FAULT-TOLERANT DISTRIBUTED SERVICES IN MESSAGE-PASSING SYSTEMS

A Dissertation

by

SAPTAPARNI KUMAR

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Jennifer L. Welch
Committee Members,	Anxiao (Andrew) Jiang
	Serap Savari
	Tiffani Williams
Head of Department,	Dilma Da Silva

August 2019

Major Subject: Computer Science

ABSTRACT

Distributed systems ranging from small local area networks to large wide area networks like the Internet composed of static and/or mobile users have become increasingly popular. A desirable property for any distributed service is fault-tolerance, which means the service remains uninterrupted even if some components in the network fail. This dissertation considers weak distributed models to find either algorithms to solve certain problems or impossibility proofs to show that a problem is unsolvable.

These are the main contributions of this dissertation.

- Failure detectors are used as a service to solve consensus (agreement among nodes) which is otherwise impossible in failure-prone asynchronous systems. We find an algorithm for crash-failure detection that uses bounded size messages in an arbitrary, partitionable network composed of badly behaved channels that can lose and reorder messages.

- Registers are a fundamental building block for shared memory emulations on top of message passing systems. The problem has been extensively studied in static systems. However, register emulation in dynamic systems with faulty nodes is still quite hard and there are impossibility proofs that point out scenarios where change in the system composition due to nodes entering and leaving (also called churn) makes the problem unsolvable.

  We propose the first emulation of a crash-fault tolerant register in a system with continuous churn where consensus is unsolvable, the size of the system can grow without bound and at most a constant fraction of the number of nodes in the system can fail by crashing. We prove a lower bound that states that fault-tolerance for dynamic systems with churn is inherently lower than in static systems.

- We then extend the results in the crash-fault tolerant case to a dynamic system with continuous churn and nodes that can be Byzantine faulty. It is the first emulation of an atomic register in a system that can withstand nodes continually entering and leaving, imposes no upper bound on the system size and can tolerate Byzantine nodes. However, the number of Byzantine faulty nodes that can be tolerated is upper bounded by a constant number. Although the algorithm requires that there be a constant known upper bound on the number of Byzantine nodes, this restriction is unavoidable, as we show that it is impossible to emulate an atomic register if the system size and maximum number of servers that can be Byzantine in the system is unknown.

DEDICATION

To ma.

ACKNOWLEDGMENTS

I arrived in College Station on August 26, 2013. I was in a state of shock after the first impression of this town. It didn’t quite live up to my “expectations". The America I knew was New York, Boston and Los Angeles; all the cities that are portrayed in Hollywood movies, and College Station was nowhere close to that image. I swallowed sadness and knew I had to move on. Then I met my hero: Dr. Jennifer Welch.

I don’t think an “acknowledgements" section does justice to my feelings for all the people who helped me go through the numerous discussions (many of them heated), sleepless nights, frustration, tears, and of course satisfaction, smiles and laughter. My deepest indebtedness to my advisor, Dr. Jennifer Welch for her time, patience, support and belief in me. This dissertation would never have been possible without her guidance and friendship. She made my PhD. journey extremely satisfying and I will dearly miss our brainstorming sessions. Of course she taught me how to be a researcher and develop critical thinking, but she also showed me how to be a beautiful human with compassion, ethics and humility. Thank you.

My deepest thanks to Dr. Hagit Attiya, Dr. Faith Ellen and Dr. Hyun Chul Chung for all the wonderful discussions we had over the past years. I also thank Dr. Andrew Jiang, Dr. Serap Savari and Dr. Tiffani Williams for serving on my committee and for their insightful comments and questions. My research was financially supported by the National Science Foundation grants 1526725 and 1816922.

Most importantly, I would like to thank my family for their undying support. Thank you ma for believing in me. Thank you Payel for always making me feel better even during days when I lost hope. Thanks baba for keeping me close to reality. Finally, I would like to thanks my friends in College Station and back in India. Without your love and support this dissertation would have been impossible.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor Jennifer L. Welch [advisor], Professor Anxiao Jiang and Professor TIffani Williams of the Department of Computer Science and Engineering and Professor Serap Savari of the Department of Electrical and Computer Engineering.

Funding Sources

Graduate study was supported by Research Assistantship from Dr. Jennifer Welch from NSF grants 1526725 and 1816922. It was also supported by Teaching Assistant fellowships and a Graduating Teaching Fellowship (GTF) from Texas A&M University.

TABLE OF CONTENTS

Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

1.1 Introduction and Related Work

A distributed system is a collection of computers (or nodes) that communicate amongst themselves via wired or wireless communication channels, to perform a given task. Distributed computing refers to the use of distributed systems to solve computational problems. There are several reasons for using distributed systems and distributed computing: (1) there might be a need for communication network in an application (for example, if data produced at one geographical location is required as an input at a different geographical location), (2) it is more cost efficient to build several low-end computers than a single high-end one, (3) distributed systems are more reliable than sequential systems as there isn't just one point of failure, and (4) it is easier to expand a distributed system than a single node.

There are two main models of computation considered in distributed systems: *shared-memory* model and *message-passing* model. In the shared memory model, memory is shared by the nodes in the system, which can exchange information by reading and writing data. In the message-passing model, communication takes place by means of messages exchanged between the nodes.

Distributed systems can be broadly categorized as *synchronous* or *asynchronous* based on the timing guarantees the system provides. More specifically, a synchronous message-passing system comes with strong guarantees: nodes have globally synchronized clocks and the upper

Parts of the material in this chapter are reprinted with permission from the following papers:

"Emulating a shared register in a system that never stops changing" by Hagit Attiya, Hyun Chul Chung, Faith Ellen, Saptaparni Kumar and Jennifer L. Welch, 2019. IEEE Transactions on Parallel and Distributed Systems, vol. 30, pp. 544-559, copyright [2019] by IEEE

"Simulating a shared register in an asynchronous system that never stops changing" by Hagit Attiya, Hyun Chul Chung, Faith Ellen, Saptaparni Kumar and Jennifer L. Welch, 2015. in Proceedings of 29th International Symposium on Distributed Computing, pp. 75-91, copyright [2015] by Springer

"Implementing $\Diamond P$ with Bounded Messages on a Network of *ADD* Channels" by Saptaparni Kumar and Jennifer L. Welch, 2019. Parallel Processing Letters, Volume 29, No 1, 1950002, copyright [2019] by World Scientific Publishing Company bound on the message delivery time is a known value D. Examples of synchronous systems are certain large centralized multiprocessing computers and VLSI chips containing many separate parallel processing elements. An asynchronous message-passing system comes with the weakest guarantees: nodes have no notion of real time as they do not possess synchronized clocks and there is no upper bound on the message delivery time.

Distributed message-passing systems can be either *static* or *dynamic* depending on the topology of the system. In static systems the system composition is fixed, i.e., the nodes in the system do not change over time. There has been a lot of research on static systems and most problems in distributed computing like leader election [1], mutual exclusion [2], etc. have been formulated with static systems in mind. In dynamic systems, nodes may enter, leave or move in the system with time, resulting in changes in the system composition. With the advent of smartphones, intelligent cars, unmanned aerial vehicles, weather bots, etc., dynamic systems have become very realistic models of computation.

In distributed message-passing systems, the possibility of channel and node failures makes problem solving challenging. A *fault-tolerant* algorithm is one that solves the problem correctly even in the presence of failure of some of its components (nodes or channels). It is important to tolerate faults as it provides robust systems that have high availability. There are several types of faults that need to be tolerated. A *crash* failure occurs when a node halts, but was working correctly until it halts. An *omission* failure occurs when a node fails to receive incoming messages or send outgoing messages. A *timing* failure occurs when a node's message delivery lies outside the specified delivery time interval (if any). A *Byzantine* failure [3] is considered to be the worst kind of failure that can happen in any distributed system. Malicious attacks, operator mistakes, software errors and conventional crash faults are all encompassed by the term Byzantine failures.

In this dissertation, we describe our work to provide fault-tolerant distributed services in distributed systems. A service can be thought of as a building block, used to solve important and interesting problems for other applications. The services we provide are introduced next.

1.1.1 Failure Detection in Partitionable Networks Composed of Ill-Behaved Channels

Failure detectors were proposed by Chandra and Toueg [4] as oracles to be used to identify failed nodes in an asynchronous message-passing system with crash failures. They are an important distributed service in the network topology layer of a distributed system, as they help circumvent the FLP [5] impossibility result which states that it is impossible to solve the agreement problem (also known as consensus [6, 7]) in a crash-prone asynchronous message-passing system.

Consensus is a very fundamental yet universal problem. In distributed computing and multi-agent systems, it is often necessary to achieve overall system reliability in the presence of a number of crash-faulty processes. This often requires processes to agree on some data value that is needed during computation. Examples of applications of consensus include whether to commit a transaction to a database, agreeing on the identity of a leader, state machine replication, and atomic broadcasts. Real world applications include clock synchronization, PageRank, opinion formation, smart power grids, state estimation, control of UAVs, load balancing and others.

Failure detector implementation in practice requires some degree of partial or even full synchrony. There are two main lines of research in the area of failure detectors. The first one involves implementing failure detectors on increasingly weaker system models that represent practical applications and the second one involves finding the weakest failure detector for solving a given problem. We contribute to the first line of research and implement a failure detector as a service to solve other important problems like consensus, leader election and clock synchronization (in the middleware layer of a distributed system), on a failure-prone, distributed message-passing system composed of channels that are ill-behaved (they may lose and reorder messages). Our failure detection algorithm uses bounded sized heartbeat messages, timeouts and path information to determine if there is a reliable path between two nodes.

1.1.2 Fault-Tolerant Register Implementation in Systems with Churn

A distributed data structure is an important service in the middleware layer of a distributed system that provides data storage and organization for access by multiple nodes in the system. The shared-memory model is considered to be a more convenient programming model than message-passing and distributed data structures provide the illusion of shared-memory on top of message-passing models.

However, implementing a data structure in a message-passing system is not a trivial task to accomplish mainly due to the presence of concurrency. Take a distributed register data structure for example. If two nodes invoke a *write* operation on a distributed register concurrently, the question remains, in which order should the *write* operations be executed? To tackle this issue, correctness for a distributed data structure is defined by the *sequential specification* of the data structure and a *consistency condition*. Each data structure has a sequential specification which specifies its behavior in the absence of concurrency. A consistency condition is a set of rules that tie together the sequential specification with what happens in the presence of concurrency. One of the most widely used consistency conditions is *linearizability*, introduced by Herlihy and Wing [8] in 1990. A data structure is said to be linearizable if it guarantees that every operation appears to happen at a single point in time between the invocation and response of the operation.

Now we move our attention to the implementation of registers. A shared register is a fundamental service used by middleware in a distributed system, that stores a value and has two operations: *read*, which returns the value stored in the register, and *write*, which updates the value stored. Implementing a shared register on top of an asynchronous message-passing system is an important task in distributed systems as registers serve as building blocks for more powerful data structures like queues, stack, etc.

Shared register implementations can be complicated by the possibility of faulty nodes in the system. A fault can range from being benign (crash faults) to extremely adversarial (Byzantine faults). To tackle the possibility of faults and for load balancing, many shared register implementations replicate the value of the register in multiple servers and require readers and writers to communicate with a certain fraction of servers. A lot of research in this area has focused on implementing linearizable (also called *atomic*) shared registers. In this dissertation, we deal with register implementation that tolerates two types of faults: crash faults and Byzantine faults. We weaken the system model by making it dynamic. Along with fault tolerance, we tolerate *churn*: change in system composition due to nodes entering and leaving, thus making the system guarantees extremely weak.

We now briefly describe the main results we have for fault-tolerant atomic register emulations.

1.1.2.1 Crash-Tolerant Registers

Most of the work in the area of crash fault tolerant registers has focused on emulating atomic shared registers. For example, the ABD emulation [9] replicates the value of the register in server nodes. It assumes that a majority of the server nodes do not fail. Consider the simplified case of a single writer and a single reader. To write the value v, the writer sends v, tagged with a sequence number, to all servers and waits for acknowledgments from a majority of them. Similarly, to read, the reader contacts all servers, waits to receive values from a majority of them, and then returns the value with the highest sequence number. This approach can be extended to the case of multiple writers and multiple readers by having each operation consist of a read phase, used by a writer to determine its sequence number and used by a reader to obtain the return value, followed by a write phase, used by a writer to disseminate the value (and sequence number) and used by a reader to announce the sequence number of the value it is about to return [10]. We know from the ABD emulation [9] and Attiya and Welch [6] that a crash-fault-tolerant atomic register in the static setting can be implemented if and only if a majority of the nodes are non-faulty.

The success of this approach for static systems, where the set of nodes (readers, writers, and servers) is fixed, has motivated several similar emulations for dynamic systems, where nodes may enter and leave. Existing simulations of atomic registers rely either on the assumption that churn eventually stops for a long enough period (e.g., DynaStore [11] and RAMBO [12]) or on the assumption that the system size is bounded (e.g., [13]). In our atomic register implementation, we want to take a different approach: We want to allow churn to continue forever, while still ensuring that read and write operations complete and nodes can enter and leave the system thus allowing changes in the system size.

Our work in Section 3.1 presents the first emulation of a register supporting any number of readers and writers in a crash-prone system that can withstand nodes continually entering and leaving and imposes no upper bound on the system size. The algorithm works as long as the number of nodes entering and leaving during a fixed time interval is at most a constant fraction of the system size at the beginning of the interval, and as long as the number of crashed nodes in the system is at most a constant fraction of the current system size.

In addition to that, we prove a lower bound on the fraction of correct nodes that is strictly larger than the fraction sufficient to solve the problem in the static case.

1.1.2.2 Byzantine-Tolerant Registers

Byzantine faults are considered to be the worst kind of faults that can happen in any distributed system. Emulating a Byzantine-tolerant register requires replicating the register value on to more than two-thirds of the servers. Emulating a register in a dynamic system where servers and clients can enter and leave the system and be faulty is harder than in static systems.

Our work in Section 3.2 presents the first emulation of a multi-reader multi-writer atomic register in a system that can withstand nodes continually entering and leaving, imposes no upper bound on the system size and can tolerate Byzantine servers. The algorithm works as long as the number of servers entering and leaving during a fixed time interval is at most a constant fraction of the system size at the beginning of the interval, and as long as the number of Byzantine servers in the system is at most f. Although the algorithm requires that there be a constant known upper bound on the number of Byzantine servers, this restriction is unavoidable, as we show that it is impossible to emulate an atomic register if the system size and maximum number of servers that can be Byzantine in the system is unknown to the nodes.

1.1.3 Roadmap

Section 2.1 describes our work [14, 15] in implementing a failure detector in an arbitrary partitionable network composed of ill-behaved channels. Paper [15] has been accepted to Parallel Processing Letters, 2019. Our work on crash-tolerant atomic register implementation in systems with continuous churn is described in Section 3.1. The preliminary version of this work [16] appeared in International Symposium on Distributed Computing (DISC), 2015. The extended version of this work [17] appears on IEEE Transactions on Parallel and Distributed Systems (TPDS), 2018. Section 3.2. describes our work on Byzantine-tolerant atomic register implementation in systems with continuous churn. This work will be submitted soon. Finally in Section 4.1, we conclude this dissertation.

2. FAILURE DETECTION

2.1 Failure Detection in an Arbitrary, Partitionable Network Composed of ADD Channels

In this section, we present an implementation of the eventually perfect failure detector ($\Diamond P$) from the original hierarchy of the Chandra-Toueg [18] oracles on an arbitrary partitionable network composed of unreliable channels that can lose and reorder messages. Prior implementations of $\Diamond P$ have assumed different partially synchronous models ranging from bounded point-to-point message delay and reliable communication to unbounded message size and known network topologies. We implement $\Diamond P$ under very weak assumptions on an arbitrary, partitionable network composed of *Average Delayed/Dropped (ADD)* channels [19] to model unreliable communication. Unlike older implementations, our failure detection algorithm uses bounded-sized messages to eventually detect all nodes that are unreachable (crashed or disconnected) from it.

2.1.1 Introduction and Related Work

The *consensus* [6, 7, 20, 21, 22] problem in distributed systems requires agreement among a number of nodes for a single data value. It is an important problem because if we can solve consensus in a distributed system, we can use it solve numerous other problems like leader election [1], atomic broadcast [23], transaction commit [24] and clock synchronization [25]. However, the FLP [5] impossibility result states that it is impossible to solve consensus in an asynchronous message-passing system even if there is just one crash failure. In 1996, Chandra and Toueg [4] proposed a hierarchy of oracles (*failure detectors*) to be used to identify failed nodes, by differentiating them from slow ones, in a crash-prone asynchronous message-passing

Parts of the material in this chapter are reprinted with permission from:
“Implementing $\Diamond P$ with Bounded Messages on a Network of *ADD* Channels" by Saptaparni Kumar and Jennifer L. Welch, 2019. Parallel Processing Letters, Volume 29, No 1, 1950002, copyright [2019] by World Scientific Publishing Company

system. These failure detectors are unreliable and can give wrong information by incorrectly suspecting correct nodes, and/or not suspecting crashed nodes. In spite of that, many oracles are powerful enough to solve consensus. However, their implementation in practice requires some degree of partial synchrony. Freiling et al. [26] provide an informative survey on the failure detector abstraction both as building blocks for the design of reliable distributed algorithms and as computability benchmarks.

Failure detectors can be described by their *accuracy* and *completeness* properties in a non-partitionable system. For example, an eventually-perfect failure detector ($\Diamond P$) satisfies *strong completeness* and *eventual strong accuracy.* Intuitively, $\Diamond P$ can give incorrect information about the nodes in the system for an unknown finite amount of time, after which it provides perfect information about all nodes in the system. Strong completeness is satisfied if the failure detector of each node eventually suspects all nodes that are crashed. Eventual strong accuracy is satisfied if the failure detector of every node eventually stops suspecting all nodes that are correct. These definitions were originally for systems that do not partition. In [27], these definitions of accuracy and completeness for failure detectors were extended to partitionable networks. The paper by Chen et al. [28] studies accuracy and completeness properties (quality of service) of failure detectors and quantifies how fast different implementations of oracles detect failures and how well they avoid false suspicions.

Ill-behaved channels make the problem of implementing failure detectors harder. Sastry and Pike [29] introduced the framework of ADD (Average Delayed/Dropped) channels as a way to model realistic systems. An ADD channel can arbitrarily lose and reorder messages but offers some weak guarantees on the delivery of “privileged" messages. Privileged messages are never lost and there is an upper bound on their delivery time. However, nodes cannot differentiate between privileged and unprivileged messages as they are distinguished solely by the channel. The authors exploit the channel properties to implement $\Diamond P$ on a fully connected network. We use these ADD channels as building blocks in our system model.

Papers [30] and [31] have discussed algorithms to perform failure detection on arbitrary networks composed of ill-behaved channels using counters as heartbeats for the nodes in the system. Unlike our approach, the message sizes in their algorithms are unbounded. Papers [32] and [33] do failure detection using bounded sized messages, but unlike our work, they assume that the underlying communication channels are reliable.

2.1.2 Contributions

There are two main lines of research in the area of failure detectors. The first one involves implementing failure detectors as a service in the network topology layer of a distributed system, on increasingly weaker system models that represent practical applications and the second one involves finding weaker failure detectors for solving a given problem. We contribute to the first line of research by presenting a novel implementation of an *eventually-perfect* failure detector ($\Diamond P$) from the original Chandra-Toueg hierarchy. Previous works in this area either perform failure detection using bounded sized messages with the assumption that the underlying communication channels are reliable or the algorithms that perform failure detection on ill-behaved channels use unbounded counters in their messages.

The motivation for this work is to extend the failure detector for a fully connected network of ADD channels [19] to a failure detector that uses bounded size messages and works in any arbitrary network composed of ADD channels in which crashes can partition the network. We present a novel algorithm that implements $\Diamond P$ in an **arbitrary (partitionable)** network composed of ADD channels that provide very weak guarantees (**unreliable channels**), using messages that are **bounded in size**.

The failure detection algorithm uses bounded sized heartbeats, timeouts and path information to determine if there is a correct path (all nodes on this path are correct) between two nodes. Periodically, every node sends out its own heartbeat to its neighbors. Every node p has an estimated timeout value for each of its neighbors and if p does not hear from a neighbor q within this estimated time, p suspects q to be crashed. If later p hears from q, indicating that q was falsely suspected, then p increments its timeout value for q. For a node q that is not a neighbor of p, p maintains a set of paths from itself to q. If none of these paths consists solely of nodes that p does not currently suspect, then p suspects q.

2.1.3 Model and Definitions

The distributed system consists of a set Π of n nodes connected in an arbitrary topology by links. We assume that every link connecting two nodes in the network is composed of two unidirectional ADD channels [11], one in each direction. An ADD channel from node p to node q ensures that at least one message out of every r consecutive messages sent by p is received by q within d time; however, the parameters r and d are not known to p and q. The other messages can either be lost or can experience unbounded delays. Nodes may fail only by crashing. Nodes that never crash are called *correct* nodes and those that have not crashed yet are called *live* nodes. Each node that *crash*es remains crashed forever. Each node knows who its neighbors are. Nodes also know the names (ids) of all the nodes in the system. This assumption is necessary, as Jimenez et al. [34] show that without this assumption, no failure detector can be implemented, even in a fully synchronous system with reliable links. Each node has a local clock which generates ticks at a constant rate. Different clocks can tick at different rates and can be unsynchronized.

In more detail, each node is modeled as a state machine with a set of local states and a transition function. Each node's local state includes a constant *neighbors*, which holds the ids of neighboring nodes with respect to the communication graph. An *event* is either the receipt of a message by a node or the expiration of a local timer at a node or a crash of a node; the occurrence of an event triggers a transition for the node, resulting in a new local state and a set of messages to send to the neighbors. A *state* of the system consists of a vector of n local states, one for each node in the system. An *execution* of the system consists of an infinite sequence of alternating states and events, starting with a state. A real time is associated with each event of an execution such that the real times are nondecreasing and they increase without bound. The following must be true:

- At most one crash event occurs per node, and once a node has crashed there are no subsequent events for that node.

- The first state consists of an initial local state for each node.

- Each node p has a local variable *clock* whose value at real time t is $a_p \cdot t + b_p$, for constants $a_p > 0$ and b_p.

- Each subsequent state follows from the previous state by the application of the transition function for the relevant node.

- Timer expiry events occur at each live node as specified in the node's transition function when its clock has the state values.

- Message receipt events occur according to the ADD channel specification: every message sent is received at most once, every message received was previously sent, and, for all neighbors p and q, at least every r-th consecutive message sent by p to q has delay at most d, assuming q is live d time after the message was sent.

The network is initially a connected graph but may eventually be partitioned as nodes start crashing. We call this network a *partitionable network.*

Definition 1. *The* ***network graph*** *at time t is the subgraph of the initial graph obtained by deleting all nodes (and their incident links) that are crashed at time t.*

We denote the network graph at time t as $G(t)$.

We address the problem of implementing, with bounded-sized messages, an *Eventually Perfect* ($\Diamond P$) failure detector that satisfies the following on an arbitrary partitionable network G composed of ADD channels. For each node p, there is a function from p's state to the set of nodes that p suspects. In every execution there exists a time t_f such that for every $t > t_f$ and every correct node p,

- *Strong Completeness:* for every node q that is disconnected from p in $G(t)$, p suspects q at time t

- *Eventual strong accuracy:* for every node q that is connected to p in $G(t)$, p does not suspect q at time t.

2.1.4 Algorithm for Failure Detection

Algorithm 1 Eventually Perfect Failure Detector, Variables for node p

```
Constants:
$neighbors$    // list of neighbors of $p$.
$T$    // integer; time between successive heartbeats
Variables:
$clock()$    // local clock
$last\_contact[.]$    // array of clock times for all neighbors, last time $p$ received
                      //  a message about that node; initially $last\_contact[q] = 0$,
                      //  for all $q \in neighbors$
$suspect\_local[\cdot]$    // array of booleans for all nodes; initially $suspect\_local[q]$
                      // = false, for all $q \in \Pi$. This stores the failure information
                      //  for nodes in $p$'s connected component
$paths[\cdot]$    // Array of sets of paths or sequences of node ids. $paths[q]$ is the set
                  // of paths taken by the heartbeat messages from $q$ to $p$; Initially
                  //  $paths[p] = \{p\}$, $paths[q] = \{q \cdot p\}$ for $q \in neighbors$ and
                  // $paths[r] =$  for all others
$suspect[\cdot]$    // array of booleans for all nodes; it is true for all nodes suspected
                    //  to have failed or disconnected; initially $suspect[q]$ = false,
                    //  for all $q \in \Pi$. This stores the failure information stored in
                    // $suspect\_local[\cdot]$ and also information about disconnected nodes
                    // derived from the $paths[\cdot]$ variable
$timeout[\cdot]$    // Array of time-outs for all neighbors. $timeout[q]$ is the estimated
                    // maximum time between the receipt of successive messages about
                    //  neighbor $q$; initially $timeout[q] = T$, for all $q \in neighbors$
```

Our failure detection algorithm (Algorithm 2) implements $\Diamond P$ over the partitionable network of ADD channels. Every node p maintains a variable *suspect_local*$[\cdot]$ which is an array of booleans to store information about nodes in p's connected component. Since the network is initially a connected graph, this variable, contains an entry for every node in the system and initially, every entry is set to false. As the algorithm progresses over time, *suspect_local*$[q]$ is set to true for node q if p stops hearing from a neighbor q or if a neighbor (of p) informs p about q being crashed or disconnected. Node p also maintains a variable

Algorithm 2 Eventually Perfect Failure Detector, Code for node p

```
Event ⟨ $timer\_expiry\_hb$ ⟩:
if $clock() = n \cdot T$ for $n \in \mathbb{N}$ then
    $send\langle suspect\_local[\cdot], paths[\cdot], p\rangle$ to every neighbor of $p$
        // Send heartbeat with a copy of the local suspect list and the path list

Event ⟨ $recv\langle suspect\_rcv[\cdot], path\_sets[\cdot], q\rangle$ ⟩:
if $suspect\_local[q] = true$ then    // Neighbor wrongfully suspected
    $timeout[q] := 2 \cdot (clock() - last\_contact[q])$
    $suspect\_local[q] := false$   // Stop suspecting this neighbor
$last\_contact[q] := clock()$
for all $r \notin neighbors$ do
    $hop\_from\_msg$ := Length of the shortest path in $path\_sets[r]$ not containing
                        $p$ or a node $u$, $u \neq r$ with $suspect\_rcv[u] = true$
    $hop$ := Length of the shortest path in $paths[r]$ not containing a node $u$, $u \neq r$
             with $suspect\_local[u] = true$
    if $hop\_from\_msg < hop$ then
        $suspect\_local[r] := suspect\_rcv[r]$
    for each path, $\pi \in path\_sets[r]$ that does not contain $p$ do
        $paths[r] := paths[r] \cup \{\pi \cdot p\}$    // Append new paths to the $paths[r]$ set
$suspect[\cdot] := suspect\_local[\cdot]$
Let $Sus$ be the set of all $u$ with $suspect\_local[u] = true$
for all $r \notin neighbors$ do
    if all paths in $paths[r]$ contain a $u \in Sus$ and $u \neq r$ then
        $suspect[r] := true$


Event ⟨ $timer\_expiry(q)$ ⟩:    // The timer for a neighbor expires
if $timeout[q] = clock() - last\_contact[q]$ then
    $suspect\_local[q] := true$
```

$paths[\cdot]$ which is an array of paths (sequences of node ids) between p and all other nodes in the system. Initially, $paths[q]$ is set to $\{q \cdot p\}$ for all neighboring nodes q and for all other nodes in the system. As the algorithm progresses and nodes exchange messages, p starts updating information about the simple paths to other nodes. Node p also maintains an array of booleans $suspect[\cdot]$ which stores failure information from $suspect\_local[\cdot]$ along with extra information derived from $paths[\cdot]$. Its value is set to true for nodes estimated to have crashed (using failure information from $suspect\_local[\cdot]$) or nodes that are estimated to be disconnected from p (using information from $paths[\cdot]$). If p's $suspect[q]$ variable is *true*, then we say p *suspects* q. Node p also maintains a $timeout[q]$ variable for every neighbor q which is used to detect failed neighbors. It is initially set to some integer T, which is also the time between consecutive heartbeats sent by p on each of its neighboring ADD channels.

Every node in the system sends out a heartbeat message containing the variables $suspect\_local[\cdot]$ and $paths[\cdot]$ to its neighbors every T units of time on line number 2. The integer T may be chosen differently by each node. The smaller the value of T, the faster the failure detection algorithm will converge, but if T is too small, the network may be overcrowded with heartbeat messages. The task of finding an optimum value of T is outside the scope of this work. When p receives a heartbeat message from a neighbor q (on line number 3), it records its current local clock time ($clock()$) as the $last\_contact$ value. On line number 4, p checks if q was wrongly suspected (because the $timeout[q]$ value estimate was too small), p stops suspecting q by setting $suspect\_local[q]$ to $false$ and increments its $timeout[q]$ value for q on line number 5. Then, p extracts information about the rest of the network from the message from q on line numbers 8 to 14. For all nodes r that are not neighbors of p, p calculates q's estimated distance from r and stores it in the variable hop_from_msg and calculates its own distance from r and stores it in the variable hop. On line number 11, p checks if q's estimated distance to r is shorter than p's estimated distance to r. If q is calculated to be nearer to r than p, then p adopts q's information about r on line number 12. On line number 13, node p goes through all the paths in the variable *path_sets*[·] received in the message from q and sees if p is already included in those paths. If p learns about any path π from r to q that does not include p, it adds the path $\pi \cdot p$ to its *paths*[r] set on line number 14.

On line numbers 15 to 19, p updates its *suspect*[·] variables using information from the *paths*[·] variable about nodes that are no more in p's connected component. As there are no paths from these nodes to p, information about these nodes is not received directly. On line number 18, p checks if at least one path in the *paths*[r] set has all nodes that have the *suspect_local*[·] variable set to $false$. If not, it suspects r, i.e., *suspect*[r] is set to *true* on line number 19.

Lines 20 to 22 implement a timer to detect if neighbors have crashed.

2.1.5 Proof of Correctness

To prove correctness of Algorithm 2, we need to show that the implementation satisfies **strong completeness** and **eventual strong accuracy**. Fix an execution of Algorithm 2. We describe some lemmas to prove that Algorithm 2 implements an eventually perfect failure detector for partitionable networks.

Lemma 2 shows that there is an upper bound on the inter-arrival time of heartbeats at all correct nodes from correct neighbors. Lemma 3 shows that eventually, the time-out estimates for neighbors stop changing. Lemmas 5 and 6 show that eventually all correct nodes suspect crashed neighbors and never suspect correct neighbors. Lemmas 7 and 8 show that eventually the *paths*[q] variable at a correct node p contains all the paths between p and q in the final network graph. Lemma 9, along with Theorem 10, proves eventual strong accuracy. Theorem 12 proves strong completeness. Finally, Theorem 13 shows that our algorithm implements $\Diamond P$ using Theorems 10 and 12.

We use a subscript to denote which node a variable belongs to; for example p's *suspect_local*[q] variable will be denoted as $suspect\_local_p[q]$. From here on we refer to nodes that are neighbors with respect to the initial network graph as *initial neighbors*.

2.1.5.1 Proof of Eventual Strong Accuracy

Lemma 2. *Let p be a correct node and q be a correct initial neighbor of p. Then there is an upper bound on the time that elapses between the receipt at p of two consecutive heartbeats from q.*

Proof. ADD channels guarantee that at least one in every r consecutive messages sent on a channel is received within d time. Thus the maximum time between the receipt of two consecutive heartbeat messages sent on Line 2 of our algorithm at any neighbor q of p is $(r + 1) \cdot T + d$ where r and d are the ADD channel parameters and T is a constant in p's algorithm.

□

Lemma 3. *For every correct node p, eventually the $timeout[q]$ variable at p for every initial neighbor q of p stops changing.*

Proof. Let q be an initial neighbor of p. If q is correct, then Lemma 2 implies that there is an upper bound on the inter-arrival times at p of heartbeats from q. Since $timeout_p[q]$ is only increased, eventually it reaches this upper bound and is subsequently never changed. If q crashes, then it sends only a finite number of messages to p; since p only changes $timeout_p[q]$ upon receiving a message from q, eventually it will stop changing $timeout_p[q]$.

□

Observation 4. *For a correct node p, the $suspect_p[q]$ variable for an initial neighbor q is equal to $suspect\_local_p[q]$ at all times.*

Let t^* be a time after which no more failures occur. We call the network graph after t^* the *final network graph* and denote it by $\mathbb{G}$. Let $t^{**} \geq t^*$ be a time after which no messages from failed nodes are received.

Lemma 5. *There exists some time t after which all correct initial neighbors p of a crashed node q in $\mathbb{G}$ suspect q.*

Proof. Let us assume that q crashes at time t_c. From lines 6 and 20 - 22 of our algorithm we know that the *suspect_local*$[\cdot]$ variables for initial neighbors are set only by the nodes themselves (i.e., nodes do not update information about their initial neighbors from other nodes). From Lemma 3, we know that by some time t^f, the variable $timeout_p[q]$ stops changing. So, by time $t = (\max(t^f, t^{**}) + \tau)$, where τ is the final value of $timeout_p[q]$, p sets $suspect\_local_p[q]$ to *true* permanently. By Observation 4, we know that $suspect_p[q] = suspect\_local_p[q]$. Thus after t, p suspects q. □

Lemma 6. *There exists some time t after which all correct initial neighbors p of a correct node q never suspect q.*

Proof. Lemma 2 states that there is an upper bound on the inter-arrival time of messages from q to p. Lemma 3 states that $timeout_p[q]$ eventually stops changing. Let the time at which $timeout_p[q]$ stops changing be t and the final value of $timeout_p[q]$ be τ. Thus after t, p receives a message from q within every τ time and thus, p never sets $suspect\_local_p[q]$ to *true*. By Observation 4, we know that $suspect_p[q] = suspect\_local_p[q]$, thus after t, p never suspects q.

□

Lemma 7. *For all p and q, $paths_p[q]$ is a subset of the set of all paths from q to p in the initial network graph.*

Proof. We do this proof by induction on the states of the execution. We number the states as $S_0, S_1, \cdots$, where S_0 is the initial state, and so on.

Base case: Initially in state S_0, the $paths_p[q]$ is set to contain only $\{q, p\}$ if q is a neighbor of p and $paths_p[q]$ is empty if q is not a neighbor. Thus the lemma is satisfied for the initial state.

Inductive hypothesis: For all $i \geq 0$, we assume that the lemma holds for state S_i.

Inductive Step: We show that the lemma holds for state S_{i+1}. If q is a neighbor of p, then the lemma holds as the $paths_p[q]$ variable is never updated. Suppose q is not a neighbor of p.

If the $paths_p[q]$ variable is empty then the lemma is vacuously true. If the $paths_p[q]$ variable is not empty but $paths_p[q]$ is not updated in state S_{i+1}, the lemma still holds. If p updates variable $paths_p[q]$ in S_{i+1}, it is done from information sent in a message from a neighbor of p, say r. It is clear from lines 13 and 14 in the algorithm that every node p only appends itself onto the path information that is received from neighbor r. Let $\pi_r = q \cdot q_1 \cdot \cdots \cdot r$ be an entry in $paths_r[q]$ variable at the node r before state S_{i+1}. From the inductive hypothesis, we know that π_r exists in the initial network graph. When p learns about π_r from neighbor r, p updates $paths_p[q]$ by adding the entry $\{\pi_r \cdot p\}$ to it, which is a path in the initial network graph as there is an edge from r to p.

□

Let C be a connected component in the final network graph $\mathbb{G}$. A node p is called an *initial neighbor* of C if $p \notin C$ is the neighbor of any node in C in the initial network graph .

Lemma 8. *For all p and q such that q is not an initial neighbor of p, eventually the $paths_p[q]$ variable contains all the simple paths in $\mathbb{G}$ from q to p.*

Proof. Let us assume that there exists a simple path π from q to p in $\mathbb{G}$ that is never included in $paths_p[q]$ variable at p. Let the length of this path be k and the nodes in this path be $q \cdot q_1 \cdot q_2 \cdot \cdots \cdot q_{k-1} \cdot p$. From Lemma 2, we know that each node in this path receives a message from the previous node in this path infinitely often as all nodes in this path are correct. From line number 14, we know that each of these paths is appended to the $paths_p[q_i]$ variable of each node $q_i \notin neighbors_p$. So, when p gets a message from q_{k-1} with the path $q \cdot q_1 \cdot q_2 \cdot \cdots \cdot q_{k-1}$ in it, it adds the path $q \cdot q_1 \cdot q_2 \cdot \cdots \cdot q_{k-1} \cdot p$ to its $paths_p[q]$ variable. Note that all $paths_p[q]$ end with p. To makes sure that no cycles are included in $paths_p[q]$, once p hears about a new path, it checks if the entry p is in it already (on line number 13), if so, p ignores this new path value.

□

A node p has *perfect information* about node q at time $t > t^*$ if any one of the following hold:

- If q is in p's connected component in $\mathbb{G}$, $suspect_p[q] = false$ at t.
- If q is crashed and is an initial neighbor of p's connected component C in $\mathbb{G}$, $suspect\_local_p[q] = true$ at t Note that if $suspect\_local_p[q]$ is $true$, then line numbers 15 to 19 imply that $suspect_p[q]$ is set to $true$ as well.
- If q is not in p's connected component in $\mathbb{G}$, $suspect_p[q] = true$ at t.

In the next lemma we abuse notation and say that a crashed initial neighbor q of a connected component C in the final network graph $\mathbb{G}$ is at a distance k from a correct node p in C if q has a correct initial neighbor r at distance $k-1$ from p .

Lemma 9. *Let p be a correct node in a connected component C in the final network graph $\mathbb{G}$. Let q be either a correct node in C at a distance at most k from p or let q be a crashed initial neighbor of C at a distance k from p. For all k, there exists a time t_k such that, for all $t \geq t_k$, p has perfect information about q.*

Proof. We prove this lemma by strong induction on the distance k of node p from q in $\mathbb{G}$.
Base case: $k = 1$. From Lemmas 5 and 6, we know that there exists a time when all correct initial neighbors have perfect information about q.
Inductive hypothesis: Let us assume that all nodes that are at most $k-1$ hops away from q in $\mathbb{G}$ have perfect information about q after some time t_{k-1}.
Inductive step: Let p be a node at a distance k from q in $\mathbb{G}$. If q is correct, there exists a path $q \cdot \pi \cdot p$ from q to p in $\mathbb{G}$ (since $p, q \in C$). If q is crashed, there exists a path $\pi \cdot p$ from an initial neighbor of q to p in $\mathbb{G}$ (since q is an initial neighbor of C). Let r be a node on this path $k-1$ hops away from q (note, r is an initial neighbor of p). From the inductive hypothesis we know that r has perfect information about q after time t_{k-1}. Before time t_{k-1}, r sent only a finite number of messages to p. Let $t^{\dagger}$ be the time when all messages sent before t_{k-1} all messages sent before t_{k-1} that are ever received have been received. All messages that r sends to p after time t_{k-1} have perfect information about q in them.

From Lemma 2, we know that there is an upper bound on the time between two consecutive receive events from r to p. Let this upper bound be τ. By Lemma 8, after some time t_a, all paths from q to r in $\mathbb{G}$ are appended to $paths_r[q]$ and sent to p in the variable $path\_sets[q]$. So, after time $t' = (\max\{t_{k-1}, t^{\dagger}, t_a\} + 2\tau)$, all messages p gets from r have perfect information about q and contain all paths between q and r in $\mathbb{G}$.

When p processes a message from r after t' we argue that the value of hop_from_msg for q on line number 9 is $k-1$. hop_from_msg cannot be greater than $k-1$ because the variable $path\_sets[q]$ contains $q \cdot \pi$ which is of length $k-1$. Also, hop_from_msg cannot be less than $k-1$ because r has perfect information about all nodes at a distance $k-1$ from r, and so, the estimate for hop_from_msg will discard all paths with length less than $k-1$ as they are no more available in $\mathbb{G}$. We also argue that the value of hop for q on line number 10 is greater than $k-1$. This is because, by the inductive hypothesis, p has perfect information about nodes that are $k-1$ hops away from p in $\mathbb{G}$. So, all entries in $paths_p[r]$ with length at most $k-1$ are discarded as they are correctly estimated to have at least one crashed node in them. Thus, the value of hop for q is greater than $k-1$. As a result, the 'if' condition on line number 11 is satisfied and p adopts r's information about q. By the inductive hypothesis, this information is perfect (note that p adopts only r's $suspect\_local_r[q]$ variable which currently contains perfect information about q).

We still have to show that p's $suspect_p[q]$ variable does not get set to something different (from the value set by the message from r) by a message coming from a node with wrong information about q. Let us assume that by contradiction, p gets a message from a node s that is at a distance $i > k$ from q and the value of hop_from_msg for q on line number 9 is miscalculated to be at most $k-1$. This scenario is possible only if there is a path π' in $paths_s[q]$ with $|\pi'| \leq k-1$ that is wrongly assumed to exist in $\mathbb{G}$. However, since s is at a distance i greater than k from q in $\mathbb{G}$, π' must have a crashed node in it. Let $z \in \pi'$ be the crashed node that s has wrong information about. Since z is less than k hops away from s, by the induction hypothesis, s already has perfect information about z. Thus the assumption that the value of hop_from_msg for q on line number 9 is calculated to be at most $k-1$ is incorrect and p permanently possesses perfect information about q.

□

Theorem 10 proves eventual strong accuracy.

Theorem 10. *There exists a time t_f such that for every $t > t_f$, every correct node p and every node q that is connected to p in $\mathbb{G}$, p does not suspect q at time t.*

Proof. The proof is direct from Lemma 9. □

2.1.5.2 Proof of Strong Completeness

Theorem 12 proves strong completeness.

Observation 11. *Let C be a connected component in $\mathbb{G}$. Let q be a node in $\Pi - C$. For every path π between $p \in C$ and q in the initial network graph, there exists a node r such that r is a crashed initial neighbor of C.*

Theorem 12. *There exists a time t_f such that for every $t > t_f$, every correct node p and every node q that is disconnected from p in $\mathbb{G}$, p suspects q at time t.*

Proof. Let C be the component of $\mathbb{G}$ containing p. We show that p eventually suspects every $q \in \Pi - C$. Since originally the network was a connected graph, there was a path from all q to p in the initial network graph. We separate this proof into two parts:

- q is an initial neighbor of p. In this case, the proof is direct from Lemma 5.
- q is not an initial neighbor of p. Let π be a path in $paths_p[q]$ after $paths_p[q]$ includes all the simple paths from q to p in $\mathbb{G}$. From Observation 11, we know that all paths from q to p have a crashed node r that is an initial neighbor of a node in C. From Lemma 9, we know that after some time t, p has perfect information about r. Thus, all $\pi \in paths_p[q]$ have a node r that has $suspect\_local_p[r] = true$. When p calculates

the *suspect* variable for q on line numbers 15 to 19, the if condition on line number 18 is satisfied and $suspect_p[q]$ is set to *true* on line number 19.

Now we show that this value of the *suspect* variable is not reversed. Since from Lemma 8, we know that $paths_p[q]$ includes all simple paths from q to p in $\mathbb{G}$ and thus stops changing and the information about all nodes r is never reversed, we can safely conclude that line number 18 is always satisfied henceforth and $suspect_p[q]$ always remains *true*.

□

2.1.5.3 Proof of Bounded Message Size

Theorem 13. *Our algorithm implements an eventually perfect failure detector for a partitionable network of ADD channels using bounded size messages.*

Proof. This proof is direct from Theorems 10 and 12 which prove eventual strong accuracy and strong completeness respectively.

The messages sent by a node $p \in \Pi$ have the variables $suspect\_local[\cdot]$ and $paths[\cdot]$ in them. Note that both these variables are bounded in size. The $suspect\_local[\cdot]$ variable has n booleans and so has size n bits. The $paths[\cdot]$ variable contains only simple paths between nodes. There are $O((n-1)!)$ such paths, each of which can be represented by $O(n \log n)$ bits. Thus the algorithm has messages of size at most $O(n! \log n)$ bits. Thus, the messages used in this algorithm are bounded in size. □

3. CRASH-TOLERANT REGISTER IMPLEMENTATION

3.1 Crash-Tolerant Register Implementation in Systems with Churn

As stated in Section 1.1, implementing concurrent data structures is an important service in the middleware layer of a distributed system. The classical shared read/write register is one of the most basic concurrent data structure. Many implementations [35] of concurrent data structures (queues, stacks, fetch&inc, compare&swap, etc) use registers as building blocks. In this section, we discuss our work on implementing a shared read/write register in a dynamic message-passing system that never stops changing.

Emulating a shared register can mask the intricacies of designing algorithms for asynchronous message-passing systems subject to crash failures, since it allows them to run algorithms designed for the simpler shared-memory model. Typically such emulations replicate the value of the register in multiple servers and require readers and writers to communicate with a majority of servers. The success of this approach for static systems, where the set of nodes (readers, writers, and servers) is fixed, has motivated several similar emulations for dynamic systems, where nodes may enter and leave. However, existing emulations need to assume that the system eventually stops changing for a long enough period or that the system size is bounded.

The work in this section presents the first emulation of a register supporting any number of readers and writers in a crash-prone system that can withstand nodes continually entering and leaving and imposes no upper bound on the system size. The algorithm works as long as the number of nodes entering and leaving during a fixed time interval is at most a constant

Parts of the material in this chapter are reprinted with permission from the following papers:
“Emulating a shared register in a system that never stops changing" by Hagit Attiya, Hyun Chul Chung, Faith Ellen, Saptaparni Kumar and Jennifer L. Welch, 2019. IEEE Transactions on Parallel and Distributed Systems, vol. 30, pp. 544-559, copyright [2019] by IEEE
“Simulating a shared register in an asynchronous system that never stops changing" by Hagit Attiya, Hyun Chul Chung, Faith Ellen, Saptaparni Kumar and Jennifer L. Welch, 2015. in Proceedings of 29th International Symposium on Distributed Computing, pp. 75-91, copyright [2015] by Springer

fraction of the system size at the beginning of the interval, and as long as the number of crashed nodes in the system is at most a constant fraction of the current system size. This work also includes a lower bound on the fraction of correct nodes that is strictly larger than the fraction sufficient to solve the problem in the static case.

3.1.1 Introduction

Emulating a shared read/write register is a way to mask the intricacies of designing algorithms for message-passing systems subject to crash failures, since it allows them to run algorithms designed for the simpler shared-memory model. Typically, such emulations replicate the value of the register in multiple servers and require readers and writers to communicate with a majority of servers.

The success of this approach for static systems, where the set of readers, writers, and servers is fixed, has motivated several similar emulations for *dynamic* systems, where nodes may enter and leave. Change in system composition due to nodes entering and leaving is called *churn*. However, existing emulations of atomic registers rely either on the assumption that churn eventually stops for a long enough period (e.g., DynaStore [11] and RAMBO [12]) or on the assumption that the system size is bounded (e.g., [13], [36]). See Section 3.1.2 for a detailed discussion of related work.

This work shows that it is possible to emulate a register supporting any number of readers and writers in a system subject to ongoing churn without bounding the system size, for a reasonable model of churn. This model assumes that, in any time interval of length D, the number of nodes that enter or leave the system is at most a constant fraction, α, of the number of nodes in the system at the beginning of the interval. The constant α is known to the nodes. Our emulation sacrifices atomicity when this constraint on churn is violated.

The parameter D is an upper bound, unknown to the nodes, on the delay of any message (between nodes that have not crashed). There is no lower bound on message delays. Moreover, nodes do not have real time clocks. As discussed later it is impossible to solve consensus in this model.

This model of churn is reasonable. If every node has a fixed probability of leaving in any time interval of some fixed length, then the expected number of nodes that leave in such an interval is a fixed fraction of the total number of nodes in the system at the beginning of the interval. Moreover, as the system size grows, the allowable number of changes grows. (See [37] for a discussion of churn behavior in practice.)

In addition to churn, our algorithm tolerates nodes that fail by crashing. In the preliminary version of this work [16], we assumed that the number of nodes that crash is bounded above by a fixed constant, f, independent of the system size. That assumption is quite restrictive as the system size can grow arbitrarily large. Here, we replace this restriction with the more flexible requirement that, at all times, the number of crashed nodes is at most a constant fraction, Δ, of the current system size. Now as the system size grows, the number of crashed nodes can grow as well. The constant Δ is known to all nodes.

Our algorithm is called CCReg, for *Continuous Churn Register*. It is intuitive, combining the simple static algorithm for multiple readers and multiple writers outlined above with a joining protocol and careful estimations of the number of nodes from which replies should be received for joining, reading, and writing. In order to join, a newly entered node announces its entry, waits to receive sufficiently many acknowledgments, and then announces it has joined, which marks the termination of its join operation. Once a node has joined, it can perform reads and writes. A node leaves the system by announcing its departure. Each node maintains a set of changes to the composition of the system, based on the announcements of nodes entering, joining and leaving. This information is also propagated through appropriate echo messages and by having each node append the set of changes it has seen to its messages that echo enter announcements. Each node keeps track of the set of nodes that it believes are *present* in the system (i.e., have entered but not left) and those that it believes are *members* of the system (i.e., have joined but not left).

When a node p first receives an acknowledgment of its entrance announcement from a node that has already joined, p calculates the number of acknowledgments it needs to receive before joining. To guarantee that information about the system composition propagates properly, this number must be sufficiently large to ensure that at least one acknowledgment is from a node that has been in the system for a sufficiently long time. The number must also be small enough to ensure that p will eventually receive enough acknowledgments to complete the join procedure. The node p sets this number to a carefully-chosen fraction, γ, of the number of nodes that it believes are present in the system.

As in [10] and [11], read and write operations wait for replies from a certain number of nodes. To guarantee that information about a written value is propagated properly, this number must be large enough to ensure that the value has been received by enough nodes. It must also be large enough to ensure that at least one reply to a read operation is from a node with an up-to-date value. This number must be small enough to ensure that the operation terminates. In the static case, a majority of the nodes suffices. In our algorithm, node p sets the number to a carefully-chosen fraction, β, of the number of nodes that p believes to be members of the system.

The contribution of this work is a proof of existence of a crash-resilient algorithm that tolerates ongoing churn with no upper bound on the system size. As we discuss in Section 3.2.4, the system parameters α and Δ and the algorithm parameters γ and β must satisfy certain constraints in order for our algorithm to work. We show that there are values for the parameters that satisfy the constraints. Further work is needed to find algorithms that work under less restrictive constraints to, say, tolerate a larger churn rate or larger failure fraction.

In all the consistent sets of parameter values that we have identified, the failure resilience, Δ, is at most $1/3$, which is worse than $1/2$, which can be achieved in the static case. We prove that worse failure resilience is an unavoidable consequence of tolerating churn, by showing the problem is unsolvable unless Δ is at least $1/(\alpha + 2)$.

3.1.2 Related Work

A simple emulation of a single-writer, multi-reader register in an asynchronous static network was presented by Attiya, Bar-Noy and Dolev [9]. Their paper also shows that it is impossible to emulate an atomic register in an asynchronous system if at least half of the nodes in the system can be faulty. It was followed by extensions that reduce complexity [38, 39, 40, 41], support multiple writers [10], or tolerate Byzantine failures [42, 43, 44, 45]. To optimize load and resilience, the simple majority quorums used in these papers can be replaced by other, more complicated, quorum systems (e.g., [46, 47]).

A survey of emulations of an atomic multi-writer, multi-reader register in a dynamic system with churn appears in [48]. We compare our results with RAMBO [12], DynaStore [11] and the results of Baldoni et al. [36]. It is important to note that the system models in all these papers are very different from one another and are thus, in some sense, incomparable.

The first such emulation was RAMBO [12]. Here, the notion of churn is abstracted by defining a sequence of *quorum configurations*. Each quorum configuration consists of a set S of nodes (which are called members) plus sets of read-quorums and write-quorums, each of which is a subset of S. The system supports *reconfiguration*, in which an older quorum configuration is replaced by a newer one. The reconfiguration protocol handles quorum configuration changes to install new quorum systems. Reconfiguration is done in two parts: first, a member proposes a new quorum configuration. Second, these proposed configurations are reconciled by running an eventually-terminating distributed consensus algorithm (a version of the Paxos algorithm [49]) among the members of the current quorum configuration. RAMBO requires intermittent periods of synchrony for the consensus to terminate. Reconfigurations can occur concurrently with reads and writes. The read and write operations are similar to those in [9, 10]. The model does not differentiate between nodes that crash and nodes that leave the system. The algorithm guarantees atomicity of operations for all executions, even when there are arbitrary crashes (or leaves) and message loss. However, liveness of reconfigurations is only ensured during periods when the system is sufficiently well-behaved with respect to synchrony, message loss, and churn. For liveness of reads and writes, the authors assume that a configuration has to remain viable for $11D$ time after it is installed, to allow sufficient time for a phase of an operation to complete. They also assume that reconfigurations are installed at least $13D$ time apart. If reconfigurations occur more frequently, read and write operations might be delayed each time a new configuration is discovered. Liveness of reads and writes does not depend on synchrony.

DynaStore [11] emulates an atomic multi-writer, multi-reader register in a dynamic system, without using consensus. The set of nodes that are in the system is called a *view*. The nodes start with some default initial *view*. The algorithm supports *read*/*write* operations and *reconfig*. Reads and writes are similar to those in [9, 10], with a read-phase followed by a write-phase. The nodes in the current view can propose the addition and removal of other nodes using the *reconfig* subroutine. Reconfig starts with a phase in which information about the new view is sent to a majority of nodes in the old view, followed by a read phase and a write phase, which are performed using the old view. DynaStore ensures atomicity for all executions. To ensure liveness of read/write operations, the algorithm makes two assumptions. First, at any point in time, the set of crashed nodes and the nodes whose removals are pending (via reconfig) is a minority of the current view and of any pending future views. Second, it assumes that only a finite number of reconfiguration requests occur (i.e., churn eventually stops).

Baldoni et al. [36] study a model with upper and lower bounds on the system size, known to the nodes. In their model, churn never stops and at most a constant fraction of nodes enter and leave periodically. They implement a regular register in an eventually synchronous system. A *join_register* module ensures nodes join with sufficient knowledge about the system. Its *read* and *write* protocols are similar to those in [9, 10]. The emulation can be shown to violate regularity if the churn assumption is violated, with an argument like the one at the end of Section 3.2.5.

Table 3.1 compares the results of [12, 11, 36] with our algorithm, considering: the consis-

Algorithm	Consistency Condition	Synchrony Assumption	Tolerates Continuous Churn	Violates Consistency Condition If Churn Assumption Violated	Failure Model
RAMBO [12]	Atomicity	Intervals with known upper bound on message delays; real-time clocks	No, liveness of reads and writes needs periods of quiescence	No	No difference between leaves and crashes
DynaStore [11]	Atomicity	No bounds on message delays	No, liveness of read and writes requires the number of reconfigurations to be finite	No	Crash failures are different from leaves
Baldoni et al. [36]	Regularity	Eventual known upper bound on message delays	Yes	Yes	No difference between leaves and crashes
CCReg (This work)	Atomicity	Unknown upper bound and no lower bound on message delays; no real-time clocks	Yes	Yes	Crash failures are different from leaves

Table 3.1: Summary of our algorithm and related algorithms.

1

tency condition, the level of synchrony needed for the correctness of the algorithms, whether the algorithm requires periods without churn, whether the consistency condition is violated if the churn assumption is violated, and whether failures and leaves are modeled as different events.

Baldoni et al. [13] prove that it is impossible to emulate a regular register when there is no upper bound on message delay. In this case, it does not help for nodes to announce when they are leaving, since messages containing such announcements can be delayed for an arbitrarily long time. Thus, *a node leaving is essentially the same as a crash.* Their proof works by considering scenarios in which at least half of the nodes fail. they invoke the lower bound in [9], which shows that emulating a register is impossible unless fewer than half the nodes are faulty. Their proof can be adapted to hold when there is an unknown upper bound, D, on the message delay and half the nodes can be replaced during any time interval of length D, provided that nodes are not required to announce when they leave. Thus, in our model, there must be an upper bound on the fraction of nodes that can crash during any time interval of length D. Also, it is necessary that either nodes announce when they leave or there is an upper bound on the fraction of nodes that can leave during this time interval.

In [50] and [11], it is claimed that termination of operations cannot be guaranteed unless the churn eventually stops. This claim does not contradict our result due to differences in the churn models, since their proofs rely on many nodes entering and leaving during a short time period, a behavior that is not allowed in our model. One of the contributions of this work is to point out that by making different, yet still reasonable, assumptions on churn it is possible to get a solution with different, yet still reasonable, properties and, in particular, to overcome the prior constraint that churn must stop to ensure termination of operations. That is, we are suggesting a different point in the solution space.

3.1.3 Research Goals

We have two main research goals in this area. First, we want to create an algorithm to implement an atomic MRMW (Multi Reader Multi Writer) register in a system that never stops changing and can change size. The second goal is to prove a lower bound that shows that unlike static systems, majority correctness is not enough to implement a SRSW (Single Reader Single Writer) register in systems with churn.

3.1.4 System Model and Problem Statement

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

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

A *step* of a node p is a 5-tuple (s', T, m, R, s) where s' is the old state, T is the triggering event, m is the message to be sent, R is a response ($\textsc{Return}_p(v)$, $\textsc{Ack}_p$, or $\textsc{Joined}_p$) or $\perp$, and s is the new state. The values of m, R and s are determined by a transition function applied to s' and T. $\textsc{Return}_p$ is the response to $\textsc{Read}_p$, $\textsc{Ack}_p$ is the response to $\textsc{Write}_p$, and $\textsc{Joined}_p$ is the response to $\textsc{Enter}_p$. If T is $\textsc{Crash}_p$, then m is $\perp$ and R is $\perp$.

A *view* of a node p is a sequence of steps such that:

- the old state of the first step is an initial state;

- the new state of each step equals the old state of the next step;
- if the old state of the first step is s_p^i, then no $\textsc{Enter}_p$ event occurs;
- if the old state of the first step is s_p^l, then the triggering event in the first step is $\textsc{Enter}_p$ and there is no other occurrence of $\textsc{Enter}_p$; and
- at most one of $\textsc{Crash}_p$ and $\textsc{Leave}_p$ occurs and if so, it is in the last step.

In our model, a node that leaves the system cannot re-enter with the same id. It can, however, re-enter with a new id. Likewise, a node that crashes does not recover. A node that crashes and recovers, but loses its state, can re-enter with a new id. Because nodes cannot measure time, a node that crashes and recovers, retaining its state, can be treated as if no crash occurred.

Time is represented by nonnegative real numbers. A *timed view* is a view whose steps occur at nondecreasing times. If a view is infinite, the times at which its steps occur must increase without bound. Given a timed view of a node, if (s', T, m, R, s) is the step with the largest time less than or equal to t, then s is the *state* of that node *at time* t. A node p is said to be *present* at time t if it entered the system (i.e., its first step has time at most t) but has not left (i.e., $\textsc{Leave}_p$ does not occur at or before t). The number of nodes that are present at time t is denoted by $N(t)$. A *crashed* node (i.e., a node for which $\textsc{Crash}_p$ occurs at or before t) is still considered to be present. A node is said to be *active* at time t if it is present and not crashed at t. A node p is said to be a *member* at time t if it has *joined* the system (i.e., $p \in S_0$ or $\textsc{Joined}_p$ occurs at or before t) but has not left (i.e., $\textsc{Leave}_p$ does not occur at or before t). Note that, at any time t, the members are a subset of the present nodes and it is possible for some members to be crashed

If a message m sent at time t is received by a node at time t', then the *delay* of this message is $t' - t$. This encompasses transmission delay as well as time for for handling the message at both the sender and receiver. Let $D > 0$ denote the *maximum message delay* that can occur in the system. Let $\alpha > 0$ and $0 < \Delta \leq 1$ be real numbers that denote the *churn rate* and *failure fraction*, respectively. The parameters α and Δ are known to the nodes, but D is not.

An *execution* e is a possibly infinite set of timed views, one for each node that is ever present in the system, that satisfies the following eight assumptions.

A1: The first step of each node $p \in S_0$ occurs at time 0 and the first step of each other node occurs after time 0.

A2: Every sent message has at most one matching receipt at each node and every message receipt has exactly one matching message send.

A3: If a message m is sent at time t and node q is active throughout $[t, t+D]$ (i.e., q enters by time t and does not leave or crash by time $t+D$), then q receives m. The delay of every received message is in $(0, D]$.

A4: Messages from the same sender are received in the order they are sent (i.e., if node p sends message m_1 before sending message m_2, then no node receives m_2 before it receives m_1). This can be achieved by tagging each message with the id of its sender and a sequence number.

A5: For all times $t > 0$, the number of $\textsc{Enter}$ and $\textsc{Leave}$ events in $[t, t+D]$ is at most $\alpha \cdot N(t)$.

A6: For all times $t \geq 0$, the number of crashed nodes at time t is at most $\Delta \cdot N(t)$.

A7: If $\textsc{Read}_p$ or $\textsc{Write}_p$ invocation occurs at time t, then p has already joined but has not left or crashed.

A8: At each node p, no $\textsc{Read}_p$ or $\textsc{Write}_p$ occurs until there have been responses to all previous $\textsc{Read}_p$ and $\textsc{Write}_p$ invocations.

Assumption A1 states that there is a nonempty finite set of nodes that are initially members. Assumptions A2 through A4 model a reliable broadcast communication service that provides nodes with a mechanism to send the same message to all nodes in the system. Sending a message to a single recipient can be accomplished by broadcasting the message and indicating the intended recipient so that others will ignore the message. Assumption A5 bounds the churn and Assumption A6 bounds the number of failures. Assumptions A7 and A8 ensure that operations are only invoked by active members and, at any time, at most one operation is pending at each node.

We consider an algorithm to be *correct* if every execution of the algorithm satisfies the following conditions:

- For every node $p \in S_0$, $\textsc{Joined}_p$ does not occur. For every node $p \notin S_0$, if $\textsc{Enter}_p$ occurs, then at least one of $\textsc{Leave}_p$, $\textsc{Crash}_p$, or $\textsc{Joined}_p$ occurs (i.e., every node that enters the system and remains active eventually joins.

- In the view of each node p, ignoring message-receipt events, each $\textsc{Read}_p$ or $\textsc{Write}_p$ is immediately followed by either $\textsc{Leave}_p$, $\textsc{Crash}_p$, or a matching response ($\textsc{Return}_p$ or $\textsc{Ack}_p$). Moreover, each $\textsc{Return}_p$ or $\textsc{Ack}_p$ is immediately preceded by a matching invocation ($\textsc{Read}_p$ or $\textsc{Write}_p$).

- The read and write operations are atomic [51, 52, 53]: there is an ordering of all completed reads and writes and some subset of the uncompleted writes such that every read returns the value of the latest preceding write (or the initial value of the register if there is no preceding write) and, if an operation op_1 finishes before another operation op_2 begins, then op_1 occurs before op_2 in the ordering.

It is the responsibility of the algorithm to complete joins, complete read and write operations, and choose the right values for the reads, as long as Assumptions A1–A8 are satisfied.

Although our model places an upper bound on message delays, it does not place any lower bound on the message delays or on local computation times. Moreover, nodes cannot access clocks to measure the passage of real time. Consequently, the well-known consensus problem is unsolvable in our model, just as it is unsolvable in a model with no upper bound on message delays [5]. In the *consensus* problem, every node has an input, and every nonfaulty node must eventually decide on an output such that all outputs are the same and, if all inputs are the same, then this common output equals the common input.

3.1.4.1 Impossibility of Consensus

Theorem 14. *It is impossible to solve consensus in our system model, even with just one crash and no churn.*

Proof. Assume, by way of contradiction, there is an algorithm A that solves consensus in our model, with delays in $(0, D]$ but no churn and at most one crash. For simplicity, let $D = 1$. Consider an execution e of A in the classic asynchronous model with no upper bound on message delays, no churn, and at most one crash.

A priori, there is no guarantee that e solves consensus, as A is only guaranteed to work correctly in our model, but the nodes exhibit some kind of behavior in e. Order the events in e by their times, breaking ties arbitrarily. Let e' be the result of changing the time of the i-th event in e to $1 - 2^i$, for all $i \geq 1$. For all positive integers k, let e'_k consist of the first k steps of e'. Since algorithm A solves consensus in our model, there exists a positive integer k such that, in e'_k, all nonfaulty nodes have terminated and decided the same output. The view of every node in e'_k is the same as its view in the first k steps of e. Thus, all nonfaulty nodes have terminated and decided the same output within the first k steps of e. In other words, A solves consensus in the classic asynchronous model, which contradicts [5]. □

3.1.5 Lower Bound on Crash-Resilience

In this section we prove that strictly more than a majority of the nodes must be nonfaulty in order to emulate an atomic read-write register in a system with churn. Specifically, we show that the failure fraction Δ must be less than $\frac{1}{\alpha+2}$, where the churn rate α is a nonnegative rational number.

Theorem 15. *It is impossible to emulate an atomic read-write register in a dynamic system with churn rate α, if the failure fraction Δ is at least $\frac{1}{\alpha+2}$.*

Proof. Assume, by way of contradiction, there is an algorithm that emulates an atomic register with $\Delta \geq \frac{1}{\alpha+2}$.

Suppose that $\alpha = u/v$ where u and v are positive integers. Let $N(0)$ be an integer that is divisible both by $u + 2v$ and v. Then $\frac{N(0)}{\alpha+2}$, $\frac{N(0)(\alpha+1)}{\alpha+2}$, and $\alpha \cdot N(0)$ are all positive integers.

Initially, there are $N(0)$ nodes that are initially in the system and members. We partition these nodes into two disjoint sets, S_1 and S_2, consisting of $\frac{N(0)}{\alpha+2}$ and $\frac{N(0)(\alpha+1)}{\alpha+2}$ nodes, respectively. First, we consider two different executions, e_1 and e_2, starting from this initial configuration.

Execution e_1:

- No Enter or Leave events occur.
- All the nodes in S_1 crash before sending any messages. No other crashes occur.
- A node $p \in S_2$ invokes an operation $\textsc{Write}_p(1)$ on the emulated register.
- All messages other than those sent to crashed nodes have delay D.
- The write invoked by p completes by some time t_w.

Since $|S_1| = \frac{N(0)}{\alpha+2} \leq \Delta \cdot N(0)$, crashing all the nodes in S_1 is allowed.

Execution e_2:

- At some time t_e, where $0 < t_e < t_w$, a set, S_3, of $\alpha \cdot N(0)$ nodes enter the system.
- All nodes in S_2 crash at some time $t_c > t_e$, before sending any messages.
- A node $q \in S_1$ invokes an operation $\textsc{Read}_q$ at some time after t_w.
- All messages other than those sent to crashed nodes have delay D.
- The read invoked by q completes by some time t_r and returns the initial value, 0, of the register.

Note that $|S_3| = \alpha \cdot N(0)$ is the maximum number of nodes that are allowed to enter at time t_e. Since $N(t_c) = |S_1| + |S_2| + |S_3| = (\alpha + 1)N(0)$, the number of crashed nodes at time t_c is $|S_2| = \frac{(\alpha+1)N(0)}{\alpha+2} = \frac{N(t_c)}{\alpha+2} \leq \Delta \cdot N(t_c)$, which is also allowed.

Finally, we construct a new execution e_3 from e_1 and e_2.

Execution e_3:

- Create a set of timed views by merging the timed views of nodes in S_2 from e_1, which contains a write of 1, and the timed views of nodes in $S_1 \cup S_3$ from e_2, which contains a read of 0. Note that there are no crashes in these timed views.
- Truncate each timed view so that it consists of all steps with associated time at most t_r, i.e. just after the read finishes.
- Extend the truncated views by delivering all pending messages. This includes all messages sent by time t_r in e_1 from nodes in S_1 to nodes in S_2 and all messages sent by time t_r in e_2 from nodes in S_2 to nodes in $S_1 \cup S_3$. These messages are all delivered at some time $t_f > t_r$. Note that some of these messages might have delay greater than D. Let V_3 denote the resulting set of timed views.
- Multiply the real time of every event in every sequence of V_3 by $\min\{\frac{D}{t_f}, 1\}$. This shrinks the delays to ensures that they are all at most D.

- Extend the sequences by receiving and sending messages, where each message has delay D, but without including any new occurrences of Enter, Read, Write, Crash, or Leave. From the code, the algorithm will eventually finish.

We verify that e_3 is an execution:

A1: Sets S_1 and S_2 are present in the system at time 0.

A2–A4: Consider any message m' in e_3 that corresponds to a message m in V_3 sent before time t_f. Since m is sent at or after time 0 and is received at every node at or before time t_f, its delay in V_3 is at most t_f. By construction, the delay of m' in e_3 is at most $t_f \cdot \min\{D/t_f, 1\}$, which equals t_f if $t_f \leq D$ and equals D if $t_f > D$. Thus the delay of m' is at most D. All other messages in e_3 have delay at most D by construction. Assumptions A2, A3, and A4 follow from this and the fact that they hold in e_1 and e_2.

A5: There are $\alpha \cdot N(0)$ Enter events in $[0, D]$ and no other Enter events. There are no Leave events.

A6: No nodes crash.

A7: The read is invoked by p after p joins and the write by q is invoked after q joins.

A8: There is only one read and only one write and they are invoked at different nodes.

In e_3, the read operation by q returning 0 starts after the operation writing 1 by p completes, which violates atomicity. □

If $\alpha = 0$, then our lower bound reduces to the requirement that the failure fraction be less than 1/2, which is well-known for the static case and is achievable [9]. If $\alpha > 0$, then the failure fraction has to be even smaller than what is sufficient in the static case and it must decrease as the churn rate increases.

Looking carefully at the proof of Theorem 15, we see that the result holds even for the emulation of a safe register [51, 52], which satisfies a weaker consistency condition, even if only one reader and one writer are supported, even if nodes never leave, and even if there is a finite upper bound on the total number of nodes.

3.1.6 The CCReg Algorithm

In our algorithm, nodes run *client* (*reader* or *writer*) threads and *server* threads. Each node runs exactly one server thread, at most one reader thread, and at most one writer thread. We assume that the code segment that is executed in response to each event executes without interruption.

The algorithm combines a mechanism for tracking the composition of the system, with a simple algorithm, very similar to [10], for reading and writing the register, which associates a unique timestamp with each value that is written. A timestamp is a pair that consists of two values: a sequence number (num) and a node id (w_id) and these (num, w_id) pairs are ordered lexicographically. Below, the local variables of node p are subscripted with p; e.g., v_p refers to node p's local variable v.

In order to track the composition of the system (Algorithm 10), each node p maintains a set of events, $Changes_p$, concerning the nodes that have entered the system. When an Enter$_q$ event occurs, q adds *enter*(q) to $Changes_q$ and broadcasts an enter message requesting information about prior events. When a node p finds out that q has entered the system, either by receiving this message or by learning indirectly from another node, it adds *enter*(q) to $Changes_p$. When q has received sufficiently many messages in reply to its request, it knows relatively accurate information about prior events and the value of the register. The fraction γ is used to calculate the number of messages that should be received before joining (stored in the $join\_bound$ local variable), based on the size of the $Present$ set. Setting γ is a key challenge in the algorithm as setting it too small might not propagate updated information, whereas setting it too large might not guarantee termination of the join.

When the required number of replies to the enter message sent by q are received, q adds *join*(q) to $Changes_q$, sets its is_joined_q flag to *true*, outputs the response Joined$_p$, and broadcasts a message saying that it has joined. When p finds out that q has joined, either by receiving this message or by learning indirectly from another node, it adds *join*(q) to

$Changes_p$. When a $\textsc{Leave}_q$ event occurs, q broadcasts a leave message and halts. When p finds out that q has left the system, either by receiving this message or by learning indirectly from another node, it adds $leave(q)$ to $Changes_p$.

When a node p receives an enter message from a node q, it responds with an enter-echo message containing $Changes_p$, its current estimate of the register value (together with its timestamp), is_joined_p (indicating whether p has joined yet), and the id q. When q receives an enter-echo in reply (i.e., that ends with q), it increments its *join-counter*. The first time q receives such an enter-echo from a joined node, it computes $join\_bound$, the number of enter-echo messages it needs to get before it can join.

Once a node has joined, its reader and writer threads can handle read and write operations. Initially, $Changes_p = \{enter(q) \mid q \in S_0\} \cup \{join(q) \mid q \in S_0\}$, if $p \in S_0$, and $\emptyset$ otherwise. A node p also maintains the set $Present_p = \{q \mid enter(q) \in Changes_p \wedge leave(q) \notin Changes_p\}$ of nodes that p considers as present, i.e., nodes that have entered, but have not left, as far as p knows. The client at node p maintains the derived variable $Members_p = \{q \mid join(q) \in Changes_p \wedge leave(q) \notin Changes_p\}$ of nodes that p considers as members, i.e., nodes that have joined but not left.

Client threads treat read and write operations in a similar manner (Algorithm 4). Both operations start with a read phase, which requests the current value of the register, using a query message, followed by a write phase, using an update message. A write operation broadcasts the new value it wishes to write, together with a timestamp, which consists of a sequence number that is one larger than the largest sequence number it has seen and its id that is used to break ties. A read operation just broadcasts the value it is about to return, keeping its sequence number. As in [9], write-back is needed to ensure the atomicity of read operations. Both the read phase and the write phase wait to receive sufficiently many reply messages. The fraction β is used to calculate the number of messages that should be received (stored in the rw_bound local variable) based on the size of the $Members$ set, for the operations to terminate. Setting β is also a key challenge in the algorithm as setting it

Algorithm 3 CCREG—Common code managing the churn, for node p.

```
Local Variables:
$is\_joined$   // Boolean to check if $p$ has joined the system; initially $false$
$join\_bound$   // if non-zero, the number of enter-echo messages $p$ should receive before joining; initially 0
$join\_counter$   // the number of enter-echo messages received so far; initially 0
$Changes$   // set of ENTER, LEAVE, and JOINED events known by $p$;
            // initially $\{enter(q) \mid q \in S_0\} \cup \{join(q) \mid q \in S_0\}$, if $p \in S_0$, and $\emptyset$, otherwise
$val$   // latest register value known to $p$; initially $\perp$
$num$   // sequence number of latest value known to $p$; initially 0
$w\_id$   // id of node that wrote latest value known to $p$; initially $\perp$

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

When ENTER$_p$ occurs:
add $enter(p)$ to $Changes$
bcast ⟨"enter", $p$⟩

When RECEIVE$_p$⟨"enter", $q$⟩ occurs:
add $enter(q)$ to $Changes$
bcast ⟨"enter-echo", $Changes$, $(val, num, w\_id)$, $is\_joined$, $q$⟩

When RECEIVE$_p$⟨"enter-echo", $C$, $(v, s, i)$, $j$, $q$⟩ occurs:
if $(s, i) > (num, w\_id)$ then
    $(val, num, w\_id) := (v, s, i)$
$Changes := Changes \cup C$
if $\neg is\_joined \wedge (p = q)$ then
    if $(j = true) \wedge (join\_bound = 0)$ then
        $join\_bound := \gamma \cdot |Present|$
    $join\_counter{+}{+}$
    if $join\_counter \geq join\_bound > 0$ then
        $is\_joined := true$
        add $join(p)$ to $Changes$
        generate JOINED$_p$ response
        bcast ⟨"joined", $p$⟩

When RECEIVE$_p$⟨"joined", $q$⟩ occurs:
add $join(q)$ to $Changes$
add $enter(q)$ to $Changes$
bcast ⟨"joined-echo", $q$⟩

When RECEIVE$_p$⟨"joined-echo", $q$⟩ occurs:
add $join(q)$ to $Changes$
add $enter(q)$ to $Changes$

When LEAVE$_p$ occurs:
bcast ⟨"leave", $p$⟩
halt

When RECEIVE$_p$⟨"leave", $q$⟩ occurs:
add $leave(q)$ to $Changes$
bcast ⟨"leave-echo", $q$⟩

When RECEIVE$_p$⟨"leave-echo", $q$⟩ occurs:
add $leave(q)$ to $Changes$
```

too small might not return/update correct information from/to the register, whereas setting it too large might not guarantee termination of the reads and writes.

A client thread maintains a sequence number, *tag*, incremented at the beginning of the read phase and identifying replies belonging to its current read or write operation.

The server thread is simple (Algorithm 8). Each node uses the variable *val* to store the latest value of the register it knows about, and the variables num and w_id to store that value's associated timestamp as an ordered pair (num, w_id). When the server receives an update message with a larger timestamp, it updates the value and the timestamp. When a server receives a query, it responds with the value and its timestamp.

Algorithm 4 CCReg—Client code, for node p.

```
Local Variables:
$temp$    // temporary storage for the value being read or written; initially 0
$tag$    // used to uniquely identify read and write phases of an operation; initially 0
$rw\_bound$    // the number of replies/acks $p$ should receive before finishing a read/write phase; initially 0
$rw\_counter$    // the number of replies/acks received so far for a read/write phase; initially 0
$rp\_pending$    // Boolean indicating whether a read phase is in progress; initially $false$
$wp\_pending$    // Boolean indicating whether a write phase is in progress; initially $false$
$read\_pending$    // Boolean indicating whether a read is in progress; initially $false$
$write\_pending$    // Boolean indicating whether a write is in progress; initially $false$
Derived Variable:
$Members = \{q \mid join(q) \in Changes \wedge leave(q) \notin Changes\}$

When READ$_p$ occurs:
$read\_pending := true$
call BeginReadPhase()

When WRITE$_p(v)$ occurs:
$write\_pending := true$
$temp := v$
call BeginReadPhase()

Procedure BeginReadPhase()
$tag{+}{+}$
bcast ⟨"query", $tag$, $p$⟩
$rw\_bound := \beta \cdot |Members|$
$rw\_counter := 0$
$rp\_pending := true$

When RECEIVE$_p$⟨"reply", $(v, s, i), rt, q$⟩ occurs:
if $rp\_pending \wedge (rt = tag) \wedge (q = p)$ then
    if $(s, i) > (num, w\_id)$ then
        $(val, num, w\_id) := (v, s, i)$
    $rw\_counter{+}{+}$
    if $rw\_counter \geq rw\_bound$ then
        $rp\_pending := false$
        call BeginWritePhase()

Procedure BeginWritePhase()
if $write\_pending$ then
    $val := temp$
    $num{+}{+}$
    $w\_id := p$
if $read\_pending$ then
    $temp := val$
bcast ⟨"update", $(temp, num, w\_id)$,$tag, p$⟩
$rw\_bound := \beta \cdot |Members|$
$rw\_counter := 0$
$wp\_pending := true$

When RECEIVE$_p$⟨"ack", $wt, q$⟩ occurs:
if $wp\_pending \wedge (wt = tag) \wedge (q = p)$ then
    $rw\_counter{+}{+}$
    if $rw\_counter \geq rw\_bound$ then
        $wp\_pending := false$
        if $read\_pending$ then
            $read\_pending := false$
            generate RETURN($temp$) response
        if $write\_pending$ then
            $write\_pending := false$
            generate ACK response
```

Algorithm 5 CCReg—Server code, for node p.

```
When Receive$_p$⟨ "update", $(v, s, i), wt, q$⟩ occurs:
if $(s, i) > (num, w\_id)$ then
    $(val, num, w\_id) := (v, s, i)$
if $is\_joined$ then
    bcast ⟨"ack", $wt, q$⟩
bcast ⟨"update-echo", $(val, num, w\_id)$⟩
When Receive$_p$⟨"query", $rt$, $q$⟩ occurs:
if $is\_joined$ then
    bcast ⟨"reply", $(val, num, w\_id), rt, q$⟩

When Receive$_p$⟨"update-echo", $(v, s, i)$⟩ occurs:
if $(s, i) > (num, w\_id)$ then
    $(val, num, w\_id) := (v, s, i)$
```

The correctness of CCReg relies on the system parameters α, Δ, and N_{min} satisfying the following constraints, for some choice of algorithm parameters β and γ:

$$\alpha \leq 1 - 2^{-1/4} \approx 0.159 \tag{3.1}$$

$$1 < \left((1-\alpha)^3 - \Delta(1+\alpha)^3\right) N_{min} \tag{3.2}$$

$$\gamma \geq \frac{1}{N_{min}(1-\alpha)^3} + (1+\Delta)\frac{(1+\alpha)^3}{(1-\alpha)^3} - 1 \tag{3.3}$$

$$\gamma \leq \frac{(1-\alpha)^3}{(1+\alpha)^3} - \Delta \tag{3.4}$$

$$\beta \leq (1+\alpha)\left(\frac{(1-\alpha)^3}{(1+\alpha)^3} - \Delta\right) \tag{3.5}$$

$$\beta > \frac{(1+\alpha)^5 - 1}{(1-\alpha)^4} \tag{3.6}$$

$$\beta > \frac{(1+\Delta)(1+\alpha)^3 - (1-\alpha)^3 + 1}{(2+2\alpha+\alpha^2)(1-\alpha)^2(1+\alpha)^{-2}} \tag{3.7}$$

Constraint (3.1) is an upper bound on the churn rate and is used in Lemma 17 to ensure that not too many nodes can leave the system in an interval of length $4D$. Constraint (3.2) is a lower bound on the minimum system size. It is used in the proof of Lemma 18 to ensure that at least one node is in the system throughout an interval of length $3D$ encompassing the time a node enters, thus ensuring that the newly entered node successfully terminates its

system parameters			algorithm parameters	
churn rate (α)	failure fraction (Δ)	minimum system size (N_{min})	*join_bound* fraction (γ)	*rw_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

Table 3.2: Values for the CCReg parameters that satisfy constraints (3.1) to (3.7).

joining protocol. Constraint (3.3) ensures that the *join_bound* fraction, γ, is large enough such that updated information about the system is obtained by an entered node before it joins the system. Constraint (3.4) ensures that γ is small enough such that for all entered nodes, a join operation terminates if the entered node does not leave or crash. Constraint (3.5) ensures that the *rw_bound* fraction, β, is small enough such that termination of read and writes is guaranteed. Constraints (3.6) and (3.7) ensure that β is large enough such that atomicity is not violated by read and write operations. Table 3.3 gives a few sets of values for which the above constraints are satisfied. Both α and Δ must be small: once α is larger than 0.04, no failures can be tolerated.

3.1.7 Correctness Proof

We will show that CCReg satisfies the three properties listed at the end of Section 3.1.4. Lemmas 16 through 23 are used to prove Theorem 24, which states that every node eventually joins, provided it does not crash or leave. Lemmas 26 through 51 are used to prove Theorem 52, which states that every operation invoked by a node that remains active eventually completes. Lemmas 55 through 35 are used to prove Theorem 37, which states that atomicity is satisfied.

Consider any execution.

3.1.7.1 Proof that Join Protocol Terminates

We begin by bounding the number of nodes that enter during an interval of time and the number of nodes that are present at the end of the interval, as compared to the number present at the beginning. (The proof is in the supplementary material.)

Lemma 16. *For all $i \in \mathbb{N}$ and all $t \geq 0$, at most $((1+\alpha)^i - 1)N(t)$ nodes enter during $(t, t+Di]$ and $(1-\alpha)^i N(t) \leq N(t+Di) \leq (1+\alpha)^i N(t)$.*

Proof. The proof is by induction on i. For $i = 0$ and all $t \geq 0$, $(t, t+Di]$ is empty, and hence, $0 = ((1+\alpha)^i - 1)N(t)$ nodes enter during this interval and

$$N(t+iD) = N(t) = (1+\alpha)^i N(t) = (1-\alpha)^i N(t).$$

Now let $i \geq 0$ and $t \geq 0$. Suppose at most $((1+\alpha)^i - 1)N(t)$ nodes enter during $(t, t+Di]$ and $(1-\alpha)^i N(t) \leq N(t+Di) \leq (1+\alpha)^i N(t)$.

Let $e \geq 0$ and $\ell \geq 0$ be the number of nodes that enter and leave, respectively, during $(t+Di, t+D(i+1)]$. By Assumption A5, $e + \ell \leq \alpha N(t+Di)$, so $e, \ell \leq \alpha N(t+Di) \leq$

$\alpha(1+\alpha)^i N(t)$. The number of nodes that enter during $(t, t+D(i+1)]$ is at most

$$\begin{aligned}((1+\alpha)^i - 1)N(t) + e \leq ((1+\alpha)^i - 1)N(t) + \alpha(1+\alpha)^i N(t)\\ = ((1+\alpha)^{i+1} - 1)N(t).\end{aligned}$$

Hence,

$$N(t + D(i+1)) \leq N(t) + ((1+\alpha)^{i+1} - 1)N(t) = (1+\alpha)^{i+1}N(t).$$

Furthermore,

$$\begin{aligned}N(t+D(i+1)) \geq N(t+Di) - \ell \geq N(t+Di) - \alpha N(t+Di)\\ = (1-\alpha)N(t+Di) \geq (1-\alpha)^{i+1}N(t).\end{aligned}$$

By induction, the claim is true for all $i \in \mathbb{N}$. □

We are also interested in the number of nodes that leave during an interval of time. The calculation of the maximum number of nodes that leave during an interval is complicated by the possibility of nodes entering during the interval, allowing additional nodes to leave.

Lemma 17. *For $\alpha > 0$, all nonnegative integers $i \leq -1/\log_2(1-\alpha)$ and every time $t \geq 0$, at most $(1-(1-\alpha)^i)N(t)$ nodes leave during $(t, t+Di]$.*

Proof. The proof is by induction on i. When $i = 0$, the interval is empty, so $0 = (1-(1-\alpha)^0)N(t)$ nodes leave during the interval. Now let $i \geq 0$, let $t \geq 0$, and suppose at most $(1-(1-\alpha)^i)N(t+D)$ nodes leave during $(t+D, t+D(i+1)]$.

Let $e \geq 0$ and $\ell \geq 0$ be the number of nodes that enter and leave, respectively, during $(t, t+D]$. By Assumption A5, $e + \ell \leq \alpha N(t)$, so $\ell \leq \alpha N(t)$ and $N(t+D) = N(t) + e - \ell = N(t) + (\ell + e) - 2\ell \leq (1+\alpha)N(t) - 2\ell$. The number of nodes that leave during $(t, t+D(i+1)]$ is the number that leave during $(t, t+D]$ plus the number that leave during $(t+D, t+D(i+1)]$, which is at most

$$\begin{aligned}
&\ell + (1-(1-\alpha)^i)N(t+D) \\
&\qquad \le \ell + (1-(1-\alpha)^i)[(1+\alpha)N(t) - 2\ell] \\
&\qquad = (1-(1-\alpha)^i)(1+\alpha)N(t) + (2(1-\alpha)^i - 1)\ell \\
&\qquad \le (1-(1-\alpha)^i)(1+\alpha)N(t) + (2(1-\alpha)^i - 1)\alpha N(t) \\
&\qquad = (1-(1-\alpha)^{i+1})N(t).
\end{aligned}$$

Note that $2(1-\alpha)^i - 1 \ge 0$, since $i \le -1/\log_2(1-\alpha)$. By induction, the claim is true for all $i \in \mathbb{N}$. □

Recall that a node is *active* at time t if it has entered by time t, but has not left or crashed by time t. The next lemma shows that some node remains active throughout any interval of length $3D$.

Lemma 18. *For every $t > 0$, at least one node is active throughout* $[\max\{0, t-2D\}, t+D]$.

Proof. Let S be the set of nodes present at time $t' = \max\{0, t-2D\}$, so $|S| = N(t') \ge N_{min}$. By Lemma 16, at most $((1+\alpha)^3 - 1)|S|$ nodes enter during $(t', t+D]$, so there are at most $(1+\alpha)^3|S|$ nodes present at time $t+D$ and at most $\Delta(1+\alpha)^3|S|$ nodes have crashed by time $t+D$. Constraint (3.1) implies that $-1/\log_2(1-\alpha) \ge 4 \ge 3$. So, by Lemma 17, at most $(1-(1-\alpha)^3)|S|$ nodes leave during $(t', t+D]$ and there are at least $(1-\alpha)^3|S|$ nodes present at time $t+D$. Thus, at least

$$((1-\alpha)^3 - \Delta(1+\alpha)^3)|S| \ge ((1-\alpha)^3 - \Delta(1+\alpha)^3)N_{min} \tag{3.8}$$

nodes in S are active at time $t+D$. By Constraint (3.2), $((1-\alpha)^3 - \Delta(1+\alpha)^3)\, N_{min} > 1$, so at least one node in S is still active at time $t+D$. □

Below, a local variable name is superscripted with t to denote the value of that variable at time t; e.g., v_p^t is the value of node p's local variable v at time t.

In the analysis, we will frequently be comparing the data in nodes' *Changes* sets to the set of ENTER, JOINED, and LEAVE events that have actually occurred. To facilitate this comparison, we define a set $\mathit{SysInfo}^I$ that contains perfect information for the time interval I. For each node q, let t_q^e, t_q^j, and t_q^ℓ be the times when the events ENTER_q, JOINED_q, and LEAVE_q occur, respectively. If $q \in S_0$, then we set $t_q^e = t_q^j = 0$. Then we have:

$$\mathit{SysInfo}^I = \{\mathit{enter}(q) \mid t_q^e \in I\} \cup \{\mathit{join}(q) \mid t_q^j \in I\} \cup \{\mathit{leave}(q) \mid t_q^\ell \in I\}.$$

In particular,

$$\mathit{SysInfo}^{[0,0]} = \{\mathit{enter}(q) \mid q \in S_0\} \cup \{\mathit{join}(q) \mid q \in S_0\}.$$

Since a node p that is active throughout $[t_p^e, t + D]$ directly receives all enter, joined, and leave messages broadcast during $[t_p^e, t]$, within D time, we have:

Observation 19. *For every node p and all times $t \geq t_p^e$, if p is active at time $t + D$, then $\mathit{SysInfo}^{[t_p^e, t]} \subseteq \mathit{Changes}_p^{t+D}$.*

By assumption, for every node $p \in S_0$, $\mathit{SysInfo}^{[0,0]} \subseteq \mathit{Changes}_p^0$, and hence Observation 42 implies:

Observation 20. *For every node $p \in S_0$, if p is active at time $t \geq 0$, then*
$\mathit{SysInfo}^{[0, \max\{0, t-D\}]} \subseteq \mathit{Changes}_p^t$.

The purpose of Lemmas 21, 22, and 23 is to show that information about nodes entering, joining, and leaving is propagated properly, via the *Changes* sets.

Lemma 21. *Suppose that, at time T'', a node $p \notin S_0$ receives an enter-echo message from a node q sent at time T' in reply to an enter message from p. Let T be any time such that $\max\{0, T'' - 2D\} \leq T \leq t_p^e$. Suppose p is active at time $T + 2D$ and q is active throughout $[U, T + D]$, where $U \leq \max\{0, T'' - 2D\}$. Then $\mathit{SysInfo}^{(U,T]} \subseteq \mathit{Changes}_p^{T+2D}$.*

Proof. Consider any node r that enters, joins, or leaves at time $\hat{t} \in (U, T]$. Note that q directly receives this event's announcement, since q is active throughout $(U, T + D]$, which contains $[\hat{t}, \hat{t} + D]$, the interval during which the announcement message is in transit. There are two cases, depending on the time, v, at which q receives this message.

Case 1: $v \leq T'$. Since q receives the enter message from p at T', information about this change to r is in $Changes_q^{T'}$, in the enter-echo message that q sends to p at time T'. Thus, this information is in $Changes_p^{T''} \subseteq Changes_p^{T+2D}$.

Case 2: $v > T'$. Messages are not received before they are sent, so $T' \geq t_p^e$. Since $v \leq \hat{t} + D$, it follows that $v + D \leq \hat{t} + 2D \leq T + 2D$. Thus $[v, v + D]$ is contained in $[t_p^e, T + 2D]$. Immediately after receiving the announcement about r, node q broadcasts an echo message in reply. Since p is active throughout this interval, it directly receives this echo message.

In both cases, the information about r's change reaches p by time $T + 2D$. It follows that $SysInfo^{(U,T]} \subseteq Changes_p^{T+2D}$. □

Lemma 22. *For every node p, if p is active at time $t \geq t_p^e + 2D$, then $SysInfo^{[0,t-D]} \subseteq Changes_p^t$.*

Proof. The proof is by induction on the order in which nodes enter the system. If $p \in S_0$, then $t_p^e = 0$, so $SysInfo^{[0,t-D]} \subseteq Changes_p^t$ follows from Observation 43.

Now consider any node $p \notin S_0$ and suppose that the claim holds for all nodes that enter earlier than p. Suppose p is active at time $t \geq t_p^e + 2D$. By Lemma 18, there is at least one node q that is active throughout $[\max\{0, t_p^e - 2D\}, t_p^e + D]$. Node q receives an enter message from p at some time $t' \in [t_p^e, t_p^e + D]$ and sends an enter-echo message back to p. This message is received by p at some time $t'' \in [t', t' + D]$.

If $q \in S_0$, then $SysInfo^{[0,\max\{0,t'-D\}]} \subseteq Changes_q^{t'}$, by Observation 43. If $q \notin S_0$, then $0 < t_q^e \leq \max\{0, t_p^e - 2D\}$, so $t_q^e \leq t_p^e - 2D$. Therefore $t_q^e + 2D \leq t_p^e \leq t'$. Since q entered earlier than p, it follows from the induction hypothesis that $SysInfo^{[0,t'-D]} \subseteq Changes_q^{t'}$. Thus, in both cases, $\mathit{SysInfo}^{[0,\max\{0,t'-D\}]} \subseteq \mathit{Changes}_q^{t'}$. At time $t'' \leq t$, p receives the enter-echo message from q, so $\mathit{SysInfo}^{[0,\max\{0,t'-D\}]} \subseteq \mathit{Changes}_p^{t''} \subseteq \mathit{Changes}_p^{t}$.

Applying Lemma 21 for q, with $U = \max\{0, t_p^e - D\}$, $T = t_p^e$, $T' = t'$ and $T'' = t''$ implies

$$\mathit{SysInfo}^{(\max\{0,t'-D\},t_p^e]} \subseteq \mathit{Changes}_p^{t_p^e+2D}.$$

Since $t \geq t_p^e + 2D$, $\mathit{Changes}_p^{t_p^e+2D}$ is a subset of $\mathit{Changes}_p^t$. Observation 42 implies $\mathit{SysInfo}^{[t_p^e,t-D]} \subseteq \mathit{Changes}_p^t$. Hence, $\mathit{SysInfo}^{[0,t-D]} \subseteq \mathit{Changes}_p^t$. □

Lemma 23. *For every node $p \notin S_0$, if p joins at time t_p^j and is active at time $t \geq t_p^j$, then $\mathit{SysInfo}^{[0,\max\{0,t-2D\}]} \subseteq \mathit{Changes}_p^t$.*

Proof. The proof is by induction on the order in which nodes join the system. Let $p \notin S_0$ be a node that joins at time $t_p^j \leq t$ and suppose the claim holds for all nodes that join before p. If $t \geq t_p^e + 2D$, then the claim follows by Lemma 22. So, suppose $t < t_p^e + 2D$.

Before joining, p receives an enter-echo message from a joined node in reply to its enter message. Suppose p first receives at time t'' an enter-echo message sent by q at time t'; $t_p^e \leq t' \leq t'' \leq t_p^j$. If $q \in S_0$, then by Observation 43, $\mathit{SysInfo}^{[0,\max\{0,t'-D\}]} \subseteq \mathit{Changes}_q^{t'}$. Otherwise, by the induction hypothesis, $\mathit{SysInfo}^{[0,\max\{0,t'-2D\}]} \subseteq \mathit{Changes}_q^{t'}$, since q joined prior to p and is active at time $t' \geq t_q^j$. Note that $\mathit{Changes}_q^{t'} \subseteq \mathit{Changes}_p^{t''} \subseteq \mathit{Changes}_p^t$. If $t \leq 2D$, then $\max\{0, t-2D\} = 0$ and the claim holds.

If $t > 2D$, then let S be the set of nodes present at time $\max\{0, t' - 2D\}$; $|S| = N(\max\{0, t' - 2D\})$. By Lemma 17 and Constraint (3.1), at most $(1-(1-\alpha)^3)|S|$ nodes leave during $(\max\{0, t'-2D\}, t'+D]$. Since $t'' \leq t' + D$, it follows that $|\mathit{Present}_p^{t''}| \geq |S| - (1-(1-\alpha)^3)|S| = (1-\alpha)^3|S|$. Hence, from lines 94 and 95 of Algorithm 10, p waits until it has received at least $\mathit{join\_bound} = \gamma \cdot |\mathit{Present}_p^{t''}| \geq \gamma \cdot (1-\alpha)^3|S|$ enter-echo messages before joining.

By Lemma 16, at most $((1+\alpha)^3 - 1)|S|$ nodes enter during $(\max\{0, t'-2D\}, t'+D]$. Thus, at time $t' + D$, at most $(1+\alpha)^3|S|$ nodes are present and at most $\Delta(1+\alpha)^3|S|$ nodes are crashed.

Hence, the number of enter-echo messages p receives before joining from nodes that were active throughout $[\max\{0, t'-2D\}, t'+D]$ is $join\_bound$ minus the total number of enters, leaves and crashes, which is at least

$$\begin{aligned}\gamma \cdot (1-\alpha)^3|S| - [((1+\alpha)^3-1)|S| + (1-(1-\alpha)^3)|S| + \Delta(1+\alpha)^3|S|] \\ = [(1+\gamma)(1-\alpha)^3 - (1+\Delta)(1+\alpha)^3]|S| \\ \geq [(1+\gamma)(1-\alpha)^3 - (1+\Delta)(1+\alpha)^3]N_{min} \quad (3.9)\end{aligned}$$

Rearranging Constraint (3.3), we get

$$[(1+\gamma)(1-\alpha)^3 - (1+\Delta)(1+\alpha)^3 N_{min}] \geq 1,$$

so expression (3.9) is at least 1. Hence p receives an enter-echo message at some time $T'' \leq t_p^j$ from a node q' that is active throughout

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

Let T' be the time that q' sent its enter-echo message in reply to the enter message from p. Applying Lemma 21 for q', with $U = \max\{0, t'-2D\}$, and $T = t-2D$ gives $SysInfo^{(\max\{0,t'-2D\},t-2D]} \subseteq Changes_p^t$.

Thus, $SysInfo^{[0,t-2D]} = SysInfo^{[0,\max\{0,t'-2D\}]} \cup SysInfo^{(\max\{0,t'-2D\},t-2D]} \subseteq Changes_p^t$. □

Next we prove that every node that remains active sufficiently long after it enters, will succeed in joining.

Theorem 24. *Every node $p \notin S_0$ that is active at time $t_p^e + 2D$ joins by time $t_p^e + 2D$.*

Proof. The proof is by induction on the order in which nodes enter the system. Let $p \notin S_0$ be a node that enters at time t_p^e and is active at time $t_p^e + 2D$. Suppose the claim holds for all nodes that enter before p.

By Lemma 18, there is a node q that is active throughout $[\max\{t_p^e - 2D, 0\}, t_p^e + D]$. If $q \in S_0$, then q joins at time 0. If not, then $t_q^e \leq t_p^e - 2D$, so, by the induction hypothesis, q joins by time $t_q^e + 2D \leq t_p^e$. Since q is active at time $t_p^e + D$, it receives the enter message from p during $[t_p^e, t_p^e + D]$ and sends an enter-echo message in reply. Since p is active at time $t_p^e + 2D$, it receives the enter-echo message from q by time $t_p^e + 2D$. Hence, by time $t_p^e + 2D$, p receives at least one enter-echo message from a joined node in reply to its enter message.

Suppose the first enter-echo message p receives from a joined node in reply to its enter message is sent by node q' at time t' and received by p at time t''. By Lemma 23, $\mathit{SysInfo}^{[0,\max\{0,t'-2D\}]} \subseteq \mathit{Changes}_{q'}^{t'} \subseteq \mathit{Changes}_p^{t''}$.

Let S be the set of nodes present at time $\max\{0, t' - 2D\}$. Since $t'' \leq t' + D$, it follows from Lemma 16 that at most $((1+\alpha)^3 - 1)|S|$ nodes enter during $(\max\{0, t' - 2D\}, t'']$. Thus, $|\mathit{Present}_p^{t''}| \leq |S| + ((1+\alpha)^3 - 1)|S| = (1+\alpha)^3|S|$. From line 94 in Algorithm 10, it follows that $\mathit{join\_bound} \leq \gamma \cdot (1+\alpha)^3|S|$.

By Lemma 17 and Constraint (3.1), at most $(1-(1-\alpha)^3)|S|$ nodes leave during $(\max\{0, t' - 2D\}, t' + D]$. Also, by Lemma 16, at most $(1+\alpha)^3|S|$ nodes are present at $t' + D$ and so at most $\Delta(1+\alpha)^3|S|$ nodes are crashed at $t' + D$. Since $t_p^e \leq t' \leq t_p^e + D$, the nodes in S that do not leave during $(\max\{0, t' - 2D\}, t' + D]$ and are not crashed at $t' + D$ are active throughout $[t_p^e, t_p^e + D]$ and send enter-echo messages in reply to p's enter message. By time $t_p^e + 2D$, p receives all these enter-echo messages. There are at least

$$|S| - (1 - (1 - \alpha)^3)|S| - \Delta(1 + \alpha)^3|S| = (1 - \alpha)^3|S| - \Delta(1 + \alpha)^3|S|$$

such enter-echo messages. By Constraint (3.4),

$$\frac{(1-\alpha)^3}{(1+\alpha)^3} - \Delta \geq \gamma,$$

so the value of *join_bound* is at most

$$\gamma \cdot (1+\alpha)^3|S| \;\leq\; (\frac{(1-\alpha)^3}{(1+\alpha)^3} - \Delta) \cdot (1+\alpha)^3|S| \;=\; (1-\alpha)^3|S| - \Delta(1+\alpha)^3|S|.$$

Thus, by time $t_p^e + 2D$, the condition in line 95 of Algorithm 10 holds and node p joins. □

3.1.7.2 Proof that Reads and Writes Terminate

Next, we show that all read and write operations terminate. Specifically, we show that the number of replies for which an operation waits is at most the number that it is guaranteed to receive.

Since $\mathit{enter}(q)$ is added to $\mathit{Changes}_p$ whenever $\mathit{join}(q)$ is, we get the following observation.

Observation 25. *For every time $t \geq 0$ and every node p that is active at time t, $\mathit{Members}_p^t \subseteq \mathit{Present}_p^t$.*

Lemma 26 relates a node's current estimate of the number of nodes present to the number of nodes that were present in the system $2D$ time units earlier. Lemma 27 relates a node's current estimate of the number of nodes that are members to the number of nodes that were present in the system $4D$ time units earlier. Lemma 26 is used in the proof of Lemma 51 and Lemma 27 is used in the proof of Theorem 37. The proofs of Lemmas 26 and 27 are very similar to each other and are thus presented together.

Lemma 26. *For every node p and every time $t \geq t_p^j$ at which p is active,*

$$(1-\alpha)^2 \cdot N(\max\{0, t-2D\}) \;\leq\; |\mathit{Present}_p^t| \;\leq\; (1+\alpha)^2 \cdot N(\max\{0, t-2D\}).$$

Proof. By Lemma 23, $\mathit{SysInfo}^{[0,\max\{0,t-2D\}]} \subseteq \mathit{Changes}_p^t$. Thus $\mathit{Present}_p^t$ contains all nodes that are present at time $\max\{0, t-2D\}$, plus any nodes that enter in $(\max\{0, t-2D\}, t]$ which p has learned about, minus any nodes that leave in $(\max\{0, t-2D\}, t]$ which p has learned about. Then, by Lemma 16,

$$|Present_p^t| \leq N(\max\{0, t-2D\}) + ((1+\alpha)^2 - 1) \cdot N(\max\{0, t-2D\})$$
$$= (1+\alpha)^2 \cdot N(\max\{0, t-2D\}).$$

Similarly, by Lemma 17 and Constraint (3.1),

$$|Present_p^t| \geq N(\max\{0, t-2D\}) - (1 - (1-\alpha)^2) \cdot N(\max\{0, t-2D\})$$
$$= (1-\alpha)^2 \cdot N(\max\{0, t-2D\}).$$

□

Lemma 27. *For every node p and every time $t \geq t_p^j$ at which p is active,*

$$(1-\alpha)^4 \cdot N(\max\{0, t-4D\}) \leq |Members_p^t| \leq (1+\alpha)^4 \cdot N(\max\{0, t-4D\}).$$

Proof. By Lemma 23, $SysInfo^{[0,\max\{0,t-2D\}]} \subseteq Changes_p^t$ and, by Theorem 24, every node that enters by time $\max\{0, t-4D\}$ joins by time $\max\{0, t-2D\}$ if it is still active. Thus $Members_p^t$ contains all nodes that are present at time $\max\{0, t-4D\}$ plus any nodes that enter in $(\max\{0, t-4D\}, t]$ which p learns have joined, minus any nodes that leave in $(\max\{0, t-4D\}, t]$ which p learns have left. Then, by Lemma 16,

$$|Members_p^t| \leq N(\max\{0, t-4D\}) + ((1+\alpha)^4 - 1) \cdot N(\max\{0, t-4D\})$$
$$= (1+\alpha)^4 \cdot N(\max\{0, t-4D\}).$$

Similarly, by Lemma 17 and Constraint (3.1),

$$\begin{aligned}|Members_p^t| &\geq N(\max\{0, t - 2D\}) - (1 - (1 - \alpha)^4) \cdot N(\max\{0, t - 4D\}) \\ &= (1 - \alpha)^4 \cdot N(\max\{0, t - 4D\}).\end{aligned}$$

□

The next lemma proves a lower bound on the number of nodes that reply to an operation's query or update message.

Lemma 28. *If node p is active at time $t \geq t_p^j$, then the number of nodes that join by time t and are still active at time $t + D$ is at least $\left[\frac{(1-\alpha)^3}{(1+\alpha)^2} - \Delta(1 + \alpha)\right] \cdot |Present_p^t|$.*

Proof. By Lemma 17 and Constraint (3.1), the maximum number of nodes that leave during $(\max\{0, t - 2D\}, t + D]$ is at most $(1 - (1 - \alpha)^3) \cdot N(\max\{0, t - 2D\})$. By Lemma 16, at most $((1 + \alpha)^3 - 1) \cdot N(\max\{0, t - 2D\})$ nodes enter during $(\max\{0, t - 2D\}, t + D]$. So, at most $\Delta(1 + \alpha)^3 \cdot N(\max\{0, t - 2D\})$ nodes are crashed by $t + D$. Thus, there are at least

$$\begin{aligned}&N(\max\{0, t - 2D\}) - (1 - (1 - \alpha)^3) \cdot N(\max\{0, t - 2D\}) \\ &\qquad - \Delta(1 + \alpha)^3 \cdot N(\max\{0, t - 2D\}) \\ &\qquad\qquad = [(1 - \alpha)^3 - \Delta(1 + \alpha)^3] \cdot N(\max\{0, t - 2D\})\end{aligned}$$

nodes that were present at time $\max\{0, t-2D\}$ and are still active at time $t+D$. This number is bounded below by $\left[\frac{(1-\alpha)^3}{(1+\alpha)^2} - \Delta(1 + \alpha)\right] \cdot |Present_p^t|$ since, by Lemma 26, $N(\max\{0, t - 2D\}) \geq |Present_p^t|/(1 + \alpha)^2$. By Theorem 24, all of these nodes are joined by time t. □

Theorem 29. *Every read or write operation invoked by a node that remains active completes.*

Proof. Each operation consists of a read phase and a write phase. We show that each phase terminates within $2D$ time, provided the client does not crash or leave.

Consider a phase of an operation by client p that starts at time t. Every node that joins by time t and is still active at time $t + D$ receives p's query or update message and replies with a reply message or an ack message by time $t + D$. By Lemma 51, there are at least $\left[\frac{(1-\alpha)^3}{(1+\alpha)^2} - \Delta(1+\alpha)\right] \cdot |\mathit{Present}_p^t|$ such nodes.

From Constraint (3.5) and Observation 48,

$$\left[\frac{(1-\alpha)^3}{(1+\alpha)^2} - \Delta(1+\alpha)\right] \cdot |\mathit{Present}_p^t| \geq \beta \cdot |\mathit{Present}_p^t| \\ \geq \beta \cdot |\mathit{Members}_p^t| = rw\_bound_p^t.$$

Thus, by time $t + 2D$, p receives sufficiently many reply or ack messages to complete the phase. □

3.1.7.3 Proof of Atomicity of CCReg

Now we prove atomicity of the CCReg algorithm. Let $\mathcal{T}$ be the set of read operations that complete and write operations that execute line 74 of Algorithm 4. For any node p, let $ts_p^t = (num_p^t, w\_id_p^t)$ denote the *timestamp* of the latest register value known to node p at time t. Note that new timestamps are created by write operations (on lines 70-71 of Algorithm 4) and are sent via enter-echo, update, and update-echo messages. Initially, $ts_p^0 = (0, \perp)$ for all nodes p.

For any operation o in $\mathcal{T}$ by p, the *timestamp of its read phase*, $ts^{rp}(o)$, is ts_p^t, where t is the end of its read phase (i.e., when the condition on line 64 of Algorithm 4 evaluates to true). The *timestamp of its write phase*, $ts^{wp}(o)$, is ts_p^t, where t is the beginning of its write phase (i.e., when it broadcasts on line 74 of Algorithm 4). The *timestamp of a read operation* in $\mathcal{T}$ is the timestamp of its read phase. The *timestamp of a write operation* in $\mathcal{T}$ is the timestamp of its write phase.

Note that w_id is equal to p and num is set to one greater than the largest sequence value observed during an operation's read phase. This implies the next observation:

Observation 30. *Each write operation in $\mathcal{T}$ has a unique timestamp.*

The next observation follows by a simple induction, since every timestamp other than $(0, \perp)$ comes from Lines 70-71 of Algorithm 4.

Observation 31. *Consider any read op_1 in $\mathcal{T}$. If the timestamp of a read op_1 is $(0, \perp)$, then op_1 returns $\perp$. Otherwise, there is a write op_2 in $\mathcal{T}$ such that $ts(op_1) = ts(op_2)$ and the value returned by op_1 equals the value written by op_2.*

Lemmas 55–35 show that write phase information propagates properly through the system. They are analogous to Observation 43 and Lemmas 21–23, regarding the propagation of information about Enter, Joined, and Leave events.

Lemma 32. *If o is an operation in $\mathcal{T}$ whose write phase w starts at t_w, node p is active at time $t \geq t_w + D$, and $t_p^e \leq t_w$, then $ts_p^t \geq ts^{wp}(o)$.*

Proof. Since p is active throughout $[t_w, t_w + D]$, it directly receives the update message broadcast by w at time t_w. Hence, from lines 38–39 of Algorithm 8, $ts_p^t \geq ts^{wp}(o)$. □

Lemma 33. *Suppose a node $p \notin S_0$ receives an enter-echo message at time t'' from a node q that sends it at time t' in reply to an enter message from p. If o is an operation whose write phase w starts at t_w, p is active at time $t \geq \max\{t'', t_w + 2D\}$, and q is active throughout $[t_w, t_w + D]$, then $ts_p^t \geq ts^{wp}(o)$.*

Proof. Since q is active throughout $[t_w, t_w + D]$, it receives the update message from w at some time $\hat{t} \in [t_w, t_w + D]$, so $ts_q^{\hat{t}} \geq ts^{wp}(o)$. At time $t'' \leq t$, node p receives the enter-echo sent by node q at time t', so $ts_p^t \geq ts_p^{t''} \geq ts_q^{t'}$. If $t' \geq \hat{t}$, then $ts_q^{t'} \geq ts_q^{\hat{t}}$, so $ts_p^t \geq ts^{wp}(o)$. If $\hat{t} > t'$, then q sends an update-echo at time $\hat{t} \leq t_w + D$, p receives it by time $\hat{t} + D \leq t_w + 2D \leq t$, and, thus, $ts_p^t \geq ts_q^{\hat{t}} \geq ts^{wp}(o)$. □

Lemma 34. *If o is an operation in $\mathcal{T}$ whose write phase w starts at t_w and node p is active at time $t \geq \max\{t_p^e + 2D, t_w + D\}$, then $ts_p^t \geq ts^{wp}(o)$.*

Proof. The proof is by induction on the order in which nodes enter the system. Suppose the claim holds for all nodes that enter before p. If $t_p^e \leq t_w$, which is the case for all $p \in S_0$, then the claim follows from Lemma 55.

If $t_w < t_p^e$, then by Lemma 18, there is at least one node q that is active throughout $[\max\{0, t_p^e - 2D\}, t_p^e + D]$. It receives an enter message from p at some time $t' \in [t_p^e, t_p^e + D]$ and sends an enter-echo message containing $ts_q^{t'}$ back to p. This message is received by p at some time $t'' \leq t' + D \leq t_p^e + 2D \leq t$, so $ts_q^{t'} \leq ts_p^{t''} \leq ts_p^t$.

The first case is when $t_w \geq \max\{0, t_p^e - 2D\}$. Since $t_w + D < t_p^e + D$, it follows that q is active throughout $[t_w, t_w + D]$. Furthermore, $t \geq t_p^e + 2D \geq \max\{t'', t_w + 2D\}$. Hence, Lemma 56 implies that $ts_p^t \geq ts^{wp}(o)$.

The second case is when $t_w < \max\{0, t_p^e - 2D\}$. Since $t_w \geq 0$, it follows that $t_p^e - 2D > 0$, $t_q^e \leq \max\{0, t_p^e - 2D\} = t_p^e - 2D$, and $t_w < t_p^e - 2D \leq t' - 2D$, so $t' \geq \max\{t_q^e + 2D, t_w + D\}$. Note that q is active at time t' and q enters before node p, so, by the induction hypothesis, $ts_q^{t'} \geq ts^{wp}(o)$. Hence, $ts_p^t \geq ts^{wp}(o)$. □

Lemma 35. *If o is an operation in $\mathcal{T}$ whose write phase starts at t_w, node $p \notin S_0$ joins at time t_p^j, and p is active at time $t \geq \max\{t_p^j, t_w + 2D\}$, then $ts_p^t \geq ts^{wp}(o)$.*

Proof. The proof is by induction on the order in which nodes enter the system. Suppose the claim holds for all nodes that join before p. If $t \geq t_p^e + 2D$, then the claim follows by Lemma 57. So, suppose $t < t_p^e + 2D$. If $t_p^e \leq t_w$, then the claim follows by Lemma 55. So, suppose $t_w < t_p^e$.

Before p joins, it receives an enter-echo message from a joined node in reply to its enter message. Suppose p first receives such an enter-echo message at time t'' and this enter-echo was sent by q at time t'. Then $t'' \leq t_p^j \leq t$ and $ts_q^{t'} \leq ts_p^{t''} \leq ts_p^t$.

Now we prove that p receives an enter-echo message from a node q' that is active throughout $[\max\{0, t' - 2D\}, t' + D]$. Let S be the set of nodes present at time $\max\{0, t' - 2D\}$, so $|S| = N(\max\{0, t' - 2D\})$. By Lemma 17 and Constraint (3.1), at most $(1 - (1 - \alpha)^3)|S|$ nodes leave during $(\max\{0, t' - 2D\}, t' + D]$. Since $t'' \leq t' + D$, it follows that $|Present_p^{t''}| \geq$

$|S| - (1-(1-\alpha)^3)|S| = (1-\alpha)^3|S|$. Hence, from lines 94 and 95 of Algorithm 10, p waits until it has received at least $\mathit{join\_bound} = \gamma \cdot |\mathit{Present}_p^{t''}| \geq \gamma \cdot (1-\alpha)^3|S|$ enter-echo messages before joining.

By Lemma 16, at most $((1+\alpha)^3 - 1)|S|$ nodes enter during $(\max\{0, t' - 2D\}, t' + D]$. Thus, at time $t' + D$, at most $(1+\alpha)^3|S|$ nodes are present and at most $\Delta(1+\alpha)^3|S|$ nodes are crashed. The number of enter-echo messages p receives before joining from nodes that were active throughout $[\max\{0, t' - 2D\}, t' + D]$ is $\mathit{join\_bound}$ minus the total number of enters, leaves and crashes, which is at least

$$\gamma \cdot (1-\alpha)^3|S| - [((1+\alpha)^3 - 1)|S| + (1-(1-\alpha)^3)|S| + \Delta(1+\alpha)^3|S|]$$
$$\geq [(1+\gamma)(1-\alpha)^3 - (1+\Delta)(1+\alpha)^3]N_{min}. \quad (3.10)$$

Rearranging Constraint (3.3), we get $[(1+\gamma)(1-\alpha)^3 - (1+\Delta)(1+\alpha)^3 N_{min}] \geq 1$, so expression (3.10) is at least 1. Hence p receives an enter-echo message at some time $T'' \leq t_p^j$ from a node q' that is active throughout $[\max\{0, t' - 2D\}, t' + D] \supseteq [\max\{0, t' - 2D\}, t - D]$. Let T' be the time that q' sent its enter-echo message in reply to the enter message from p. Then $ts_{q'}^{T'} \leq ts_p^{T''} \leq ts_p^t$.

Note that $t_w < t_p^e \leq t'$ so $t_w + D \leq t' + D$. If $t_w \geq \max\{0, t' - 2D\}$, then q' is active throughout $[t_w, t_w + D]$. Since $t \geq \max\{T'', t_w + 2D\}$, it follows by Lemma 56 that $ts_p^t \geq ts^{wp}(o)$. So, suppose $t_w < \max\{0, t' - 2D\}$.

Since $t_w \geq 0$, it follows that $t' > t_w + 2D$. If $q \in S_0$, then $t_q^e = 0 \leq t_w$, so, by Lemma 55, $ts_q^{t'} \geq ts^{wp}(o)$. If $q \notin S_0$, then, by the induction hypothesis, $ts_q^{t'} \geq ts^{wp}(o)$, since q joins at time $t_q^j < t_p^j \leq t'$. Thus, in both cases, $ts_p^t \geq ts^{wp}(o)$. □

Lemma 59 is the key lemma for proving atomicity of CCReg. It shows that for two non-overlapping operations in $\mathcal{T}$, the timestamp of the read phase of the latter operation is at least as big as the timestamp of the write phase of the former. Theorem 37 uses Lemma 59 to show that the timestamps of two non-overlapping operations respect real time ordering and completes the proof of atomicity.

Lemma 36. *For any two operations op_1 and op_2 in $\mathcal{T}$, if op_1 finishes before op_2 starts, then $ts^{wp}(op_1) \leq ts^{rp}(op_2)$.*

Proof. Let p_1 be the node that invokes op_1, let w denote the write phase of op_1, let t_w be the start time of w, and let $\tau_w = ts^{wp}(op_1) = ts_{p_1}^{t_w}$. Similarly, let p_2 be the node that invokes op_2, let r denote the read phase of op_2, let t_r be the start time of r, and let $\tau_r = ts^{rp}(op_2) = ts_{p_2}^{t_r}$.

Let Q_w be the set of nodes that p_1 hears from during w (i.e., that sent messages causing p_1 to increment *rw_counter* on line 80 of Algorithm 4) and Q_r be the set of nodes that p_2 hears from during r (i.e., that sent messages causing p_2 to increment *rw_counter* on line 62 of Algorithm 4). Let $P_w = |Present_{p_1}^{t_w}|$ and $M_w = |Members_{p_1}^{t_w}|$ be the sizes of the *Present* and *Members* sets belonging to p_1 at time t_w, and $P_r = |Present_{p_2}^{t_r}|$ and $M_r = |Members_{p_2}^{t_r}|$ be the sizes of the *Present* and *Members* sets belonging to p_2 at time t_r.

Case I: $t_r > t_w + 2D$. We start by showing that there exists a node q in Q_r such that $t_q^j \leq t_r - 2D$. Each node $q \in Q_r$ receives and responds to r's query, so it joins by time $t_r + D$. By Theorem 24, the number of nodes that can join during $(t_r - 2D, t_r + D]$ is at most the number of nodes that can enter in $(\max\{0, t_r - 4D\}, t_r + D]$. By Lemma 16, the number of nodes that can enter during $(\max\{0, t_r - 4D\}, t_r + D]$ is at most $((1+\alpha)^5 - 1) \cdot N(\max\{0, t_r - 4D\})$. By Lemma 27, $N(\max\{0, t_r - 4D\}) \leq M_r/(1-\alpha)^4$. From the code and Constraint (3.6), it follows that $|Q_r| \geq \beta M_r > M_r(1+\alpha)^5 - 1)/(1-\alpha)^4 \geq (1+\alpha)^5 - 1) \cdot N(\max\{0, t_r - 4D\})$, which is at most the number of nodes that can enter in $(\max\{0, t_r - 4D\}, t_r + D]$. Thus, a node $q \in Q_r$ joins by time $t_r - 2D$.

Suppose q receives r's query message at time $t' \geq t_r \geq t_w + 2D$. If $q \in S_0$, then $t_q^j = 0 \leq t_w$, so, by Lemma 55, $ts_q^{t'} \geq ts^{wp}(op_1) = \tau_w$. Otherwise, $q \notin S_0$, so $0 < t_q^j \leq t_r - 2D < t'$. Since $t_w + 2D < t_r \leq t'$, Lemma 35 implies that $ts_q^{t'} \geq ts^{wp}(op_1) = \tau_w$. In either case, q responds to r's query message with a timestamp at least as large as τ_w and, hence, $\tau_r \geq \tau_w$.

Case II: $t_r \leq t_w + 2D$. Let $J = \{p \mid t_p^j < t_r \text{ and } p \text{ is active at time } t_r\} \cup \{p \mid t_r \leq t_p^j \leq t_r + D\}$, which contains the set of all nodes that reply to r's query. By Theorem 24, all nodes that are present at time $\max\{0, t_r - 2D\}$ join by time t_r if they remain active. Therefore all nodes in J are either active at time $\max\{0, t_r - 2D\}$ or enter during $(\max\{0, t_r - 2D\}, t_r + D]$. By Lemma 16, $|J| \leq (1+\alpha)^3 N(\max\{0, t_r - 2D\})$.

Let K be the set of all nodes that are present at time $\max\{0, t_r - 2D\}$ and do not leave or crash during $(\max\{0, t_r - 2D\}, t_r + D]$. Note that K contains all the nodes in Q_w that do not leave or crash during $[t_w, t_r + D] \subseteq [\max\{0, t_r - 2D\}, t_r + D]$. By Lemma 17 and Constraint (3.1), at most $(1 - (1-\alpha)^3)N(\max\{0, t_r - 2D\})$ nodes leave during $[\max\{0, t_r - 2D\}, t_r + D]$. By Lemma 16, at most $((1+\alpha)^3 - 1)N(\max\{0, t_r - 2D\})$ nodes enter during $[\max\{0, t_r - 2D\}, t_r + D]$. So, at most $\Delta(1+\alpha)^3 N(\max\{0, t_r - 2D\})$ nodes are crashed at $t_r + D$.

From the code, $|Q_r| \geq \beta M_r$ and, by Lemma 27, $M_r \geq (1-\alpha)^4 N(\max\{0, t_r - 4D\})$. So,

$$|Q_r| \geq \beta(1-\alpha)^4 N(\max\{0, t_r - 4D\}).$$

Similarly,

$$|Q_w| \geq \beta M_w \geq \beta(1-\alpha)^4 N(\max\{0, t_w - 4D\}).$$

Therefore, the size of K is at least

$$\begin{aligned} &|Q_w| - (1 - (1-\alpha)^3 + \Delta(1+\alpha)^3)N(\max\{0, t_r - 2D\}) \\ &\qquad \geq (\beta(1-\alpha)^4 N(\max\{0, t_w - 4D\})) - (1 - (1-\alpha)^3 \\ &\qquad\qquad + \Delta(1+\alpha)^3)N(\max\{0, t_r - 2D\}). \end{aligned} \tag{3.11}$$

Since $t_r - t_w < 2D$, it follows that $\max\{0, t_r - 4D\} - \max\{0, t_w - 4D\} < 2D$. By Lemma 16, $N(\max\{0, t_r - 4D\}) \leq (1+\alpha)^2 \cdot N(\max\{0, t_w - 4D\})$. Thus we can replace

$N(\max\{0, t_w - 4D\})$ in Formula (3.11) with $(1+\alpha)^{-2} \cdot N(\max\{0, t_r - 4D\})$ and get:

$$\begin{aligned}|Q_r| + |K| \geq \beta(1-\alpha)^4 N(\max\{0, t_r - 4D\})& \\ &+ \beta(1-\alpha)^4(1+\alpha)^{-2} N(\max\{0, t_r - 4D\}) \\ &- (1 - (1-\alpha)^3 + \Delta(1+\alpha)^3) N(\max\{0, t_r - 2D\}) \\ = \beta(1-\alpha)^4(1+\alpha)^{-2}&(2 + 2\alpha + \alpha^2) N(\max\{0, t_r - 4D\}) \\ &- (\Delta(1+\alpha)^3 - (1-\alpha)^3 + 1) N(\max\{0, t_r - 2D\}).\end{aligned}$$

By Lemma 16,

$$N(\max\{0, t_r - 4D\}) \geq (1-\alpha)^{-2} N(\max\{0, t_r - 2D\}).$$

Thus,

$$\begin{aligned}|Q_r| + |K| &\geq \beta(1-\alpha)^2(1+\alpha)^{-2}(2 + 2\alpha + \alpha^2) N(\max\{0, t_r - 2D\}) \\ &\quad - (\Delta(1+\alpha)^3 - (1-\alpha)^3 + 1) N(\max\{0, t_r - 2D\}) \\ &= (\beta(1-\alpha)^2(1+\alpha)^{-2}(2 + 2\alpha + \alpha^2) - (\Delta(1+\alpha)^3 \\ &\qquad - (1-\alpha)^3 + 1)) N(\max\{0, t_r - 2D\})\end{aligned}$$

By Constraint (3.7),

$\beta(1 - \alpha)^2(1 + \alpha)^{-2}(2 + 2\alpha + \alpha^2) > (1 + \Delta)(1 + \alpha)^3 - (1 - \alpha)^3 + 1,$

so

$$\begin{aligned}|Q_r| + |K| > (((1+\Delta)(1+\alpha)^3 - (1-\alpha)^3 + 1) \\ - (\Delta(1+\alpha)^3 - (1-\alpha)^3 + 1)) \cdot N(\max\{0, t_r - 2D\}) \\ = (1+\alpha)^3 N(\max\{0, t_r - 2D\}) \geq |J|.\end{aligned}$$

This implies that K and Q_r intersect, since $K, Q_r \subseteq J$. For each node p in the intersection, $ts_p \geq \tau_w$ when p sends its reply to r and, thus, $\tau_w \leq \tau_r$. □

Theorem 37. CCReg *ensures atomicity.*

Proof. We show that, for every execution, there is a total order on the set of all completed read operations and all write operations that execute Line 74 of Algorithm 4 such that every read returns the value of the latest preceding write (or the initial value if there is no preceding write) and, if an operation op_1 finishes before another operation op_2 begins, then op_1 is ordered before op_2.

We first order the write operations in order of their (unique) timestamps. Then, we go over all reads in the ordering of the start times, and place a read with timestamp $(0, \bot)$ at the beginning of the total order. Place every other read after the write operation it reads from, and after all the previous reads that read from this write operation. By the Observation 54, every read in the total order returns the value of the latest preceding write (or $\bot$ if there is no preceding write).

We show that the total order respects the real-time order of non-overlapping operations in the execution. Let op_1 and op_2 be two operations in $\mathcal{T}$ such that op_1 finishes before op_2 starts. By the definition of timestamps, $ts(op_1) \leq ts^{wp}(op_1)$ and $ts^{rp}(op_2) \leq ts(op_2)$. By Lemma 59, $ts^{wp}(op_1) \leq ts^{rp}(op_2)$. Therefore, if op_2 is a read, then

$$ts(op_1) \leq ts(op_2) \tag{3.12}$$

If op_2 is a write, then $ts^{wp}(op_2) = ts^{rp}(op_2) + 1$, and

$$ts(op_1) < ts(op_2) \tag{3.13}$$

We consider the following cases:

- Suppose op_1 and op_2 are both writes. By (3.13), $ts(op_1) < ts(op_2)$ and thus the construction orders op_1 before op_2.

- Suppose op_1 is a write and op_2 is a read. By (3.12) and the construction, op_2 is placed after the write op_3 that op_2 reads from. If $ts(op_1) = ts(op_2)$ then $op_1 = op_3$ and op_2 is placed after op_1. If $ts(op_1) < ts(op_2)$ then op_3 is placed after op_1 as $ts(op_1) < ts(op_3)$ and thus op_2 is placed after op_1 in the total order.

- Suppose op_1 is a read and op_2 is a write. By 3.13, $ts(op_1) < ts(op_2)$. Now, either op_2 is the first write in the execution and op_1's timestamp is $(0, \perp)$ or there exists another write op_3 that op_1 reads from. If op_1's timestamp is $(0, \perp)$ then the construction orders op_1 before op_2. Otherwise, the construction orders op_3 before op_2. Since op_1 is ordered after op_3 but before any subsequent write, op_1 precedes op_2 in the total order.

- Finally, suppose that op_1 and op_2 are both reads. By 3.12, $ts(op_1) \leq ts(op_2)$. If op_1 and op_2 have the same timestamp, then they are placed after the same write (or before the first write) and the construction orders them based on their starting times. Since op_1 completes before op_2 starts, the construction places op_1 before op_2. If op_2 has a timestamp greater than that of op_1, then $ts(op_2)$ cannot be $(0, \perp)$ and so there is a write operation op_3 whose timestamp is greater than that of op_1 and equal to that of op_2. The construction places op_1 before op_3 and op_2 after op_3.

Thus, CCReg ensures atomicity. □

3.1.7.4 Proof that CCReg *Violates Atomicity if Churn Assumption is Violated.*

CCReg violates atomicity if Assumption A5 is violated. This is demonstrated by the following execution, in which large numbers of nodes enter and leave very quickly.

Let $|S_0| = n$ and let p be a node in S_0. Suppose the following sequence of events occur before time D. First, a set of nodes, denoted S_{new}, enter the system, with $|S_{new}| = m \gg n$. All join-related messages between $S_0 - \{p\}$ and $S_{new} \cup \{p\}$ take D time, while the rest of the messages take time $\ll D$. Thus, nodes in S_{new} hear from p before any other joined node and they use n, p's estimate of the system size, to calculate the number of messages they should hear from before joining. Thus all nodes in S_{new} join before time D but no node in S_0 other than p knows about S_{new} so far.

Second, immediately after joining, some node q in S_{new} invokes *write*(1). All write-related messages between S_0 and S_{new} take D time, while the rest of the messages take time $\ll D$. S_{new} is sufficiently large that the write protocol completes for q based solely on hearing from nodes in S_{new}. Thus the write completes before time D but no node in S_0 knows about the enters or the write so far.

Third, immediately after the write finishes, all the nodes in S_{new} leave. All leave-related messages between S_0 and S_{new} take D time, while the rest of the messages take time $\ll D$. Thus no node in S_0 knows about the enters, the write, or the leaves so far.

Finally, immediately after the leaves, node $p' \neq p$ in S_0 invokes a read. All read-related messages take time $\ll D$. Node p' uses its estimate of the system size as n to decide how many messages to wait for and is able to complete its read before time D by hearing only from nodes in $S_0 - \{p\}$. Since none of these nodes knows anything about the write, the read returns 0, which violates atomicity.

3.2 Byzantine-Tolerant Register Implementation in Systems with Churn

As described in Section 3.1, a shared read/write register emulation provides the illusion of shared-memory on top of message-passing models. Emulating a Byzantine-tolerant register requires replicating the register value on to more than two-thirds of the servers. Emulating a register in a dynamic system where servers and clients can enter and leave the system and be faulty is harder than in static systems.

3.2.1 Introduction

A long standing vision in distributed systems is to build reliable systems from unreliable components. We are increasingly dependent on services provided by distributed systems resulting in added vulnerability when it comes to failures in computer systems. In a dependable computing system, the term “Byzantine" fault is used to represent the worst kind of failures imaginable. Malicious attacks, operator mistakes, software errors and conventional crash faults are all encompassed by the term Byzantine faults [3]. The growing reliance of industry and government on distributed systems and increased connectivity to the Internet exposes systems to malicious attacks. Operator mistakes are also a very common cause of Byzantine faults [54]. The growth in the size of software in general leads to an increased number of software errors. Naturally, over the past four decades, there has been a significant work on consensus and replication techniques that tolerate Byzantine faults [3, 55, 56, 57] as it promises dependable systems that can tolerate any type of bad behavior.

The shared-memory model is a more convenient programming model than message-passing, and shared register emulations provide the illusion of shared-memory on top of message-passing models. In this work, we emulate a Byzantine-tolerant atomic register on top of a dynamic, message-passing system that never stops changing. Typically, crash-fault-tolerant emulations [9, 10] of a shared read/write register replicate the value of the register in multiple servers and require readers and writers to communicate with majority of servers. For instance, the ABD emulation [9] replicates the value of the shared register in servers.

It assumes that a majority of the servers do not fail. This problem of emulating a shared register has been extended to static systems with servers subject to Byzantine faults and these simulations typically require that two-thirds [58] or three-fourths [59] of the servers be non-faulty. It is shown in [60] that more than two-third correctness is necessary for a Byzantine-tolerant register simulation. Byzantine quorum systems (BQS) [57, 61, 62, 44] are a well known tool for ensuring consistency and availability of a shared register. A BQS is a collection of subsets of servers, each pair of which intersect in a set containing sufficiently many correct servers to guarantee consistency of the replicated register as seen by clients. Dantas et. al [63] present a comparative evaluation of several Byzantine quorum based storage algorithms in the literature.

The success of this replicated approach for static systems, where the set of readers, writers, and servers is fixed, has motivated several similar emulations for *dynamic* systems, where nodes may enter and leave. Change in system composition due to nodes entering and leaving is called *churn*. Ko et. al [37] provide a detailed discussion of churn behavior in practice. Most existing emulations of atomic registers in dynamic systems deal with crash-faults and rely either on the assumption that churn eventually stops for a long enough period (e.g., DynaStore [11] and RAMBO [12]) or on the assumption that the system size is bounded (e.g., [13], [36]). Attiya et al. [16, 17] proposed an emulation of a crash-fault tolerant shared register in a system that does not require churn to ever stop.

Bonomi et al. [64] present an emulation of a server based regular read/write storage in a synchronous message-passing system that is subject to “mobile Byzantine failures". They prove that the problem is impossible to solve in an asynchronous setting. The system size, however, is fixed and mobility, in this work refers to the Byzantine agents that can be moved from server to server. Baldoni et al. [65] provide the first emulation of a Byzantine-tolerant *safe* [52] register in an eventually synchronous system with churn but the size of the system is upper bounded by a known parameter. To the best of our knowledge, there isn't much work on implementing a shared register in a dynamic system with no upper bound on the system size and where servers are subject to Byzantine faults.

The first contribution of this work is an algorithm that emulates an atomic multi-reader/multi-writer register that does not require churn to ever stop, does not have an upper bound on the system size and tolerates up to a constant number of Byzantine servers in the system.Although our algorithm requires that there be a constant known upper bound on the number of Byzantine servers, this restriction is unavoidable as shown in our second contribution. This is an impossibility result that shows that it is impossible to emulate an atomic register in a system with churn if the system size and maximum number of servers that can be Byzantine in the system is unknown to the nodes.

Our system model is similar to the one in [17]. We assume that there exists a parameter D, an upper bound, unknown to the nodes, on the delay of any message (between correct nodes). There is no lower bound on message delays and nodes do not have real time clocks. It is proved in [17], that it is impossible to solve consensus in this model. Churn is modeled as follows: we assume that in any time interval of length D, the number of servers that enter or leave the system is at most a constant fraction, α (known to all nodes), of the number of servers in the system at the beginning of the interval. Our register emulation sacrifices atomicity when this constraint on churn is violated (as shown in [17]). We also assume the messages are authenticated with *digital signatures* [66]. In real world systems digital signatures in messages are implemented using public-key signatures [67] and message authentication codes [68]. Intuitively, this means that Byzantine servers cannot lie about the sender of a message.

Our algorithm is called BCCReg, for *Byzantine Continuous Churn Register*. It is loosely based on the algorithm in [17]. There are several challenges with working with Byzantine servers in a dynamic system. Unlike crash faults, data may easily be corrupted by Byzantine servers by sending old information, modified information or even different information to different sets of nodes while replying to a particular message. Byzantine servers may choose to not reply to a message at all, even if it is active or they may even choose to reply to a single message multiple times. Our algorithm uses a masking mechanism where at every stage of the algorithm, we wait for at least $f+1$ replies from distinct servers before taking any major steps, to make sure at least one reply from a non-faulty server was received. We also have a procedure to check if a message from a server should be ignored either because it sent out multiple replies to a query or because it lied about leaving the system. While updating the local register values with information from other servers, we need to make sure that only writes that are confirmed by more than f servers are considered.

3.2.2 Model

Each node in the system is either a client or a server. Each node p takes steps triggered by entering the system ($\textsc{Enter}_p$), leaving the system ($\textsc{Leave}_p$), or receiving a message ($\textsc{Receive}_p$). When a node p takes a step, its output can be a message to be broadcast either to all the servers or all the clients. If p is a client then it also takes steps triggered by the invocation of a read ($\textsc{Read}_p$) or a write ($\textsc{Write}_p$). The output of a step by client p can include a response of $\textsc{Join}_p$ (after entering), $\textsc{Return}_p(v)$ (after invoking a read), or $\textsc{Ack}_p$ (after invoking a write).

A node is *present* in the system at time t if it has entered but not left by time t. The number of servers present at time t is denoted $NS(t)$ and is called the *system size*. A node is *active* at time t if it is present at time t and has not crashed by time t (present servers are always active but a client that has crashed is present but not active).

There are four key system parameters: the maximum message delay D, the churn rate α, the maximum number of Byzantine servers f, and the minimum system size $NS_{min}(f)$ which is a function of f. An execution of the system must satisfy the following:

A1: At every time t, $NS(t)$ is finite and at least $NS_{min}(f)$.

A2-A4: Every message broadcast (either to all servers or to all clients) is received at most once by each node; only messages broadcast are received; if an intended recipient p is active throughout $[t, t+D]$, where t is the send time, then p receives the message; and the delay of every received message is at most D.

A5: For all times t, the number of ENTER and LEAVE events for servers during $[t, t+D]$ is at most $\alpha \cdot NS(t)$.

A6: Clients experience only crash failures and any number can crash. Servers can experience (authenticated) Byzantine faults and up to f can be faulty.

A7: Reads and writes are not invoked on a client until that client has joined.

A8: At most one operation (read or write) is pending at a time at a given client.

An algorithm is correct if each of its executions satisfies:

C1: Every client that enters the system and does not leave or crash eventually joins.

C2: If a client does not leave or crash, then it eventually produces a response for each operation (RETURN for READ, and ACK for WRITE).

C3: The read and write operations are atomic [51, 52, 53]: there is an ordering of all completed reads and writes and some subset of the uncompleted writes such that every read returns the value of the latest preceding write (or the initial value of the register if there is no preceding write) and, if an operation op_1 finishes before another operation op_2 begins, then op_1 occurs before op_2 in the ordering.

Although our model places an upper bound on message delays, it does not place any lower bound on the message delays or on local computation times. Moreover, nodes cannot access clocks to measure the passage of real time. Consequently, the well-known consensus problem is unsolvable in our model as proved in [17], just as it is unsolvable in a model with no upper bound on message delays [5].

3.2.3 Impossibility of a Uniform Algorithm with Byzantine Servers

An algorithm is called *uniform* if the code run by every node is independent of both the system size and the maximum number of Byzantine servers in the system. Thus in a uniform algorithm, for any particular node id p, there is only one state machine that node p can have regardless of when it enters the system, the system size, or the maximum number of Byzantine servers.

Theorem 38. *It is impossible to simulate an atomic read/write register in our model with a uniform algorithm.*

Proof. Suppose in contradiction, there is a uniform algorithm, $\mathbb{A}$, which simulates an atomic register and can tolerate f Byzantine server failures, as long as the system size is at least $NS_{min}(f)$, for some NS_{min} and any $f \in \mathbb{N}^+$. Consider the following executions of $\mathbb{A}$.

Execution e_1: The maximum number of Byzantine servers is 1. The set of servers in the system initially is S_1, where $|S_1| = NS_{min}(1)$, and all servers are correct. All message delays are D. The initial value of the simulated register is v. A new client p enters the system at time t_e and joins by time $t_j \geq t_e$. Client p invokes a read on the simulated register at time $t_r > t_j$. No other operation on the simulated register is invoked. By assumed correctness of algorithm $\mathbb{A}$, the read invoked by p returns v at some time $t'_r \geq t_r$.

Execution e'_1: Multiply the real time of every event in e_1 by $\min\{\frac{D}{t'_r}, 1\}$. As a result, all events in the time interval $[0, t'_r]$ in e_1 are compressed into the interval [0,D] in e'_1.

Execution e_2: The maximum number of Byzantine servers is $f_2 = |S_1|$. The set of servers in the system initially is S_2, where $|S_2| = N_{min}(f_2)$ and S_1 is a subset of S_2. There is at least one client in the system initially. All message delays are D. The initial value of the simulated register is v. No churn happens in this execution. Client q that was in the system at time 0 invokes a write of $v' \neq v$ at time t_w. By the assumed correctness of algorithm $\mathbb{A}$, the write completes at some time $t'_w \geq t_w$. No other operation on the simulated register is invoked.

Finally we construct the prefix e_3 of a new execution from executions e'_1 and e_2. First, we specify a set of timed views[2] and then we show that this set indeed forms the prefix of an

[2] A *timed view* for a node is the restriction of the execution to just the events involving that node, together with the real times when the events occur.

execution. Let e_3^1 be the set of timed views in e_2. Note that e_3^1 includes the write operation invoked by client q. Truncate each each timed view in e_3^1 immediately after the latest step with associated time at most t'_w, i.e., just after the write by client q finishes. Then append steps that result in the immediate delivery of all messages that are in transit at t'_w. Call the resulting set of timed views e_3^2. Construct e_3 from e_3^2 as follows.

Execution e_3: Add to the set the prefix of the timed view of client p from e'_1 that ends at time D, but change the time associated with each step by adding t'_w to it. For each server s in S_1, append the prefix of s's timed view in e'_1 that ends at time D, but change the time associated with step by adding t'_w to it. Append nothing to the timed views for the remaining nodes (client or server).

The idea behind e_3 is to have all the nodes behave correctly through q's write of v', and then have a new client p enter, join, and invoke a read during which time it communicates only with the servers in S_1. However, the servers in S_1 are Byzantine and start acting as they did in e'_1, causing p's read to incorrectly return the value v, instead of v'. An important technicality in the construction of e_3 is to adjust the time of steps taken from e'_1.

In order for the existence of the incorrect read by p in e_3 to contradict the assumed correctness of $\mathbb{A}$, we must show that e_3 is the prefix of an execution (otherwise, bad behavior by $\mathbb{A}$ is irrelevant).

We show that e_3 is a prefix of an execution by verifying properties A_1 through A_8. A_1, A_5, and A_6–A_8 are clear.

A_2–A_4: Every message sent by a node (client or server) has exactly one matching receipt. We show this in two parts: (i) If the message was sent before t'_w, it was either delivered before t'_w (from e_2) or at t'_w if it was pending at t'_w (from construction of e_3). (ii) If the message was sent after t'_w: Messages exchanged between p and S_1 after t'_p are all delivered within D time (from e'_1) and all other messages in e_3 after t'_w are delivered with delay D. All message delays in e'_1 are $\leq D$ and the message delays in e_2 are D. Therefore, the message delays in e_3 are at most D.

In e_3, p's read returns v, whereas the latest preceding write wrote $v' \neq v$. The value returned by node p is incorrect and as e_3 is the prefix of an execution, this violates the safety property of the register. Therefore, it is impossible to simulate a shared register in dynamic systems where new nodes entering have no information about the system size and no information on the number of Byzantine servers present in the system. □

3.2.4 The BCCReg Algorithm

In [17], all nodes in the system have a server thread and a client thread. This work introduces a new model where the set of clients is disjoint from the set of servers in the system, at most f servers can be Byzantine and any number of clients can crash. We do not allow clients to be Byzantine because, a Byzantine client can maliciously contact separate sets of servers and write different values which results in an inconsistent register thus violating safety. The BCCReg algorithm is loosely based on the algorithm in [17] along with modifications to accommodate the new client-server model. It is divided into two main parts: Algorithm 8 for servers and Algorithm 9 for clients. Algorithm 10 contains a set of common procedures used by both servers and clients nodes. The server algorithm contains a mechanism for tracking the composition of the system with respect to servers and for assisting clients with reads and writes. The client algorithm is for newly entered clients to join the system and for joined clients to read from and write to the shared register. Note that the algorithm in [17] is based on [9] and [10].

Each node p maintains a set of events, $Server\_Changes_p$, concerning the servers that have entered, joined and left the system. A node p also maintains the set $Present_p$ that stores information about servers that have entered, but have not left, as far as p knows. A server p is called a *member* if it has joined the system but not left. Client p maintains the derived variable $Members_p$ of servers that p considers as members. The variables val_p, num_p and w_id_p store the latest register value and its timestamp known by p. The set $Known\_Writes[\mathrm{q}]_p$ stores the values of the writes that server q claims to know about.

Algorithm 8: When a server p enters the system, it broadcasts to all the servers an enter message requesting information about prior events. When a server q finds out that node p has entered (or joined or left) the system, q updates $Server\_Changes_p$ accordingly and sends out an echo message with information about the system (stored in $Server\_Changes_p$) and the shared register (stored in the variable $Known\_Writes[\mathrm{p}]_p$). When node p receives at least $f + 1$ enter-echo messages from joined servers (to make sure at least one reply is

from a correct server), it calculates the number of replies it needs in order to join using γ and $Present_p$. Setting γ is a key challenge in the algorithm as setting it too small might not propagate updated information, whereas setting it too large might not guarantee termination of the join. The algorithm sends out messages that are authenticated with digital signatures. As a result Byzantine servers can send out incorrect information about everything except for node ids. Algorithm 8 is run by servers and at any point in this algorithm, Byzantine servers can modify information about anything sent out in messages, subject to the following restrictions:

- $Server\_Changes_p$: Byzantine servers can only send out subsets of the $Server\_Changes_p$ set. They cannot modify entries as each entry in this variable contains a server node id which was digitally signed by the sending server.

- val, num and w_id: Byzantine servers can modify variables val and num. But it cannot modify the w_id variable which is a client node id or $\perp$.

- $Known\_Writes[]_p$: Byzantine servers can send out subsets of $Known\_Writes[]_p$, but cannot add entries. For an entry $(val, (num, w\_id))$ in $Known\_Writes[q]_p$, Byzantine servers can modify the val and num variables of this entry for node q.

The JoinProtocol$_p$ procedure in Algorithm 10 is used by both newly entered servers and clients to join the system. Once joined, servers can reply to read/write queries from clients.In addition to that, for all nodes p, there exists an in-built procedure, IsClient$_p(q)$ that can check from a node id, q if q is a client or not. This procedure helps check whether Byzantine servers are pretending to be clients.

Algorithm 9: Clients might be in the system from the start or may enter the system at any time. Similar to servers, a newly entered client p, runs the JoinProtocol$_p$ procedure in Algorithm 10 to join the system. Clients treat both read and write operations in a similar manner. Both operations start with a read phase, which requests the current value of the register, using a query message, followed by a write phase, using an update message. A

Algorithm 6 BCCReg—Local Variables for server p.

```
In-built Procedure:
IsClient($q$)    // returns $true$ if $q$ is a client and $false$ if $q$ is a server
Local Variables:
$Server\_Changes$    // set that stores information about entering, leaving and joining of
                     // servers known by $p$. Initially $\{enter(q) \mid q \in S_0\} \cup \{join(q) \mid q \in S_0\}$,
                     // if $p$ is in the system at time 0 and $\emptyset$ otherwise
$join\_bound$    // if non-zero, the number of enter-echo messages $p$ should receive before joining;
                 // initially 0
$enter\_echo\_counter$    // number of enter-echo messages received so far; initially 0
$enter\_echo\_from\_joined\_counter$    // number of enter-echo messages from joined servers
                                       // received so far; initially 0
$is\_joined$    // Boolean to check if $p$ has joined the system; initially $false$
$val$    // latest register value known to $p$; initially $\perp$
$num$    // sequence number of latest value known to $p$; initially 0
$w\_id$    // id of node that wrote latest value known to $p$; initially $\perp$
$Known\_Writes[]$    // map from the set of node ids to the powerset of value-timestamp pairs.
                     // Initially each entry is $\emptyset$
Derived Variables:
$Present = \{q \mid enter(q) \in Server\_Changes \wedge leave(q) \notin Server\_Changes\}$
$valid\_val$ = value-timestamp pair with latest timestamp that occurs in at least $(f+1)$ elements
               of $Known\_Writes[]$, else $(\perp, (0, \perp))$
```

write operation broadcasts to all servers the new value it wishes to write, together with a timestamp, which consists of a sequence number that is one larger than the largest sequence number it has seen and its id that is used to break ties. A read operation just broadcasts to all servers the value it is about to return, keeping its sequence number as is. As in [9], write-back is needed to ensure the atomicity of read operations. Both the read phase and the write phase wait to receive sufficiently many reply messages. The fraction β is used to calculate the number of messages that should be received (stored in the rw_bound local variable) based on the size of the $Members_p$ set, for the operations to terminate. Setting β is also a key challenge in the algorithm as setting it too small might not return/update correct information from/to the register, whereas setting it too large might not guarantee termination of the reads and writes. The fraction β also has to ensure that enough replies from correct servers are heard so that these replies can efficiently mask incorrect replies from Byzantine servers.

Algorithm 10: The JoinProtocol() procedure helps newly entered nodes to join the

Algorithm 7 BCCREG—Code for server p.

```
When ENTER$_p$ occurs:
$Server\_Changes :=$
        $Server\_Changes \cup \{enter(p)\}$
s-bcast ⟨“enter”, $p$⟩
c-bcast ⟨“server-info”, $Server\_Changes$⟩

When RECEIVE$_p$⟨“enter”, $q$⟩ occurs:
if IsValidMessage( “enter", $q$) then
    $Server\_Changes :=$
            $Server\_Changes \cup \{enter(q)\}$
    s-bcast ⟨“enter-echo”, $Server\_Changes$,
        $Known\_Writes[p]$, $is\_joined$, $q, p$⟩
    c-bcast ⟨“server-info”,
            $Server\_Changes$⟩

When RECEIVE$_p$⟨“enter-client”, $q$⟩
        occurs:
if IsClient($q$) then
    c-bcast ⟨“enter-client-echo”,
        $Server\_Changes$, $Known\_Writes[p]$,
          $is\_joined$, $q, p$⟩

When RECEIVE$_p$⟨“enter-echo”, $C$, $K$,
        $j$, $q, r$⟩ occurs:
if IsValidMessage( “enter-echo",
        $q$, $r$) then
    $Server\_Changes :=$
            $Server\_Changes \cup C$
    if $(j = true)$ then
        $Known\_Writes[r] :=$
                $Known\_Writes[r] \cup K$
    if $\neg is\_joined \wedge (p = q)$ then
        call JoinProtocol($j$)
    call SetValueTimestamp()

When RECEIVE$_p$⟨“joined”, $q$⟩ occurs:
if IsValidMessage( “joined", $q$) then
    $Server\_Changes := Server\_Changes$
    $\cup\{enter(q), join(q)\}$
    s-bcast ⟨“joined-echo”, $q, p$⟩
    c-bcast ⟨“server-info”, $Server\_Changes$⟩

When RECEIVE$_p$⟨“joined-echo”, $q$, $s$⟩
        occurs:
if IsValidMessage( “joined-echo",
        $q$, $s$) then
    $Server\_Changes :=$
        $Server\_Changes \cup \{enter(q), join(q)\}$
    c-bcast ⟨“server-info”, $Server\_Changes$⟩

When LEAVE$_p$ occurs:
$Server\_Changes :=$
        $Server\_Changes \cup \{leave(p)\}$
s-bcast ⟨“leave”, $p$⟩
c-bcast ⟨“server-info”, $Server\_Changes$⟩
halt

When RECEIVE$_p$⟨“leave”, $q$⟩ occurs:
if IsValidMessage( “leave", $q$) then
    $Server\_Changes :=$
            $Server\_Changes \cup \{leave(q)\}$
    s-bcast ⟨“leave-echo”, $q, p$⟩
    c-bcast ⟨“server-info”, $Server\_Changes$⟩

When RECEIVE$_p$⟨“leave-echo”, $q$, $s$⟩
        occurs:
if IsValidMessage( “leave-echo", $q$, $s$) then
    $Server\_Changes :=$
            $Server\_Changes \cup \{leave(q)\}$
    c-bcast ⟨“server-info”, $Server\_Changes$⟩

When RECEIVE$_p$⟨“query”, $rt$, $q$⟩ occurs:
if $is\_joined \wedge$ IsClient($q$) then
    c-bcast ⟨“reply”, $Known\_Writes[p], rt, q, p$⟩

When RECEIVE$_p$⟨“update”, $(v, s, i), wt, q$⟩
        occurs:
if IsClient($q$) then
    if $(s, i) > (num, w\_id)$ then
        $(val, num, w\_id) := (v, s, i)$
        $Known\_Writes[p] := Known\_Writes[p] \cup$
                $\{(val, num, w\_id)\}$
    if $is\_joined$ then
        c-bcast ⟨“ack”, $wt, q, p$⟩
    s-bcast ⟨“update-echo”, $Known\_Writes[p]$,
            $p$ ⟩

When RECEIVE$_p$⟨“update-echo”, $K, s$⟩
        occurs:
$Known\_Writes[s] := Known\_Writes[s] \cup K$
call SetValueTimestamp()
```

Algorithm 8 BCCReg—Local Variables for client p.

```
In-built Procedure:
IsClient($q$)    // returns $true$ if $q$ is a client and $false$ if $q$ is a server
Local Variables:
$Server\_Changes$    // set that stores information about entering, leaving and joining of servers
                     // known by $p$. Initially $\{enter(q) \mid q \in S_0\} \cup \{join(q) \mid q \in S_0\}$, if $p$ is in
                     // the system at time 0 and $\emptyset$ otherwise
$enter\_echo\_counter$    // number of enter-echo messages received so far; initially 0
$enter\_echo\_from\_joined\_counter$    // number of enter-echo messages from joined servers
                                       // received so far; initially 0
$is\_joined$    // Boolean to check if $p$ has joined the system; initially $false$
$val$    // latest register value known to $p$; initially $\perp$
$num$    // sequence number of latest value known to $p$; initially 0
$w\_id$    // id of node that wrote latest value known to $p$; initially $\perp$
$Known\_Writes[]$    // map from set of node ids to the powerset of value-timestamp pairs.
                     // Initially each entry is $\emptyset$
$temp$    // temporary storage for the value being read or written; initially 0
$tag$    // used to uniquely identify read and write phases of an operation; initially 0
$rw\_bound$    // the number of replies/acks $p$ should receive before finishing a read/write phase;
               // initially 0
$rw\_counter$    // the number of replies/acks received so far for a read/write phase; initially 0
$rp\_pending$    // Boolean indicating whether a read phase is in progress; initially $false$
$wp\_pending$    // Boolean indicating whether a write phase is in progress; initially $false$
$read\_pending$    // Boolean indicating whether a read is in progress; initially $false$
$write\_pending$    // Boolean indicating whether a write is in progress; initially $false$
Derived Variables:
$Present = \{q \mid enter(q) \in Server\_Changes \wedge leave(q) \notin Server\_Changes\}$
$Members = \{q \mid join(q) \in Server\_Changes \wedge leave(q) \notin Server\_Changes\}$
$valid\_val$ = value-timestamp pair with latest timestamp that occurs in at least $(f+1)$ elements
            of $Known\_Writes[]$, else $(\perp, (0, \perp))$
```

Algorithm 9 BCCReg—Code for client, p.

```
When $\text{Enter}_p$ occurs:
s-bcast ⟨"enter-client", $p$⟩

When $\text{Receive}_p$⟨"enter-client-echo", $C$, $K$, $j$, $q, r$⟩ occurs:
if IsValidMessage( "enter-client-echo", $q$) $\wedge$ $(p = q)$ then
    $Server\_Changes := Server\_Changes \cup C$
    if $(j = true)$ then
        $Known\_Writes[r] := Known\_Writes[r] \cup K$
    if $\neg is\_joined \wedge (p = q)$ then
        call JoinProtocol($j$)
call SetValueTimestamp()

When $\text{Receive}_p$⟨"server-info", $C$⟩ occurs:
$Server\_Changes := Server\_Changes \cup C$

Procedure BeginReadPhase()
$tag$++
s-bcast ⟨"query", $tag$, $p$⟩
$rw\_bound := \beta \cdot |Members|$
$rw\_counter := 0$
$rp\_pending := true$

When $\text{Receive}_p$⟨"reply", $K, rt, q, s$⟩ occurs:
if IsValidMessage( "reply", $q$, $rt$, $s$) then
    if $rp\_pending \wedge (rt = tag) \wedge (q = p)$ then
        $rw\_counter$++
        $Known\_Writes[s] := Known\_Writes[s] \cup K$
    if $rw\_counter \geq rw\_bound$ then
        call SetValueTimestamp()
        $rp\_pending := false$
        call BeginWritePhase()

Procedure BeginWritePhase()
if $write\_pending$ then
    $val := temp$
    $num$++
    $w\_id := p$
if $read\_pending$ then
    $temp := val$
s-bcast ⟨"update", $(temp, num, w\_id)$,$tag, p$⟩
$rw\_bound := \beta \cdot |Members|$
$rw\_counter := 0$
$wp\_pending := true$

When $\text{Receive}_p$⟨"ack", $wt, q, s$⟩ occurs:
if IsValidMessage( "ack", $q$, $wt$, $s$) then
    if $wp\_pending \wedge (wt = tag) \wedge (q = p)$ then
        $rw\_counter$++
        if $rw\_counter \geq rw\_bound$ then
            $wp\_pending := false$
            if $read\_pending$ then
                $read\_pending := false$
                generate $\text{Return}(temp)$ response
            if $write\_pending$ then
                $write\_pending := false$
                generate Ack response

When $\text{Leave}_p$ occurs:
halt
```

system. The other procedures in this algorithm are used to deal with Byzantine servers and their arbitrary nature. The procedure SetValueTimestamp() checks and updates the value-timestamp triple $((val, (num, w\_id))_p)$ to $valid\_val_p$ if the timestamp of $valid\_val_p$ is higher than the latest known $(num, w\_id)_p$ pair. The variable $valid\_val_p$ is necessary to make sure that before writing any value (learned from other servers) to the register, the value was seen by at least $f+1$ servers. A Byzantine server p may send out more than one reply for a given message or keep replying after it has sent out a $leave_p$ message. The three IsValidMessage() procedures deal with these situations. They check to make sure that only one reply from each server for a message is processed by all nodes. They also check whether the sender q has already sent a $leave_q$ message. Reads and writes invoked by Byzantine servers are ignored by correct servers by the IsClient() checks on Lines 35 and 37 in Algorithm 9.

The correctness of BCCReg relies on the system parameters α, f, and NS_{min} satisfying the following constraints, for some choice of algorithm parameters β and γ:

$$\alpha \le 1 - 2^{-1/4} \approx 0.159 \tag{3.14}$$

$$1 \le (1-\alpha)^3 NS_{min} - 2f \tag{3.15}$$

$$\gamma \ge \frac{1+2f}{(1-\alpha)^3 NS_{min}} + \frac{(1+\alpha)^3}{(1-\alpha)^3} - 1 \tag{3.16}$$

$$\gamma \le \frac{(1-\alpha)^3}{(1+\alpha)^3} - \frac{f}{(1+\alpha)^3 NS_{min}} \tag{3.17}$$

$$\beta \le \frac{(1-\alpha)^3}{(1+\alpha)^2} - \frac{f}{(1+\alpha)^2 NS_{min}} \tag{3.18}$$

$$\beta > \frac{(1+\alpha)^5 - 1 + 2f/NS_{min}}{(1-\alpha)^4 - f/NS_{min}} \tag{3.19}$$

$$\beta > \frac{(1+\alpha)^3 - (1-\alpha)^3 + 1 + (1+3f)/NS_{min}}{[(2+2\alpha+\alpha^2)(1-\alpha)^2(1+\alpha)^{-2}] - 2f/NS_{min}} \tag{3.20}$$

Constraint (3.14) is an upper bound on the churn rate to ensure that not too many servers can leave the system in an interval of length $4D$. Constraint (3.15) is a lower bound on the minimum system size to ensure that at least $f+1$ correct servers are in the system

Algorithm 10 BCCREG—Procedures used by client/server p

```
Procedure JoinProtocol($j$)
$enter\_echo\_counter ++$
if $(j = true) \wedge (join\_bound = 0)$ then
    $enter\_echo\_from\_joined\_counter ++$
    if $enter\_echo\_from\_joined\_counter > f$ then
        $join\_bound := \gamma \cdot |Present|$
if $enter\_echo\_counter \geq join\_bound > 0$ then
    $is\_joined := true$
    if  IsClient($p$) then
        generate JOINED$_p$ response
    else
        $Server\_Changes := Server\_Changes \cup \{join(p)\}$
        s-bcast ⟨“joined”, $p$⟩
        c-bcast ⟨“server-info”, $Server\_Changes$⟩

Procedure SetValueTimestamp()
if $valid\_val \neq \perp$ then
    if timestamp of $valid\_val > (num, w\_id)$ then
        $(val, num, w\_id) := valid\_val$
        $Known\_Writes[p] := Known\_Writes[p] \cup$
      $\{(val, num, w\_id)\}$

Procedure IsValidMessage( $type$, $r$)
if $type =$ (“enter"∨ “joined"∨“leave") ∧ $(leave(r) \notin Server\_Changes)$ then
    return $true$ if this is the first $type$ message received from $r$,
            else return $false$

Procedure IsValidMessage( $type$, $q$, $r$)
if $type =$ (“enter-echo" ∨ “enter_client-echo" ∨
          “joined-echo" ∨ “leave-echo")
        ∧ $(leave(r) \notin Server\_Changes)$ then
    return $true$ if this is the first $type$ message for $q$ received from $r$,
           else return $false$

Procedure IsValidMessage( $type$, $q$, $tag$, $r$)
if $type =$ (“reply" ∨ “ack") ∧ $(leave(r) \notin Server\_Changes)$  then
    return $true$ if this is the first $type$ message for $q$ with sequence $tag$ received
            from $r$, else return $false$
.
```

throughout an interval of length $3D$ encompassing the time a node enters, thus ensuring that the newly entered node successfully terminates its joining protocol. Constraint (3.16) ensures that the *join_bound* fraction, γ, is large enough such that updated information about the system is obtained by an entered node before it joins the system. Constraint (3.17) ensures that γ is small enough such that for all entered nodes, a join operation terminates if the entered node is not Byzantine or it does not leave or crash. Constraint (3.18) ensures that the *rw_bound* fraction, β, is small enough such that termination of reads and writes is guaranteed. Constraints (3.19) and (3.20) ensure that β is large enough such that atomicity is not violated by read and write operations. above constraints are satisfied. In all consistent sets of parameters, the churn rate α is never more than 0.05. The algorithm can tolerate any size of f as long as NS_{min} is proportionally big. Table 3.3 provides a few sets of values for system parameters f,NS_{min} and α and algorithm parameters γ and β that satisfy Constraints 3.14 to 3.20

BCCReg violates atomicity if Assumption A5 is violated.

3.2.5 Correctness Proof of BCCReg

We will show that BCCReg satisfies the properties C1 to C3 listed at the end of Section 3.2.2. Lemmas 39 through 46 are used to prove Theorem 47, which states that every client and any correct server eventually joins, provided it does not crash or leave. Lemmas 49 through 51 are used to prove Theorem 52, which states that every operation invoked by a client that remains active eventually completes. Lemmas 55 through 35 are used to prove Theorem 37, which states that atomicity is satisfied.

Consider any execution. We begin by bounding the number of servers that enter during an interval of time and the number of servers that are present at the end of the interval, as compared to the number present at the beginning.

Lemmas 39 and 40 bound the maximum number of servers that can enter and/or leave the system in any interval of time. Lemma 41 proves that at least $f+1$ correct servers are active throughout any interval of length $3D$. This lemma is necessary to ensure that at all times,

system parameters			algorithm parameters	
maximum failures (f)	minimum system size (NS_{min})	churn rate (α)	$join\_bound$ fraction (γ)	rw_bound fraction (β)
1	8	0	N/A	0.86
1	10	0.01	0.82	0.84
1	13	0.02	0.79	0.80
1	190	0.05	0.79	0.80
2	19	0.01	0.80	0.83
2	24	0.02	0.81	0.82
2	347	0.05	0.70	0.77
5	44	0.01	0.80	0.83
5	57	0.02	0.79	0.82
5	826	0.05	0.79	0.82
10	85	0.01	0.80	0.83
10	113	0.02	0.79	0.82
10	1630	0.05	0.79	0.82
100	838	0.01	0.79	0.82
100	1107	0.02	0.79	0.82
100	16015	0.05	0.79	0.82
1000	8360	0.01	0.79	0.82
1000	11042	0.02	0.79	0.82
1000	159935	0.05	0.79	0.82

Table 3.3: Values for the BCCReg parameters that satisfy constraints (3.14) to (3.20)

an active node (client or server) that expects replies, hears back from at least $f+1$ correct servers in order to mask the bad information sent by Byzantine servers. Lemmas 44 to 46 show that information about correct servers entering, joining, and leaving is propagated to active clients and correct servers properly, via the *Server_ Changes* sets. Lemmas 39 through 46 are used to prove Theorem 47 which states that every client and every correct server eventually completes the join protocol in Algorithm 10, provided it does not crash or leave.

3.2.5.1 Proof that Join Protocol Terminates

Lemma 39. *For all $i \in \mathbb{N}$ and all $t \geq 0$, at most $((1+\alpha)^i - 1)NS(t)$ servers enter during $(t, t+Di]$ and $(1-\alpha)^i NS(t) \leq NS(t+Di) \leq (1+\alpha)^i NS(t)$.*

Refer to Lemma 16 for the proof. We are also interested in the number of servers that leave during an interval of time. The calculation of the maximum number of servers that leave during an interval is complicated by the possibility of servers entering during the interval, allowing additional servers to leave.

Lemma 40. *For $\alpha > 0$, all nonnegative integers $i \leq -1/\log_2(1-\alpha)$ and every time $t \geq 0$, at most $(1-(1-\alpha)^i)NS(t)$ servers leave during $(t, t+Di]$.*

Refer to Lemma 17 for the proof.

Recall that a server is *active* at time t if it has entered by time t, but has not left by time t. The next lemma shows that there are $f+1$ correct servers that remain active throughout any interval of length $3D$.

Lemma 41. *For every $t > 0$, at least $f+1$ correct servers are active throughout $[\max\{0, t-2D\}, t+D]$.*

Proof. Let S be the set of servers present at time $t' = \max\{0, t-2D\}$, so $|S| = NS(t') \geq NS_{min}$. Constraint (3.14) implies that $-1/\log_2(1-\alpha) \geq 4 \geq 3$. So, by Lemma 40, at most $(1-(1-\alpha)^3)|S|$ servers leave during $(t', t+D]$ and there are at least $(1-\alpha)^3|S|$ servers present throughtout time interval $(t', t + D]$. At any point in time, there are at most f Byzantine servers in the system. Thus, at least

$$(1-\alpha)^3|S| - f \geq (1-\alpha)^3 NS_{min} - f$$

correct servers in S are active at time $t + D$. By Constraint (3.15), $(1-\alpha)^3 NS_{min} - f \geq f + 1$, so at least $f + 1$ correct servers in S are still active at time $t + D$. □

Below, a local variable name is superscripted with t to denote the value of that variable at time t; e.g., v_p^t is the value of node p's local variable v at time t.

In the analysis, we will frequently be comparing the data in nodes' *Server_ Changes* sets to the set of ENTER, JOINED, and LEAVE events that have actually occurred. To facilitate this comparison, we define a set $\mathit{SysInfo}^I$ that contains perfect information about correct servers for the time interval I. For each server q, let t_q^e, and t_q^ℓ be the times when the events ENTER_q, and LEAVE_q occur, and let t_q^j be the time when server q sends out a joined message. Similarly, for each client q, let t_q^e, t_q^j, and t_q^ℓ be the times when the events ENTER_q, JOINED_q, and LEAVE_q occur, respectively.

Recall that S_0 is the set of servers that were in the system initially. If $q \in S_0$, then we set $t_q^e = t_q^j = 0$. Then we have:

$$\mathit{SysInfo}^I = \{\mathit{enter}(q) \mid t_q^e \in I\} \cup \{\mathit{join}(q) \mid t_q^j \in I\} \cup \{\mathit{leave}(q) \mid t_q^\ell \in I\}.$$

In particular,

$$\mathit{SysInfo}^{[0,0]} = \{\mathit{enter}(q) \mid q \in S_0\} \cup \{\mathit{join}(q) \mid q \in S_0\}.$$

Since a client or correct server p that is active throughout $[t_p^e, t + D]$ directly receives all enter, joined, and leave messages broadcast by active clients or correct servers during $[t_p^e, t]$, within D time, we have:

Observation 42. *For every client and any correct server p and all times $t \geq t_p^e$, if p is* *active at time* $t + D$*, then* $\mathit{SysInfo}^{[t_p^e,t]} \subseteq \mathit{Changes}_p^{t+D}$.

Let C_0 be the set of clients that are in the system initially. By assumption, for every node $p \in S_0 \cup C_0$, $\mathit{SysInfo}^{[0,0]} \subseteq \mathit{Server\_Changes}_p^0$, and hence Observation 42 implies:

Observation 43. *For every client and any correct server* $p \in S_0 \cup C_0$*, if* p *is active at time* $t \geq 0$*, then* $\mathit{SysInfo}^{[0,\max\{0,t-D\}]} \subseteq \mathit{Changes}_p^t$.

The purpose of Lemmas 44, 45, and 46 is to show that information about servers entering, joining, and leaving is propagated properly, via the *Server_ Changes* sets. From now on, enter messages refer to both enter and enter-client messages and enter-echoes refer to both enter-echoes and enter-client-echoes.

Lemma 44. *Suppose that, at time* T''*, a client or correct server* $p \notin S_0 \cup C_0$ *receives an enter-echo message from a correct server* q *sent at time* T' *in reply to an enter message from* p*. Let* T *be any time such that* $\max\{0, T'' - 2D\} \leq T \leq t_p^e$*. Suppose* p *is active at time* $T + 2D$ *and* q *is active throughout* $[U, T + D]$*, where* $U \leq \max\{0, T'' - 2D\}$*. Then* $\mathit{SysInfo}^{(U,T]} \subseteq \mathit{Server\_Changes}_p^{T+2D}$.

Proof. The proof is adapted from Lemma 21 to include Byzantine servers.

□

Lemma 45. *For every client and any correct server* p*, if* p *is active at time* $t \geq t_p^e + 2D$*, then* $\mathit{SysInfo}^{[0,t-D]} \subseteq \mathit{Changes}_p^t$.

Proof. The proof is adapted from Lemma 22 to include Byzantine servers.

□

Lemma 46. *For every client and any correct server* $p \notin S_0 \cup C_0$*, if* p *joins at time* t_p^j *and is active at time* $t \geq t_p^j$*, then* $\mathit{SysInfo}^{[0,\max\{0,t-2D\}]} \subseteq \mathit{Changes}_p^t$.

Proof. The proof is by induction on the order in which clients and correct servers join the system. Let $p \notin S_0 \cup C_0$ be a client or correct server that joins at time $t_p^j \leq t$ and suppose the claim holds for all clients and correct servers that join before p. If $t \geq t_p^e + 2D$, then the claim follows by Lemma 45. So, suppose $t < t_p^e + 2D$.

Before joining, p receives $f+1$ enter-echo message from joined servers in reply to its enter message (Line number 93). Out of these, at most f can be from Byzantine servers. Thus, at least one reply is from a correct server. Suppose p receives the first enter-echo message at time t'' sent by correct server q at time t'; $t_p^e \leq t' \leq t'' \leq t_p^j$. From Lemma 45, we know that this message from q has a perfect information about the $\mathit{Server\_Changes}^{t'-2D}$ set. This in turn means that it has perfect information about the derived set $\mathit{Present}^{t'-2D}$. Byzantine servers can only modify the information about the *Server_ Changes* set by sending a subset of its *Server_ Changes* set. So, when node p receives at least one reply is from a correct server, the incomplete information sent by Byzantine servers is overshadowed by this one reply from q and thus p has a perfect information about $\mathit{Present}^{t'-2D}$.

If correct server $q \in S_0$, then by Observation 43, $\mathit{SysInfo}^{[0,\max\{0,t'-D\}]} \subseteq \mathit{Changes}_q^{t'}$. Otherwise, by the induction hypothesis, $\mathit{SysInfo}^{[0,\max\{0,t'-2D\}]} \subseteq \mathit{Changes}_q^{t'}$, since q joined prior to p and is active at time $t' \geq t_q^j$. Note that $\mathit{Server\_Changes}_q^{t'} \subseteq \mathit{Server\_Changes}_p^{t''}$ $\subseteq \mathit{Server\_Changes}_p^{t}$. If $t \leq 2D$, then $\max\{0, t-2D\} = 0$ and the claim holds.

If $t > 2D$, then let S be the set of servers present at time $\max\{0, t'-2D\}$; $|S| = N(\max\{0, t'-2D\})$. By Lemma 40 and Constraint (3.14), at most $(1-(1-\alpha)^3)|S|$ servers leave during $(\max\{0, t'-2D\}, t'+D]$. Since $t'' \leq t'+D$, it follows that $|\mathit{Present}_p^{t''}| \geq |S| - (1-(1-\alpha)^3)|S| = (1-\alpha)^3|S|$. Hence, from lines 94 and 95 of Algorithm 10, p waits until it has received at least $\mathit{join\_bound} = \gamma \cdot |\mathit{Present}_p^{t''}| \geq \gamma \cdot (1-\alpha)^3|S|$ enter-echo messages before joining.

By Lemma 39, at most $((1+\alpha)^3 - 1)|S|$ servers enter during $(\max\{0, t'-2D\}, t'+D]$. Thus, at time $t'+D$, at most $(1+\alpha)^3|S|$ servers are present, at most f of which are Byzantine.

Hence, the number of enter-echo messages p receives before joining from servers that were active throughout $[\max\{0, t'-2D\}, t'+D]$ is $\mathit{join\_bound}$ minus the total number of server enters, leaves and faults (as Byzantine servers may not reply at all), which is at least

$$\begin{aligned}&\gamma\cdot(1-\alpha)^3|S| - [((1+\alpha)^3-1)|S| + (1-(1-\alpha)^3)|S| + f]\\&\quad = [(1+\gamma)(1-\alpha)^3-(1+\alpha)^3]|S| - f \geq [(1+\gamma)(1-\alpha)^3-(1+\alpha)^3]NS_{min} - f \quad (3.21)\end{aligned}$$

Rearranging Constraint (3.16), we get

$$[(1+\gamma)(1-\alpha)^3-(1+\alpha)^3]NS_{min} - f \geq f+1,$$

so expression (3.21) is at least $f+1$. Hence p receives an enter-echo message at some time $T'' \leq t_p^j$ from a correct server q' that is active throughout

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

Let T' be the time that q' sent its enter-echo message in reply to the enter message from p. Applying Lemma 44 for q', with $U = \max\{0, t'-2D\}$, and $T = t-2D$ gives $\mathit{SysInfo}^{(\max\{0,t'-2D\},t-2D]} \subseteq \mathit{Server\_Changes}_p^t$.

Thus, we get $\mathit{SysInfo}^{[0,t-2D]} = \mathit{SysInfo}^{[0,\max\{0,t'-2D\}]} \cup \mathit{SysInfo}^{(\max\{0,t'-2D\},t-2D]}$ $\subseteq \mathit{Server\_Changes}_p^t$. □

Next we prove that every client and any correct server that remains active sufficiently long after it enters, will succeed in joining.

Theorem 47. *Every client and any correct server $p \notin S_0 \cup C_0$ that is active at time $t_p^e + 2D$ joins by time $t_p^e + 2D$.*

Proof. The proof is by induction on the order in which clients and correct servers enter the system. Let $p \notin S_0 \cup C_0$ be a client or correct server that enters at time t_p^e and is active at time $t_p^e + 2D$. Suppose the claim holds for all client and correct servers that enter before p.

By Lemma 41, there are $f+1$ correct servers that are active throughout $[\max\{t_p^e -$

$2D, 0\}, t_p^e + D]$. Let q be one such server. If $q \in S_0$, then q joins at time 0. If not, then $t_q^e \leq t_p^e - 2D$, so, by the induction hypothesis, q joins by time $t_q^e + 2D \leq t_p^e$. Since q is active at time $t_p^e + D$, it receives the enter message from p during $[t_p^e, t_p^e + D]$ and sends an enter-echo message in reply. Since p is active at time $t_p^e + 2D$, it receives the enter-echo message from q by time $t_p^e + 2D$. Hence, by time $t_p^e + 2D$, p receives at least one enter-echo message from a correct joined server in reply to its enter message.

Suppose the first enter-echo message p receives from a correct joined server in reply to its enter message is sent by server q' at time t' and received by p at time t''. By Lemma 46, $\mathit{SysInfo}^{[0, \max\{0, t'-2D\}]} \subseteq \mathit{Changes}_{q'}^{t'} \subseteq \mathit{Server\_Changes}_p^{t''}$.

Let S be the set of servers present at time $\max\{0, t' - 2D\}$. Since $t'' \leq t' + D$, it follows from Lemma 39 that at most $((1+\alpha)^3 - 1)|S|$ servers enter during $(\max\{0, t' - 2D\}, t'']$. Thus, $|\mathit{Present}_p^{t''}| \leq |S| + ((1+\alpha)^3 - 1)|S| = (1+\alpha)^3|S|$. From line 94 in Algorithm 10, it follows that $\mathit{join\_bound} \leq \gamma \cdot (1+\alpha)^3|S|$.

By Lemma 40 and Constraint (3.14), at most $(1 - (1-\alpha)^3)|S|$ servers leave during $(\max\{0, t' - 2D\}, t' + D]$. At most f servers are Byzantine at $t' + D$. Since $t_p^e \leq t' \leq t_p^e + D$, the servers in S that do not leave during $(\max\{0, t' - 2D\}, t' + D]$ and are not Byzantine at $t' + D$ are active throughout $[t_p^e, t_p^e + D]$ and send enter-echo messages in reply to p's enter message. By time $t_p^e + 2D$, p receives all these enter-echo messages. There are at least

$$|S| - (1 - (1-\alpha)^3)|S| - f = (1-\alpha)^3|S| - f$$

such enter-echo messages. By Constraint (3.17),

$$\frac{(1-\alpha)^3}{(1+\alpha)^3} - \frac{f}{(1+\alpha)^3 NS_{min}} \geq \gamma,$$

so the value of $join\_bound$ is at most

$$\gamma \cdot (1 + \alpha)^3|S| \;\leq\; \left(\frac{(1-\alpha)^3}{(1+\alpha)^3} - \frac{f}{(1+\alpha)^3 NS_{min}}\right) \cdot (1 + \alpha)^3|S| \;=\; (1 - \alpha)^3|S| - f.$$

Thus, by time $t_p^e{+}2D$, the condition in line 95 of Algorithm 10 holds and node p joins. □

3.2.5.2 Proof that Reads and Writes by Clients Terminate

Next, we show that all read and write operations terminate. Specifically, we show that the number of replies for which an operation waits is at most the number that it is guaranteed to receive.

Since $enter(q)$ is added to $Server\_Changes_p$ whenever $join(q)$ is, for server q, we get the following observation.

Observation 48. *For every time $t \geq 0$ and every client p that is active at time t, $Members_p^t \subseteq Present_p^t$.*

Lemma 49 relates a node's current estimate of the number of servers present to the number of servers that were present in the system $2D$ time units earlier. Lemma 50 relates a client's current estimate of the number of servers that are members to the number of servers that were present in the system $4D$ time units earlier. Lemma 49 is used in the proof of Lemma 51 and Lemma 50 is used in the proof of Theorem 60.

Lemma 49. *For every client and any correct server p and every time $t \geq t_p^j$ at which p is active,*

$$(1 - \alpha)^2 \cdot N(\max\{0, t - 2D\}) - f \;\leq\; |Present_p^t| \;\leq\; (1 + \alpha)^2 \cdot N(\max\{0, t - 2D\}).$$

Proof. The proof is adapted from Lemma 26 to include f Byzantine servers in the lower bound.

□

Lemma 50. *For every client p and every time $t \geq t_p^j$ at which p is active,*

$$(1 - \alpha)^4 \cdot N(\max\{0, t - 4D\}) - f \; \leq \; |Members_p^t| \; \leq \; (1 + \alpha)^4 \cdot N(\max\{0, t - 4D\}).$$

Proof. The proof is adapted from Lemma 27 to include f Byzantine servers in the lower bound.

□

The next lemma proves a lower bound on the number of servers that reply to a client's query or update message.

Lemma 51. *If a client or correct server p is active at time $t \geq t_p^j$, then the number of correct servers that are joined by time t and are still active at time $t + D$ is at least $\left[\frac{(1-\alpha)^3}{(1+\alpha)^2}\right] \cdot |Present_p^t| - f$.*

Proof. By Lemma 40 and Constraint (3.14), the maximum number of servers that leave during $(\max\{0, t - 2D\}, t + D]$ is at most $(1 - (1 - \alpha)^3) \cdot N(\max\{0, t - 2D\})$. Thus, there are at least

$$\begin{aligned} &N(\max\{0, t - 2D\}) - (1 - (1 - \alpha)^3) \cdot N(\max\{0, t - 2D\}) - f \\ &\qquad\qquad = [(1 - \alpha)^3] \cdot N(\max\{0, t - 2D\}) - f \end{aligned}$$

correct servers that were present at time $\max\{0, t - 2D\}$ and are still active at time $t + D$. This number is bounded below by $\left[\frac{(1-\alpha)^3}{(1+\alpha)^2}\right] \cdot |Present_p^t| - f$ since, by Lemma 49, $N(\max\{0, t - 2D\}) \geq |Present_p^t|/(1 + \alpha)^2$. By Theorem 47, all of these servers are joined by time t. □

Theorem 52. *Every read or write operation invoked by a client that remains active completes.*

Proof. Each operation consists of a read phase and a write phase. We show that each phase terminates within $2D$ time, provided the client remains active (does not crash or leave).

Consider a phase of an operation by client p that starts at time t. Every correct server that joins by time t and is still active at time $t + D$ receives p's query or update message and replies with a reply message or an ack message by time $t + D$. By Lemma 51, there are at least $\left[\frac{(1-\alpha)^3}{(1+\alpha)^2}\right] \cdot |Present_p^t| - f$ such servers.

From Constraint (3.18), Lemma 49 and Observation 48,

$$\left[\frac{(1-\alpha)^3}{(1+\alpha)^2}\right] \cdot |Present_p^t| - f \geq \beta \cdot |Present_p^t| \geq \beta \cdot |Members_p^t| = rw\_bound_p^t.$$

Thus, by time $t + 2D$, p receives sufficiently many replies or ack messages to complete the phase. □

3.2.5.3 Proof of Atomicity of BCCReg

Now we prove atomicity of the BCCReg algorithm. Let $\mathcal{T}$ be the set of read operations that complete and write operations that execute line 74 of Algorithm 9. For any node p, let $ts_p^t = (num_p^t, w\_id_p^t)$ denote the *timestamp* of the latest value known to node p that is recorded in its $Known\_Writes[p]_p$ variable. Note that new timestamps are created by write operations (on lines 70-71 of Algorithm 9) and are sent via enter-echo, update, and update-echo messages. Initially, $ts_p^0 = (0, \perp)$ for all nodes p.

For any operation o in $\mathcal{T}$ by client p, the *timestamp of its read phase*, $ts^{rp}(o)$, is ts_p^t, where t is the end of its read phase (i.e., when the condition on line 64 of Algorithm 9 evaluates to true). The *timestamp of its write phase*, $ts^{wp}(o)$, is ts_p^t, where t is the beginning of its write phase (i.e., when it s-bcasts on line 74 of Algorithm 9). The *timestamp of a read operation* in $\mathcal{T}$ is the timestamp of its read phase. The *timestamp of a write operation* in $\mathcal{T}$ is the timestamp of its write phase.

Note that w_id is equal to p and num is set to one greater than the largest sequence number occurring in at least $f + 1$ replies observed during an operation's read phase. This implies the next observation:

Observation 53. *Each write operation in $\mathcal{T}$ has a unique timestamp.*

The next observation follows by a simple induction, since every timestamp other than $(0, \perp)$ comes from Lines 70-71 of Algorithm 9.

Observation 54. *Consider any read op_1 in $\mathcal{T}$. If the timestamp of a read op_1 is $(0, \perp)$, then op_1 returns $\perp$. Otherwise, there is a write op_2 in $\mathcal{T}$ such that $ts(op_1) = ts(op_2)$ and the value returned by op_1 equals the value written by op_2.*

If a read operation op_1 returns the value written by a write operation op_2, then we say that op_1 reads from op_2.

Lemmas 55–35 show that write phase information propagates properly through the system. They are analogous to Observation 43 and Lemmas 44–46, regarding the propagation of information about server Enter, Joined, and Leave events.

Lemma 55. *If o is an operation in $\mathcal{T}$ whose write phase w starts at t_w, correct server p is active at time $t \geq t_w + D$, and $t_p^e \leq t_w$, then $ts_p^t \geq ts^{wp}(o)$.*

Proof. Since server p is active throughout $[t_w, t_w + D]$, it directly receives the update message s-bcast by w at time t_w. Hence, from lines 38–40 of Algorithm 8, $ts_p^t \geq ts^{wp}(o)$. □

Lemma 56. *Suppose a correct server $p \notin S_0$ receives $(f+1)$ enter-echo messages from correct servers by time t''. Let the $f + 1$st enter-echo message from a correct server be received from q that sends it at time t' in reply to an enter message from p. If o is an operation whose write phase w starts at t_w, p is active at time $t \geq \max\{t'', t_w + 2D\}$, and the $f + 1$ correct servers that send enter-echo messages are active throughout $[t_w, t_w + D]$, then $ts_p^t \geq ts^{wp}(o)$.*

Proof. By Lemma 41, there are at least $f+1$ correct joined servers that are active throughout $[t_w, t_w + D]$. Since q is active throughout $[t_w, t_w + D]$, it receives the update message from w at some time $\hat{t} \in [t_w, t_w + D]$, so $ts_q^{\hat{t}} \geq ts^{wp}(o)$. At time $t'' \leq t$, p receives the enter-echo sent by q at time t'. By the above argument, all f earlier enter-echo messages have timestamp $\geq ts^{wp}(o)$. So, the value of the timestamp in *valid_val*$_p$ and in *Known_ Writes*$[p]_p$ is set to $\geq ts^{wp}(o)$. So $ts_p^t \geq ts_p^{t''} \geq ts_q^{t'}$. If $t' \geq \hat{t}$, then $ts_q^{t'} \geq ts_q^{\hat{t}}$, so $ts_p^t \geq ts^{wp}(o)$. If $\hat{t} > t'$, then q

sends an update-echo at time $\hat{t} \leq t_w + D$, and p receives it by time $\hat{t} + D \leq t_w + 2D \leq t$. The same argument works for the other f correct, active servers in the system. Thus the timestamp of the variable $valid\_val_p$ and $Known\_Writes[p]_p$ is either the timestamp of w or of a later write. Thus, $ts_p^t \geq ts_q^{\hat{t}} \geq ts^{wp}(o)$. □

Lemma 57. *If o is an operation in $\mathcal{T}$ whose write phase w starts at t_w and correct server p is active at time $t \geq \max\{t_p^e + 2D, t_w + D\}$, then $ts_p^t \geq ts^{wp}(o)$.*

Proof. The proof is by induction on the order in which correct servers enter the system. Suppose the claim holds for all correct servers that enter before p. If $t_p^e \leq t_w$, which is the case for all $p \in S_0$, then the claim follows from Lemma 55.

If $t_w < t_p^e$, then by Lemma 41, there are at least $f+1$ correct joined servers that are active throughout $[\max\{0, t_p^e - 2D\}, t_p^e + D]$. These servers receive an enter message from p and send an enter-echo message containing $ts_q^{t'}$ back to p. Let q be the server whose enter-echo is the $(f+1)$th enter-echo from a correct joined server to reach p. Let server q receive the enter message from p at some time $t' \in [t_p^e, t_p^e + D]$. The enter-echo message sent by q is received by p at some time $t'' \leq t' + D \leq t_p^e + 2D \leq t$. So, the value of the timestamp in $valid\_val_p$ and in turn $Known\_Writes[p]_p$ for p is set to $ts_p^t \geq ts_p^{t''} \geq ts_q^{t'}$

The first case is when $t_w \geq \max\{0, t_p^e - 2D\}$. Since $t_w + D < t_p^e + D$, it follows that the $f+1$ correct joined servers including q are active throughout $[t_w, t_w + D]$. Furthermore, $t \geq t_p^e + 2D \geq \max\{t'', t_w + 2D\}$. Hence, Lemma 56 implies that $ts_p^t \geq ts^{wp}(o)$.

The second case is when $t_w < \max\{0, t_p^e - 2D\}$. Since $t_w \geq 0$, it follows that $t_p^e - 2D > 0$, $t_q^e \leq \max\{0, t_p^e - 2D\} = t_p^e - 2D$, and $t_w < t_p^e - 2D \leq t' - 2D$, so $t' \geq \max\{t_q^e + 2D, t_w + D\}$. Note that q is active at time t' and q enters before p, so, by the induction hypothesis, $ts_q^{t'} \geq ts^{wp}(o)$. The above argument is true for all the other f correct joined servers that p hears from. Hence, $ts_p^t \geq ts^{wp}(o)$.

□

Lemma 58. *If o is an operation in $\mathcal{T}$ whose write phase starts at t_w, correct server $p \notin S_0$ joins at time t_p^j, and p is active at time $t \geq \max\{t_p^j, t_w + 2D\}$, then $ts_p^t \geq ts^{wp}(o)$.*

Proof. The proof is adapted from Lemma 35 to tolerate f Byzantine servers . □

Lemma 59 is the key lemma for proving atomicity of BCCReg. It shows that for two non-overlapping operations in $\mathcal{T}$, the timestamp of the read phase of the latter operation is at least as big as the timestamp of the write phase of the former. Theorem 60 uses Lemma 59 to show that the timestamps of two non-overlapping operations respect real time ordering and completes the proof of atomicity.

Lemma 59. *For any two operations op_1 and op_2 in $\mathcal{T}$, if op_1 finishes before op_2 starts, then $ts^{wp}(op_1) \leq ts^{rp}(op_2)$.*

Proof. Let p_1 be the client that invokes op_1, let w denote the write phase of op_1, let t_w be the start time of w, and let $\tau_w = ts^{wp}(op_1) = ts_{p_1}^{t_w}$. Similarly, let p_2 be the client that invokes op_2, let r denote the read phase of op_2, let t_r be the start time of r, and let $\tau_r = ts^{rp}(op_2) = ts_{p_2}^{t_r}$.

Let Q_w be the set of servers that p_1 hears from during w (i.e., that sent messages causing p_1 to increment *rw_counter* on line 80 of Algorithm 9) and Q_r be the set of servers that p_2 hears from during r (i.e., that sent messages causing p_2 to increment *rw_counter* on line 62 of Algorithm 9). Let $P_w = |Present_{p_1}^{t_w}|$ and $M_w = |Members_{p_1}^{t_w}|$ be the sizes of the *Present* and *Members* sets belonging to p_1 at time t_w, and $P_r = |Present_{p_2}^{t_r}|$ and $M_r = |Members_{p_2}^{t_r}|$ be the sizes of the *Present* and *Members* sets belonging to p_2 at time t_r.

Case I: $t_r > t_w + 2D$. We start by showing that there exists $f+1$ correct servers in Q_r such that $t_q^j \leq t_r - 2D$.

Each server $q \in Q_r$ receives and responds to r's query, so q is joined by time $t_r + D$. By Theorem 47, the number of servers that can join during $(t_r - 2D, t_r + D]$ is at most the number of servers that can enter in $(\max\{0, t_r - 4D\}, t_r + D]$. By Lemma 39, the number of servers that can enter during $(\max\{0, t_r - 4D\}, t_r + D]$ is at most $((1+\alpha)^5 - 1) \cdot N(\max\{0, t_r - 4D\})$. By Lemma 50, $(1-\alpha)^4 N(\max\{0, t_r - 4D\}) - f \leq M_r$.

From the code and Constraint (3.19), it follows that

$$\begin{aligned}|Q_r| \geq \beta M_r &> \left[\frac{(1+\alpha)^5 - 1 + 2f/NS_{min}}{(1-\alpha)^4 - f/NS_{min}}\right] \cdot M_r \\ &\geq \left[\frac{(1+\alpha)^5 - 1 + 2f/NS_{min}}{(1-\alpha)^4 - f/NS_{min}}\right] \cdot ((1-\alpha)^4 N(\max\{0, t_r - 4D\}) - f) \\ &\geq [(1+\alpha)^5 - 1] \cdot N(\max\{0, t_r - 4D\}) + 2f\end{aligned}$$

which is $2f+1$ more than the maximum number of servers that can enter in $(\max\{0, t_r - 4D\}, t_r + D)$. At most f of these can be Byzantine. Thus, at least $f+1$ correct servers in Q_r join by time $t_r - 2D$.

Suppose correct server $q \in Q_r$ receives r's query message at time $t' \geq t_r \geq t_w + 2D$. If $q \in S_0$, then $t_q^j = 0 \leq t_w$, so, by Lemma 55, $ts_q^{t'} \geq ts^{wp}(op_1) = \tau_w$. Otherwise, $q \notin S_0$, so $0 < t_q^j \leq t_r - 2D < t'$. Since $t_w + 2D < t_r \leq t'$, Lemma 35 implies that $ts_q^{t'} \geq ts^{wp}(op_1) = \tau_w$. In either case, q responds to r's query message with a timestamp at least as large as τ_w and, hence, $\tau_r \geq \tau_w$.

Case II: $t_r \leq t_w + 2D$. Let $J = \{p \mid t_p^j < t_r \text{ and } p \text{ is an active server at time } t_r\} \cup \{p \mid t_r \leq t_p^j \leq t_r + D\}$, which contains the set of all servers that reply to r's query. By Theorem 47, all correct servers that are present at time $t_r - 2D$ join by time t_r if they remain active. Therefore all servers in J are either active at time $\max\{0, t_r - 2D\}$ or enter during $(\max\{0, t_r - 2D\}, t_r + D]$. By Lemma 39, $|J| \leq (1+\alpha)^3 N(\max\{0, t_r - 2D\})$.

Let K be the set of all servers that are present at time $\max\{0, t_r - 2D\}$ and do not leave during $(\max\{0, t_r - 2D\}, t_r + D]$. Note that K contains all the servers in Q_w that do not leave during $[t_w, t_r + D] \subseteq [\max\{0, t_r - 2D\}, t_r + D]$. By Lemma 40 and Constraint (3.14), at most $(1 - (1-\alpha)^3)N(\max\{0, t_r - 2D\})$ servers leave during $[\max\{0, t_r - 2D\}, t_r + D]$.

From the code, $|Q_r| \geq \beta M_r$ and, by Lemma 50, $M_r \geq (1-\alpha)^4 N(\max\{0, t_r - 4D\}) - f$.

So,

$$|Q_r| \geq \beta \left[(1-\alpha)^4 N(\max\{0, t_r - 4D\}) - f\right].$$

Similarly, $|Q_w| \geq \beta M_w \geq \beta\left[(1-\alpha)^4 N(\max\{0, t_w - 4D\}) - f\right]$. Therefore, the size of K is at least

$$\begin{aligned}|K| &\geq |Q_w| - (1-(1-\alpha)^3)N(\max\{0, t_r - 2D\}) - f \\ &\geq \beta\left[(1-\alpha)^4 N(\max\{0, t_w - 4D\}) - f\right] - (1-(1-\alpha)^3)N(\max\{0, t_r - 2D\}) - f.\end{aligned}$$

Since $t_r - t_w \leq 2D$, it follows that $\max\{0, t_r - 4D\} - \max\{0, t_w - 4D\} \leq 2D$. By Lemma 39, $N(\max\{0, t_r - 4D\}) \leq (1+\alpha)^2 \cdot N(\max\{0, t_w - 4D\})$. Thus we can replace $N(\max\{0, t_w - 4D\})$ in the above expression with $(1+\alpha)^{-2} \cdot N(\max\{0, t_r - 4D\})$ and get the following expression for $|Q_r| + |K|$:

$$\begin{aligned}|Q_r| + |K| \geq & \beta[(1-\alpha)^4 N(\max\{0, t_r - 4D\}) - f] \\ & + \beta[(1-\alpha)^4(1+\alpha)^{-2} N(\max\{0, t_r - 4D\}) - f] \\ & - (1-(1-\alpha)^3)N(\max\{0, t_r - 2D\}) - f \\ = & \beta[(1-\alpha)^4(1+\alpha)^{-2}(2+2\alpha+\alpha^2)N(\max\{0, t_r - 4D\}) - 2f] \\ & - (1-(1-\alpha)^3)N(\max\{0, t_r - 2D\}) - f.\end{aligned}$$

By Lemma 39, $N(\max\{0, t_r - 4D\}) \geq (1-\alpha)^{-2} N(\max\{0, t_r - 2D\})$. Thus,

$$\begin{aligned}|Q_r| + |K| \geq & \beta[(1-\alpha)^2(1+\alpha)^{-2}(2+2\alpha+\alpha^2)N(\max\{0, t_r - 2D\}) - 2f] \\ & - (1-(1-\alpha)^3)N(\max\{0, t_r - 2D\}) - f\end{aligned}$$

By Constraint (3.20),

$$\beta > \frac{(1+\alpha)^3 - (1-\alpha)^3 + 1 + (1+3f)/NS_{min}}{[(2+2\alpha+\alpha^2)(1-\alpha)^2(1+\alpha)^{-2}] - 2f/NS_{min}}.$$

Let $N_{tr2D} = N(\max\{0, t_r - 2D\})$. So,

$$\begin{aligned} |Q_r| + |K| \geq & \left(\frac{\left((1+\alpha)^3 - (1-\alpha)^3 + 1 + \frac{(1+3f)}{NS_{min}}\right)}{((2+2\alpha+\alpha^2)(1-\alpha)^2(1+\alpha)^{-2}) - 2f/NS_{min}} \right) \\ & \cdot \left((2+2\alpha+\alpha^2)(1-\alpha)^2(1+\alpha)^{-2} - \frac{2f}{N_{tr2D}} \right) N_{tr2D} \\ & - (1-(1-\alpha)^3)]N_{tr2D} - f \end{aligned}$$

Since, $N_{tr2D} \geq NS_{min}$, we get

$$\begin{aligned} |Q_r| + |K| &\geq \left((1+\alpha)^3 - (1-\alpha)^3 + 1 + (1+3f)/NS_{min}\right) N_{tr2D} \\ &\quad - f - (1-(1-\alpha)^3)]N_{tr2D} - f \\ &\geq (1+\alpha)^3 N_{tr2D} + 2f + 1 \geq |J| + 2f + 1. \end{aligned}$$

This implies that the intersection of K and Q_r has at least $2f+1$ servers. For each servers p in the intersection, $ts_p \geq \tau_w$ when p sends its reply to r. Since at most f servers can be Byzantine, there are at least $f+1$ correct servers that reply with $ts_p \geq \tau_w$. Thus the timestamp of $valid\_val_p$ on Line number 103 is $\geq \tau_w$, thus, $\tau_w \leq \tau_r$. □

Theorem 60. BCCREG *ensures atomicity.*

Proof. The proof is adapted from Theorem 37.

□

4. CONCLUSION

4.1 Conclusion and Future Work

This dissertation describes my research to provide fault-tolerant distributed services in the network topology and middleware layers of a distributed system in message-passing models.

In Section 2.1 we have implemented the eventually perfect failure detector ($\Diamond P$) in an arbitrary partitionable network model composed of ADD channels which experience unbounded message loss and unbounded message delay for a majority of the messages. This work is an important step towards understanding the minimal assumptions on network topology, message sizes, reliability of channels and partial synchrony necessary to implement this oracle. The algorithm is quite practical for sparsely connected graphs as the number of paths between two nodes (and the message size) will be $\ll n!$. Even though the message size for this algorithm is bounded, can we do better than our current results using smaller messages or fewer messages? We think that these are important questions that need to be answered in the future.

In Section 3.1 we have shown how to emulate an atomic read/write register in a crash-prone system where nodes can enter and leave continually and there is no upper bound on the system size. Our churn model places a limit on the number of nodes entering and leaving during each time interval of length D as a fraction of the number of nodes in the system at the beginning of the interval. This definition is easy to state and does not depend on the way our algorithm works. Our failure model requires the number of crashed nodes to be at most a fraction of the nodes in the system. Separating crashes, which are unannounced, from leaves, which are announced, allows more flexibility. We also proved a lower bound showing that the existence of churn makes it impossible to achieve the same level of failure-resiliency as in the static case.

There are a number of directions for future work. A natural question is whether the small churn rate and failure fraction of our algorithm can be improved, perhaps with a tighter analysis. Proving additional lower bounds or tradeoffs on these parameters is one interesting avenue. However, it might be possible to completely avoid the bound α on the churn rate. To prevent the number of nonfaulty nodes from becoming too small, a node might need to obtain permission before leaving, similarly to what our algorithm does for joining. This might enable an algorithm to ensure atomicity even when churn rate is high. Or, perhaps some of the ideas in [69] can be adapted to obtain a modification of our algorithm that ensures this.

Since the number of crashed nodes must never exceed a fixed fraction of the nodes present in the system, the system can get into a situation in which no more nodes can crash or leave unless more nodes enter. If crashed nodes could be detected in some way, then they could be treated as nodes that have left, thus freeing up the ability for more nodes to crash or leave. If some mechanism outside the system identifies crashed nodes and informs nodes in the system, then leave messages can be sent on behalf of these crashed nodes, analogously to [11] and [12]. It may even be possible to use ongoing, but bounded, churn to detect crashed nodes, rather than relying on an out-of-system mechanism.

The communication complexity of our algorithm grows without bound. Our algorithm sends increasingly large *Changes* sets. The amount of information communicated might be reduced by sending only recent events, or by removing very old events. Furthermore, the unbounded counters might be avoided with ideas from [70].

In Section 3.2 we provide an algorithm that emulates a Byzantine-tolerant atomic register in a dynamic system that never stops changing and has no upper bound on the system size. We also provide an impossibility proof that in our model, the maximum number of Byzantine servers cannot be described as a fraction of the current system size.

There are several directions for future work. The values of α, f and NS_{min} that satisfy our algorithm are quite restrictive. It will be nice to see if such restrictions are necessary or if they can be improved either with a better algorithm or a tighter correctness analysis.

Currently our model tolerates the most severe end of the fault severity spectrum and paper [17] considered the most benign end of the fault severity spectrum. The impossibility result in Section 3.2.3 showing that we cannot have Byzantine servers as a fraction of the system size is quite restrictive too. In future, we would like to investigate if the same impossibility extends to less severe failures like omission failures and timing failures [71]. Our work assumes that digital signatures are available for authenticated Byzantine fault tolerance. A different direction for future work is the implementation of these digital signatures in dynamic systems and explore its relation to network security.

The current way of restricting churn relies on the unknown upper bound D on message delay. Even though nodes have no way of measuring D and it was shown in [17] that consensus is impossible to solve in this model, it may be a bit confusing and we would like to explore an alternative way of bounding the churn that doesn't rely on the existence of D. An alternative is to define the churn rate with respect to messages in transit. For example, at all times, the number of servers that can enter/leave the system when any message is in transit at most α times the system size when the message was sent. It may be possible to prove that the two models are indeed equivalent, or to show that the algorithms still work, or can be modified to work, in the new model.

REFERENCES

[1] P. H. Gallager, R.G. and P. Spira, "A distributed algorithm for minimum weight spanning trees," pp. 66–77, 1979.

[2] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Commun. ACM*, vol. 8, no. 9, p. 569, 1965.

[3] L. Lamport, R. E. Shostak, and M. C. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

[4] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[5] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

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

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

[8] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Programming Languages and Systems*, vol. 12, pp. 463–492, July 1990.

[9] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *J. ACM*, vol. 42, pp. 124–142, Jan. 1995.

[10] N. A. Lynch and A. A. Shvartsman, "Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts," in *Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, pp. 272–281, 1997.

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

[12] 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*, pp. 173–190, 2002.

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

[14] S. Kumar and J. L. Welch, “Implementing $\diamond P$ with bounded messages on a network of ADD channels,” *CoRR*, vol. abs/1708.02906, 2017.

[15] S. Kumar and J. Welch, “Implementing $\diamond\mathbb{P}$ with bounded messages on a network of *add* channels,” in *Parallel Processing Letters*, vol. 29, 2019.

[16] H. Attiya, H. C. Chung, F. Ellen, S. Kumar, and J. L. Welch, “Simulating a shared register in an asynchronous system that never stops changing,” in *Proceedings of 29th International Symposium on Distributed Computing*, pp. 75–91, 2015.

[17] 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.

[18] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[19] S. Sastry and S. M. Pike, “Eventually perfect failure detectors using ADD channels,” in *Parallel and Distributed Processing and Applications, 5th International Symposium, ISPA 2007, Niagara Falls, Canada, August 29-31, 2007, Proceedings*, pp. 483–496, 2007.

[20] I. Abraham, Y. Amit, and D. Dolev, “Optimal resilience asynchronous approximate agreement,” in *Principles of Distributed Systems, 8th International Conference, OPODIS 2004, Grenoble, France, December 15-17, 2004, Revised Selected Papers*, pp. 229–239, 2004.

[21] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl, "Reaching approximate agreement in the presence of faults," *J. ACM*, vol. 33, no. 3, pp. 499–516, 1986.

[22] M. J. Fischer, N. A. Lynch, and M. Merritt, "Easy impossibility proofs for distributed consensus problems," in *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing, Minaki, Ontario, Canada, August 5-7, 1985*, pp. 59–70, 1985.

[23] J.-M. Chang and N. Maxemchuk, "Reliable broadcast protocols,," pp. 251–273, 1984.

[24] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[25] A. S. Tanenbaum, *Computer networks, 4th Edition.* Prentice Hall, 2002.

[26] F. C. Freiling, R. Guerraoui, and P. Kuznetsov, "The failure detector abstraction," *ACM Comput. Surv.*, vol. 43, no. 2, pp. 9:1–9:40, 2011.

[27] M. K. Aguilera, W. Chen, and S. Toueg, "Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks," *Theor. Comput. Sci.*, vol. 220, no. 1, pp. 3–30, 1999.

[28] W. Chen, S. Toueg, and M. K. Aguilera, "On the quality of service of failure detectors," *IEEE Trans. Computers*, vol. 51, no. 1, pp. 13–32, 2002.

[29] S. Sastry and S. M. Pike, "Eventually perfect failure detectors using ADD channels," in *Parallel and Distributed Processing and Applications, 5th International Symposium, ISPA 2007, Niagara Falls, Canada, August 29-31, 2007, Proceedings*, pp. 483–496, 2007.

[30] R. van Renesse, Y. Minsky, and M. Hayden, "A gossip-style failure detection service," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, (London, UK, UK), pp. 55–70, Springer-Verlag, 1998.

[31] M. Hutle, "An efficient failure detector for sparsely connected networks," in *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks, Innsbruck, Austria, February 17-19, 2004*, pp. 369–374, 2004.

[32] P. Fraigniaud, S. Rajsbaum, C. Travers, P. Kuznetsov, and T. Rieutord, "Perfect failure detection with very few bits," in *Stabilization, Safety, and Security of Distributed Systems - 18th International Symposium, SSS 2016, Lyon, France, November 7-10, 2016, Proceedings*, pp. 154–169, 2016.

[33] C. Fernández-Campusano, R. Cortiñas, and M. Larrea, "A performance study of consensus algorithms in omission and crash-recovery scenarios," in *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy, February 12-14, 2014*, pp. 240–243, 2014.

[34] E. Jiménez, S. Arévalo, and A. Fernández, "Implementing unreliable failure detectors with unknown membership," *Inf. Process. Lett.*, vol. 100, no. 2, pp. 60–63, 2006.

[35] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, 1991.

[36] R. Baldoni, S. Bonomi, and M. Raynal, "Implementing a regular register in an eventually synchronous distributed system prone to continuous churn," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 1, pp. 102–109, 2012.

[37] S. Y. Ko, I. Hoque, and I. Gupta, "Using tractable and realistic churn models to analyze quiescence behavior of distributed protocols," in *IEEE Symposium on Reliable Distributed Systems*, pp. 259–268, 2008.

[38] H. Attiya, "Efficient and robust sharing of memory in message-passing systems," *J. Alg.*, vol. 34, pp. 109–127, Jan. 2000.

[39] P. Dutta, R. Guerraoui, R. R. Levy, and A. Chakraborty, "How fast can a distributed atomic read be?," in *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pp. 236–245, 2004.

[40] R. Guerraoui and R. Levy, "Robust emulations of shared memory in a crash-recovery model," in *Proceedings of the International Conference on Distributed Computing Systems*, pp. 400–407, 2004.

[41] R. Guerraoui and M. Vukolić, "Refined quorum systems," in *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, pp. 119–128, 2007.

[42] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi, "Byzantine disk paxos: optimal resilience with Byzantine shared memory," *Dist. Comp.*, vol. 18, no. 5, pp. 387–408, 2006.

[43] A. S. Aiyer, L. Alvisi, and R. A. Bazzi, "Bounded wait-free implementation of optimally resilient byzantine storage without (unproven) cryptographic assumptions," in *Proceedings of 21st International Symposium on Distributed Computing*, pp. 7–19, 2007.

[44] J.-P. Martin, L. Alvisi, and M. Dahlin, "Minimal byzantine storage," in *Proceedings of the 16th International Conference on Distributed Computing*, pp. 311–325, 2002.

[45] D. Malkhi and M. K. Reiter, "Byzantine quorum systems," *Dist. Comp.*, vol. 11, no. 4, pp. 203–213, 1998.

[46] D. Malkhi, M. K. Reiter, A. Wool, and R. N. Wright, "Probabilistic quorum systems," *Information and Computation*, vol. 170, no. 2, pp. 184–206, 2001.

[47] M. Vukolic, *Quorum Systems: With Applications to Storage and Consensus*. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers, 2012.

[48] P. Musial, N. Nicolaou, and A. A. Shvartsman, "Implementing distributed shared memory for dynamic networks," *Communications ACM*, vol. 57, no. 6, pp. 88–98, 2014.

[49] L. Lamport, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[50] A. Spiegelman and I. Keidar, "On liveness of dynamic storage," *CoRR*, vol. abs/1507.07086, July 2015.

[51] L. Lamport, “On interprocess communication. part I: basic formalism,” *Distributed Computing*, vol. 1, no. 2, pp. 77–85, 1986.

[52] L. Lamport, “On interprocess communication. part II: algorithms,” *Distributed Computing*, vol. 1, no. 2, pp. 86–101, 1986.

[53] M. P. Herlihy and J. M. Wing, “Linearizability: A correctness condition for concurrent objects,” *Trans. on Prog. Lang. Sys.*, vol. 12, pp. 463–492, July 1990.

[54] B. Murphy and B. Levidow, “Windows 2000 dependability,” *IEEE International Conference on Dependable Systems and Networks.*, 2000.

[55] M. Castro and B. Liskov, “Practical byzantine fault tolerance and proactive recovery,” *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.

[56] M. Castro and B. Liskov, “Practical byzantine fault tolerance,” pp. 173–186, 1999.

[57] L. Alvisi, E. T. Pierce, D. Malkhi, M. K. Reiter, and R. N. Wright, “Dynamic byzantine quorum systems,” in *2000 International Conference on Dependable Systems and Networks (DSN 2000) (formerly FTCS-30 and DCCA-8), 25-28 June 2000, New York, NY, USA*, pp. 283–292, 2000.

[58] H. Attiya and A. Bar-Or, “Sharing memory with semi-byzantine clients and faulty storage servers,” in *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings.*, pp. 371–378, Oct 2003.

[59] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi, “Wait-free regular storage from byzantine components,” *Information Processing Letters*, vol. 101, no. 2, pp. 60 – 65, 2007.

[60] J. Martin, L. Alvisi, and M. Dahlin, “Minimal byzantine storage,” in *Distributed Computing, 16th International Conference, DISC 2002, Toulouse, France, October 28-30, 2002 Proceedings*, pp. 311–325, 2002.

[61] D. Malkhi and M. K. Reiter, “Byzantine quorum systems,” *Distributed Computing*, vol. 11, no. 4, pp. 203–213, 1998.

[62] D. Malkhi, M. K. Reiter, and A. Wool, “The load and availability of byzantine quorum systems,” *SIAM J. Comput.*, vol. 29, no. 6, pp. 1889–1906, 2000.

[63] W. S. Dantas, A. N. Bessani, J. d. S. Fraga, and M. Correia, “Evaluating byzantine quorum systems,” in *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pp. 253–264, Oct 2007.

[64] S. Bonomi, A. D. Pozzo, M. Potop-Butucaru, and S. Tixeuil, “Optimal mobile byzantine fault tolerant distributed storage: Extended abstract,” in *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pp. 269–278, 2016.

[65] R. Baldoni, S. Bonomi, and A. S. Nezhad, “A protocol for implementing byzantine storage in churn-prone distributed systems,” *Theor. Comput. Sci.*, vol. 512, pp. 28–40, 2013.

[66] Y. Lindell, A. Lysyanskaya, and T. Rabin, “On the composition of authenticated byzantine agreement,” *J. ACM*, vol. 53, no. 6, pp. 881–917, 2006.

[67] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Commun. ACM*, vol. 21, pp. 120–126, Feb. 1978.

[68] G. Tsudik, “Message authentication with one-way hash functions,” *SIGCOMM Comput. Commun. Rev.*, vol. 22, pp. 29–38, Oct. 1992.

[69] K. M. Konwar, N. Prakash, N. A. Lynch, and M. Médard, “RADON: repairable atomic data object in networks,” in *20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain*, pp. 28:1–28:17, 2016.

[70] A. Mostéfaoui and M. Raynal, “Two-bit messages are sufficient to implement atomic read/write registers in crash-prone systems,” in *Proceedings of the 2016 ACM Sympo-*

sium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016, pp. 381–389, 2016.

[71] F. Cristian, “Understanding fault-tolerant distributed systems,” *Commun. ACM*, vol. 34, no. 2, pp. 56–78, 1991.