes themselves also have arbitrary delays on communication, for the purposes of simulating asynchronous message-passing. We would like messages to take non-uniform amounts of time to be received. When a node receives a message, it waits a random amount of time before proceeding with its response. The random delay is therefore placed in each of the message receiving functions. If applied to the sender, the delay is only chosen once and then applied to all communications uniformly, so all recipients wait the same amount of time. The random delay also cannot be placed in the scheduler, as that has the same issue of only choosing the delay once. See Figure 3.2 for a pictorial illustration of where delays could be placed.

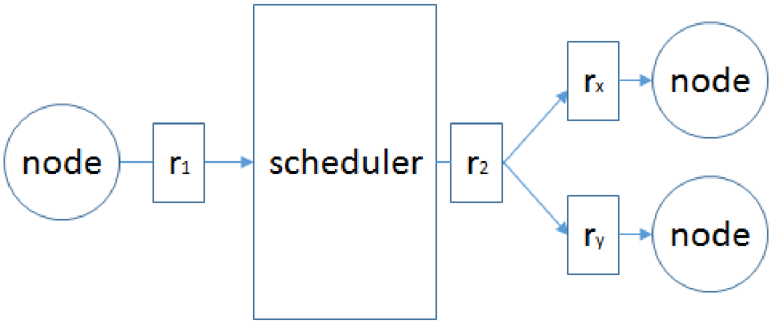

Figure 3.2: Inducing randomness on message delays. Observe that placing randomness in r_1 or r_2 would result in that value being applied uniformly to all communications, which is what we want to avoid. Allowing each receiver to decide its own random delay value, indicated by r_x and r_y, avoids this issue.

We also require that nodes respond to messages asynchronously (that is, for nodes to respond to incoming messages while in the middle of doing other tasks). For this, we need to use DistAlgo's receive() handlers, as other means of responding to messages could block other tasks from being executed.

3.2 Simulation Parameters

The general approach is to fix a known set of allowable β and γ values, as a particular configuration for the algorithm. We then vary α and Δ, parameters of the system, past what is theoretically allowed. A table of allowed parameter sets is available in Table 3.1, reproduced from the CCReg paper. Observe the limited amount of churn and crashing the algorithm can tolerate, according to theoretical analysis.

Table 3.1: A table of theoretically allowed parameter values for CCReg.

churn rate α	failure fraction Δ	minimum system size N_{min}	join bound fraction γ	read/write bound fraction β
0	0.33	N/A	N/A	0.665
0.01	0.26	7	0.67	0.684
0.02	0.19	7	0.69	0.701
0.03	0.13	8	0.70	0.726
0.04	0.06	9	0.72	0.737
0.05	0	10	0.74	0.755

We fix β and γ to be 0.726 and 0.7, respectively. These values constrain that $\alpha = 0.03$ and $\Delta = 0.13$, according to the theoretical analysis. Other values of β and γ could be chosen, but extreme values of either force either α or Δ to be 0 (a system with no churn or no crashing), which is less indicative of the algorithm's performance as a whole.

To actually push the boundaries of the algorithm, we go past the theoretical bounds of α and Δ. For controlled analysis of when the properties are violated, we fix one and slowly increment the other (by 0.05 at a time).

We also set the number of initial nodes to be 50. We should set the number of initial nodes high enough so we can actually see some churning behavior. If the number of initial nodes is too low, the churn constraint $\alpha * N(t)$ is < 1, which means no churns (and no crashes) are allowed.

The DistAlgo implementation has two additional parameters, used for running the CCReg algorithm on real hardware.

The first parameter is D, the maximum message delay. The D parameter exists in the CCReg paper and is used for theoretical analysis, though the algorithm itself does not know what D is. In particular, D is used to determine if churn operations are allowed, since the scheduler examines the past operations from times $[t - D, t]$. This is summarized in Figure 3.3.

$$D = d_{real} + max(d_{random})$$

Figure 3.3: Formula for calculating D.

We set the maximum message delay D to be equal to the average real-time communication delay d_{real}, plus the maximum value of the induced random delay d_{random}. The average real-time communication delay is calculated by measuring how long it takes a read operation to finish without any induced random delays. The length of a read operation is then divided by 4, as the read operation takes two round trips of up to $2D$ each. (The same is true of the write operation.) In practice, for a maximum $d_{random} = 3$ seconds, we found D to be around 4.5 seconds.

The second implementation parameter is the maximum amount of time until the next operation, D_{op}. After the scheduler invokes some operation on some node, the scheduler selects a random number of seconds to wait, from 0 to D_{op}, before invoking the next operation. This parameter has no equivalent in the CCReg paper, and is used only for the implementation. We choose $D_{op} = 5$ seconds.

While choosing D_{op} may seem like an arbitrary choice, there are some considerations to take. Consider the churn constraint, counting the number of churn events in $[t - D, t]$. If D_{op} is $>> D$, then operations are spaced out very far in time. Then, the churn events happen too far in the past for them to be considered in the time interval $[t - D, t]$. So the churn constraint ends up untested. We would like to stress-test all parts of the algorithm at once, so we would like multiple churn events to happen in the $[t - D, t]$ interval. On the other hand, we would like to avoid having *all* operations overlap, as this would prevent us from testing the linearizability ordering property. Thus, we choose D_{op} to be slightly larger than D; in our case we chose $D_{op} = 5$. We rely on the randomness of D_{op} selections, such that some operations happen quickly and overlap, but some do not.

We run the DistAlgo program on a single machine, where each node is given a separate thread. Each time the scheduler invokes an operation, we log the operation (read/write/etc), the node chosen to start that operation, and the realtime when that operation began. Each time any operation finishes, we log the realtime when it finishes, and the output of that operation (for reads).

We verify that operations maintain the liveness condition and the linearizability condition. The liveness condition is easy to observe in practice, as the algorithm implementation fails to terminate if some operations are left incomplete.

For verifying the linearizability condition, we take the output logs and examine the read and write operations. We use an algorithm described by Gibbons and Korach [3] to check that an order of operations that satisfies linearizability exists. This particular algorithm (described in Theorem 4.1) requires that every read can be uniquely mapped to exactly one write. For our purposes, we define this as: if a read operation returns x, there is only one write operation writing x, and not multiple $write(x)$ operations the read could correspond to. To do this, our implementation chooses increasing write values, to guarantee that each write value is used exactly once.

4. RESULTS

4.1 Increasing Churn Rate

To our surprise, increasing churn rate did not seem to have an adverse effect on the algorithm's performance. Even with $\alpha = 1$, the algorithm successfully completed all operations and maintained linearizability across multiple executions. This goes against our intuition. It also poses a conflict with the CCReg paper; section 4 of the paper offers a proof that for rapid churns, linearizability is violated. There are multiple possible reasons for this discrepancy.

First, even though $\alpha = 1$, this only indicates that churn is always permitted; this does not mean that churn is always happening. Because the scheduler picks operations randomly, even if churn is permitted, the scheduler may not choose to invoke an enter/leave operation. With uniformly-distributed operation selection, a churn operation is only invoked 20% of the time, which seems to be a more reasonable value of churn rate.

It is also possible that churns were simply not happening "fast enough" to approximate the rapid churns that would violate linearizability. The system composition was allowed to change constantly, but the composition was changing so slowly that it didn't matter.

The general theme of the above ideas is that we came across many randomized executions, but we didn't encounter any executions with truly adversarial behavior. The set of potential executions is very, very large, and the adversarial executions could be a very small subset of those executions.

4.2 Increasing Crash Fraction

Also to our surprise, increasing the crash fraction Δ did not seem to have an adverse effect on the algorithm's performance. Even with $\Delta = 1$, the algorithm successfully completed all operations and maintained linearizability across multiple executions. Section 5 of the CCReg paper offers a lower bound on crash resiliency, which conflicts with this result.

There are multiple possible reasons for this discrepancy. The intuition behind random operation selection could apply again (even though crashes were always allowed, the scheduler did not always schedule crashes).

Basing our line of inquiry on the known lower bound, we decided to try one adversarial execution where a certain percentage of nodes crashed in the beginning, and the rest of the execution was randomized as normal. The percentage of nodes crashed was more than what was possible given the lower bound. Unsurprisingly, the algorithm failed to terminate in that case.

We then lowered the number of initial crash failures. At $\Delta = 0.28$ the algorithm failed to terminate and no operations were completed; at $\Delta = 0.26$ all operations completed. In a sense this is not very surprising, since each node is waiting to hear from $\beta = 72.8\%$ of the system. If 28% of the nodes have crashed, then of course that node will never hear from more than 72% of the system anyways.

This suggests that an alternate approach is required for measuring the impacts of increased crashing. That is, instead of fixing β and γ (which imply constraints on the failure fraction), we should fix α and Δ, and lower β and γ until linearizability or liveness is broken. The idea behind the β and γ parameters is that if p_i invokes an operation, after hearing back from that many nodes, at least one "good" node will have the correct memory state. Decreasing these parameters will decrease the number of nodes heard from, and increase the chance that p_i never hears from this "good" node.

4.3 Discussion and Future Work

The "average" case behavior provided by randomized executions is promising; it suggests that the algorithm is not brittle, and can withstand higher churn rates and crash failure fractions than the bounds would suggest. However, this analysis is not enough to form definite conclusions. This motivates a future direction for this research: carefully-crafted adversarial executions to determine a more precise breaking point, stripping away the uncertainty of randomized analysis.

As an example, one adversarial execution focuses on the churn rate by having many nodes enter at once (more than what the theoretical α allows), entering as fast as possible. One possible flaw of the randomized approach is that, by randomly choosing which node to invoke an operation on, some nodes could remain idle (not doing any operations of their own). This adversarial execution aims to avoid that problem by keeping new nodes as busy as possible. As soon as a node finishes the join procedure, it will repeatedly invoke read and write operations. The intuition is that, because so many nodes are joining at once, the new nodes only hear from the nodes originally present in the system and have an outdated view of the system size. Then when a node invokes a write, its change to the shared variable is not propagated across the system; other nodes fail to hear about it, and consequently return the wrong read value.

REFERENCES

[1] 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 Transactions on Parallel and Distributed Systems*, vol. 30, pp. 544–559, March 2019.

[2] Y. A. Liu, S. D. Stoller, and B. Lin, “From clarity to efficiency for distributed algorithms,” *ACM Trans. Program. Lang. Syst.*, vol. 39, pp. 12:1–12:41, May 2017.

[3] P. B. Gibbons and E. Korach, “Testing shared memories,” *SIAM J. Comput.*, vol. 26, pp. 1208–1244, Aug. 1997.

[4] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. USA: John Wiley & Sons, Inc., 2004.

[5] N. A. Lynch and A. A. Shvartsman, “Rambo: A reconfigurable atomic memory service for dynamic networks,” in *Proceedings of the 16th International Conference on Distributed Computing*, DISC ’02, (Berlin, Heidelberg), pp. 173–190, Springer-Verlag, 2002.

[6] R. Baldoni, S. Bonomi, and M. Raynal, “Implementing a regular register in an eventually synchronous distributed system prone to continuous churn,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, pp. 102–109, Jan 2012.

[7] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer, “Dynamic atomic storage without consensus,” *J. ACM*, vol. 58, pp. 7:1–7:32, Apr. 2011.

[8] M. P. Herlihy and J. M. Wing, “Linearizability: A correctness condition for concurrent objects,” *ACM Trans. Program. Lang. Syst.*, vol. 12, pp. 463–492, July 1990.