Efficient Replication of Large Data Objects

by

Rui Fan

B.S. in Computer Science and Mathematics, Caltech (2000)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2003

Author ....
Department of Electrical Engineering and Computer Science
February 5, 2003

Certified by ....
Professor Nancy A. Lynch
NEC Professor of Software Science and Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Efficient Replication of Large Data Objects
by
Rui Fan

Submitted to the Department of Electrical Engineering and Computer Science on February 5, 2003, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering

Abstract

Replication is an important technique for improving the reliability and scalability of data services. The primary problem encountered in replication is the trade-off between amount of replication, performance, and consistency. A rule of thumb states that any replication algorithm must sacrifice at least one of these criteria. In this thesis, we investigate replicating large data objects, such as files, whose size is large compared to metadata used by the replication algorithm. With this assumption, we present a distributed replication algorithm which simultaneously achieves a high replication factor, nearly optimal performance, and strong data consistency. Furthermore, our algorithm makes only basic assumptions about its environment. Our algorithm works in any asynchronous, reliable message-passing network, without relying on higher level functions such as distributed locking or group communication. Our algorithm is suitable for implementation in both LAN and WAN settings.

This thesis is divided into two parts. In the first part, we formally state the assumptions and guarantees of our replication algorithm in terms of its trace properties. We then formally implement our algorithm in the IOA modeling language. We also give rigorous proofs of the algorithm's correctness and its performance analysis. The main idea of our algorithm is to separately maintain copies of the data, and information about the locations of the up-to-date copies. Our algorithm then mostly performs cheap operations on the location information, and avoids expensive operations on the actual data.

The second part of this thesis presents two lower bounds on the costs of data replication. The first lower bound gives the minimum number of writes that must occur during a read operation. The second lower bound states that for a certain class of efficient replication algorithms, the replicas must use storage proportional to the maximum number of concurrent writers. The motivation for these lower bounds was certain algorithmic techniques we used in our replication algorithm. The lower bounds suggest that these techniques are necessary. The lower bounds are also of independent interest.

Thesis Supervisor: Professor Nancy A. Lynch
Title: NEC Professor of Software Science and Engineering

Acknowledgments

I am grateful to Prof. Nancy Lynch for her guidance and her many insightful suggestions, improvements, and corrections. Prof. Lynch has also taught me invaluable lessons about writing in a clear and precise manner.

I would also like to thank my fellow students in the Theory of Distributed Systems group. Their ideas, comments and good humor were a great help to me in completing this thesis.

I dedicate this thesis to my parents.

Contents

List of Figures

Chapter 1

Introduction

Replication is a widely used technique for improving the performance and reliability of data services. In replication, multiple copies of a data item are created, and clients access the data by accessing the copies according to some protocol. Replication can reduce the latency of the data service by load-balancing client accesses across the copies. It also increases the fault tolerance of the service by making the data available even if some of the copies fail.

To be most useful, replication should be transparent to the user. Thus, a correctness requirement for many replication algorithms is *atomicity*, which allows the replicated service to exhibit the same behavior as an unreplicated service. However, atomicity imposes a trade-off between the performance and fault tolerance of the algorithm: the more faults the algorithm tolerates, the more copies of the data must be created, and the more work must be done for each operation on the data to make it atomic. Often, this trade-off makes it prohibitively expensive to implement a service with a high degree of replication. In the first part of this thesis, we present a new algorithm called *Layered Data Replication (LDR)* for replicating read/write data, which mitigates the performance/fault tolerance trade-off. In particular, *LDR* performs a nearly constant amount of communication for each read operation, independent of how many faults it must tolerate. For a write operation, the communication is proportional to the number of faults *LDR* tolerates. Because of the low cost of read and write operations, we can increase the fault tolerance of our algorithm and still achieve high performance. Thus, we can simultaneously realize both benefits of replication. In the second part of the thesis, we will prove two lower bounds on the communication and memory cost of any replication algorithm, which suggest that some of the constructions used in our algorithm are necessary.

1.1 Background

There is a wide body of literature on replicated data algorithms, for example, [3, 6, 12, 5, 7, 4]. These algorithms make different trade-offs between the consistency of the data and the performance of the algorithm. A replicated data algorithm is typically divided into two parts, *replica control* and *concurrency control* [15].

Replica control deals with which replicas are queried or updated during an operation. Concurrency control deals with how operations are serialized, and which operations are allowed to proceed in parallel. We will first discuss some relevant replica control algorithms, then discuss concurrency control algorithms. Lastly, we discuss an algorithm whose ideas we use in our algorithm, which combines replica control with concurrency control.

The simplest replica control algorithm is the *primary copy* method [1]. Here, a single replica is designated as a *primary*. A user write is directed at the primary, which processes the operation, then propagates the result to other replicas in the background. To read, a user first gets a timestamp for the latest value from the primary, then reads from *any* replica which has an equal timestamp. The advantage of the primary copy method is that the primary has knowledge of all the writes that occur, and thus can help "direct" the user to an up-to-date replica. The problem with this method is that the primary is a performance bottleneck and a single point of failure. However, we will make use of the primary-as-director idea, and solve the performance and fault-tolerance problem of the primary by, in effect, replicating the primary.

Another popular replica control algorithm is the *weighted voting* method [5]. Here, a user must read or write to a set of replicas during a read or write operation. The requirement on the sets is that the size of any read set plus the size of any write set must be greater than the total number of replicas. This ensures that any read and write operation intersect in at least one replica, and when combined with the appropriate concurrency control algorithm, ensures that a read will see the value of the last write. The advantage to weighted voting is that it tolerates the failures of some replicas. The disadvantage is that both read and write operations become slower, since they have to access multiple replicas. A natural adaptation of weighted voting is *quorum-based replication* [10]. Here, the quorums can be tuned to give improved read and write performance. But the inherent need to access multiple replicas still remains. Our algorithm will make use of quorum consensus. But the data we access using quorum consensus will be small, and so this does not hurt the performance of the algorithm too much.

Two interesting algorithms designed to mitigate the costs of quorum consensus are *voting with witnesses* [12], and *voting with ghosts* [14]. Both algorithms use the idea of creating some replicas that only store information about what the latest timestamp for the data is. Since the size of the timestamps is small, it is cheap to read them from a quorum of processors to determine the latest timestamp. This improves the latency of a write operation in quorum based replication algorithms, because the first phase of such algorithms typically consists of determining a timestamp for the write. It also helps a read operation determine the timestamp of a value it should return. However, the timestamp replicas don't directly give a way to directly find an up-to-date replica, so a read operation is still slow because it might have to read from multiple replicas to find one with an up-to-date copy of the data. Our algorithm uses a similar technique of separating the storage of the timestamp of the latest write, from storage of values of the write itself. However, our timestamp replicas also know the location of some replicas with the latest data. The main accomplishment of our algorithm consists in synchronizing the updates of the latest timestamp, the set of most up-to-date replicas, and the data itself, to achieve much better performance.

A number of algorithms use *lazy replication* [7] with *gossiping* [13, 4] to improve performance. The idea is to let the user return from a write operation after writing to only a few replicas. Then, more replicas are updated with the new write by random exchanges of information between replicas. A read operation can read from any replica. If enough time elapses between the read and the last write for the value of the write to propagate to many replicas, then the read sees the value of the last write. However, in general, lazy replication using gossiping does not guarantee atomicity. We make use of gossiping in our algorithm, but only as an optimization. That is, our algorithm always maintains atomicity, but uses gossiping to increase the number of up-to-date replicas, to increase the speed of reads by allowing a user to access a nearby replica.

We now discuss the concurrency control part of a replicated data algorithm. The most prevalent concurrency control algorithm is locking [3]. For example, in quorum based replication, each replica has a read and write lock. To access the data, a user must acquire locks from a quorum of replicas. The requirement is that only one user at a time can hold the write lock at a replica. Performing locking in a distributed system is a complicated problem, and requires expensive solutions [11]. In addition, to make locking fault-tolerant against users who fail while holding locks, we must make some assumptions about reliable fault detection. For this reason, we would like to avoid the use of locking for concurrency control.

One algorithm which circumvents the need for locking is the *ABD* algorithm [2] of Attiya, Bar-Noy and Dolev. *ABD* was originally designed to simulate shared memory in a message passing network. However, since atomic read/write shared memory and atomic read/write replicated data have the same semantics, *ABD* can be used as a simple, effective replicated data algorithm as well. The major disadvantage of *ABD* in this regard is that its read operation is slow. In fact, a read operation has embedded in it a write of the data to a quorum of replicas, so that reads are slower than writes. In a typical replicated data system, the number of reads is much larger than the number of writes. Thus, we would particularly like to optimize a replication algorithm to have fast reads. *LDR* does this by avoiding writing the data to any replicas during a read, and reading the data from only one replica. To do so, we make use of *ABD* to store the locations of the replicas with the most up-to-date value of the data. Since the size of this (meta)data on which we apply *ABD* is small, it does not hurt the performance of our algorithm much. On the other hand, this technique allows us to perform much less work on actual data, which we imagine is much larger than the metadata. The following sections describe in more detail what our algorithm accomplishes.

1.2 The Problem and Motivation

In this thesis, we consider the problem of fault-tolerant replication of read/write data with atomic semantics in a message passing network. Some examples where our algorithm can be applied are in implementing a fault-tolerant shared data-structure, or a replicated file system.

Our goal is to design a fault-tolerant, efficient and self-contained replication algorithm. We are especially interested in obtaining fast read response, since in a typical replicated data system, the number of reads is greater than the number of writes. The minimum amount of work any replication algorithm must perform for a read operation is to read one copy of the data. If the algorithm must tolerate f replica faults, the minimum amount of work for a write operation is to write $f + 1$ copies of the data. We would like to get as close as possible to these lower bounds. In order to achieve this, our algorithm might need to perform some more operations on metadata. But if the size of the metadata is small compared to the size of the data, the algorithm is still efficient.

We are also interested in designing a self-contained algorithm which does not rely on external communication, fault-detection, or concurrency control schemes. For example, we are not interested in algorithms which are built on top of group communication services, or which depend on distributed locking protocols. In fact, both group communication and distributed locking are strictly more difficult problems than the atomic replicated data problem we are trying to solve. Our algorithm will be both simpler and more efficient if it is self-contained.

Lastly, we would like to know that our algorithm is memory-efficient. To this end, we are interested in knowing some lower bounds on the memory costs of replication, and comparing these lower bounds to the costs of our algorithm. Though it is difficult for our algorithm to meet such lower bounds exactly, we would like to achieve them up to some constant factors.

1.3 Our Contributions

In this thesis, we introduce the *Layered Data Replication (LDR)* algorithm. *LDR* provides atomic semantics on the replicated data, high fault-tolerance, and efficient read and write operations. To tolerate f faults, we only need to replicate the data at $f + 1$ replicas. We never write the data during a read operation, and read only one copy of the data in each read operation.

Our algorithm exploits the fact that the size of the data being replicated is often much larger than the size of metadata used to keep different copies of the data consistent. In particular, we replicate the data at arbitrary locations, and use a shared data-structure to atomically store the set of locations with the most up-to-date copy of the data. This allows us to resolve inconsistent views of the data by operations on the shared data-structure, instead of the data itself. Since the data-structure is small compared to the data, this approach decreases the communication and latency of the algorithm. We show in our analysis that in the limiting case where the data is much larger than the shared data, our algorithm achieves asymptotically optimal communication and latency. In addition, our algorithm is particularly optimized for fast read response. This improves the overall performance of the replicated system.

We also prove two results containing time and memory lower bounds on the cost of replication. We first prove that for any atomic replication algorithm that tolerates the failure of f replicas, clients must sometimes write to at least f replicas during a read operation. We then prove that for any *selfish* consistent replication algorithm, in which clients don't "help" each other complete their operations, the replicas must use storage which is proportional to the maximum number of concurrently writing clients. In our algorithm, clients write metadata but not data during a read, and the storage used by each replica is linear in the number of concurrent writers. The lower bounds show that these properties are in some sense necessary.

1.4 Organization

Chapter 2 defines I/O automata, which we use to model our algorithm, and atomicity, the correctness condition of our algorithm. It also describes the *ABD* algorithm [2], which we adapt for use in our algorithm. Chapter 3 formally defines the replication problem, and chapter 4 defines our model of computation. Chapter 5 describes our replication algorithm *LDR* (*layered data replication*), and chapter 6 proves its correctness and analyzes its performance. Chapter 8 presents our lower bounds. Chapter 9 gives the conclusions of this thesis.

Chapter 2

Preliminaries

2.1 I/O Automata

The I/O automaton (IOA) model is a formal model for describing distributed algorithms. We provide a brief description of IOA, following Chapter 8 of [8].

An IOA is a simple state machine in which the transitions are associated with atomic, named actions. The actions are classified as either *input*, *output* or *internal*. The inputs and outputs are used for communication with the automaton's environment, while the internal actions are visible only to the automaton itself. The input actions are assumed not to be under the automaton's control—they just arrive from the outside—while the automaton itself specifies what output and internal actions should be performed.

Let A be an IOA. Denote by $in(A)$ (resp., $out(A)$, $int(A)$) the input actions (resp., output actions, internal actions) of A. Let $acts(A) = in(A) \cup out(A) \cup int(A)$ be the *actions* of A. Let $ext(A) = in(A) \cup out(A)$ be the *external actions* of A, and let $local(A) = out(A) \cup int(A)$ be the *internal actions* of A. Formally, A consists of the five following components.

- $sig(A)$, the *signature* of A, where $sig(A) = (in(A), out(A), int(A))$.
- $states(A)$, the *states* of A, which is an arbitrary set.
- $start(A)$, the *start states* of A, which is a nonempty subset of $states(A)$.
- $trans(A)$, the *state transition relation*, or *transitions* of A, where $trans(A) \subseteq states(A) \times acts(A) \times states(A)$.
- $tasks(A)$, the *task partition* of A, which is an equivalence relation on $local(A)$.

2.1.1 Executions and Traces

An *execution fragment* of an IOA A is either a finite sequence, $s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r$, or an infinite sequence, $s_0, \pi_1, s_1, \pi_2, \ldots$, of alternating states and actions of A such that $(s_k, \pi_{k+1}, s_{k+1}) \in trans(A)$ for every $k \geq 0$. An execution fragment beginning with a start state is called an *execution*. The *trace* of some execution α of A, denoted by $trace(\alpha)$, is the subsequence of α consisting of all the external actions. Denote by $execs(A)$ the set of all executions of A. Denote by $finexecs(A)$ the set of all finite executions of A. Denote by $exfrags(A)$ the set of all execution fragments of A. Denote by $traces(A)$ the set of all traces of A.

Let $\alpha_1, \alpha_2 \in execs(A)$, where α_1 is finite, and where the last state of α_1 equals the first state of α_2. We write $\alpha_1 \cdot \alpha_2$ for the execution consisting of α_1 followed by α_2. We sometimes omit the $\cdot$. We write $\alpha_1 \sqsubseteq \alpha_2$ if α_1 is a consecutive subsequence of α_2.

Let $\alpha = s_0\pi_1 s_1 \ldots \pi_n s_n \in execs(A)$. We say π *occurs in* α if $\exists i, 1 \leq i \leq n : \pi = \pi_i$. We write $\alpha(i) = s_0\pi_1 s_1 \ldots \pi_i s_i, 1 \leq i \leq n$, for the length $2i+1$ prefix of α. We let $\alpha(0) = s_0$. We also let $|\alpha| = n$, the number of actions in α. Finally, if $n \geq 1$, that is, α contains at least one action, then we write $\alpha.lact = \pi_n$ for the final action of α, and $\alpha.lstate = s_n$ for the final state of α.

Let $\beta \in traces(A)$, and $P \subseteq ext(A)$. We write $\beta|P$ for the subsequence of β consisting of all actions that belong to P.

2.1.2 Operations on Automata

Composition

The composition operation allows an automaton representing a complex system to be constructed by composing automata representing individual system components. The composition identifies actions with the same name in different component automata. A finite collection of automata can be composed if the set of internal actions of each automaton is disjoint from the sets of actions of all the other automata, and the sets of output actions of all the automata are disjoint. Under this condition, the composition of the collection of automata is, roughly, an automaton whose states is the cross product of the states of the component automata, and whose transition relation is the union of the transition relations of the component automata. See [8] for a more detailed exposition.

Let A be a composition of automata, B be an automaton in the composition, and s be a state of A. We write $s|B$ for the state of B in s.

Hiding

Hiding is an operation which reclassifies output actions of an IOA as internal actions. This prevents them from being used for further communication and means that they are no longer included in traces. We first define the hiding operation for signatures: if S is a signature and $\Phi \subseteq out(S)$, then $hide_\Phi(S)$ is defined to be the new signature S', where $in(S') = in(S)$, $out(S') = out(S) - \Phi$, and $int(S') = int(S) \cup \Phi$. Now, if A is an automaton, $hide_\Phi(A)$ is defined as the same automaton as A, but with signature $hide_\Phi(sig(A))$.

2.2 Quorum Systems

We will define a variant of the standard quorum system, consisting of two collections of sets, where every set from the first collection intersects with every set from the second collection.

Definition 2.2.1 *Let S be a set. $(\mathcal{Q}_1, \mathcal{Q}_2)$ is a* quorum system pair *over S if*

1. $\mathcal{Q}_1, \mathcal{Q}_2 \subseteq 2^S$.

2. $\forall Q_1 \in \mathcal{Q}_1\ \forall Q_2 \in \mathcal{Q}_2 : Q_1 \cap Q_2 \neq \emptyset$.

For a quorum system pair $\mathcal{Q} = (\mathcal{Q}_1, \mathcal{Q}_2)$, we refer to $\mathcal{Q}_1$ as the *read* quorum of $\mathcal{Q}$, and $\mathcal{Q}_2$ as the *write* quorum of $\mathcal{Q}$.

Fix a set S. One example of a quorum system pair $(\mathcal{Q}_1, \mathcal{Q}_2)$ is $\mathcal{Q}_1 = \mathcal{Q}_2 = \{T \mid (T \subseteq S) \wedge (|T| > |S|/2)\}$. That is, $\mathcal{Q}_1$ and $\mathcal{Q}_2$ consist of sets which have more than half the elements of S.

2.3 Variable Type

Following [8], we define a *variable type* as consisting of the following:

- V, a set of values.
- $v_0 \in V$, an initial value.
- A set of *invocations*.
- A set of *responses*.
- A function $f : \mathit{invocations} \times V \rightarrow \mathit{responses} \times V$.

Let T be a variable type. A *trace* of an object with type T is a sequence $v_0 a_1 b_1 v_1 a_2 b_2 v_2 \ldots$, where $\forall i : v_i \in V$, and $\forall i : (b_{i+1}, v_{i+1}) = f(a_{i+1}, v_i)$. Thus, a trace is a sequence of value, invocation, and response triples, where the initial value is v_0, and where later values and responses are calculated by applying f to the preceding invocation and value.

2.4 Atomicity

Again following [8], we define what it means for a trace to satisfy the atomicity property for some variable type.

Let A be an IOA. Call a subset of $in(A)$ the *invocations* of A, and for each invocation, select an action in $out(A)$ to be the *corresponding response* to the invocation. Informally, an *operation* is a pair consisting of the occurrence of an invocation in a trace and the next occurrence of the corresponding response in the trace. Formally, we define two types of operations. A *complete operation* in

$\beta \in traces(A)$ is a pair (ι, ρ), where ι and ρ are events in β, and where ι is an invocation, and ρ is the first occurrence of ι's corresponding response in β after ι. An *incomplete operation* in β is an event ι of β, such that ι is an invocation, and the corresponding response to ι does not occur in β after ι. We say the *interval* of a complete operation (ι, ρ) in trace β is the consecutive subsequence of β starting with ι and ending with ρ. The *interval* of an incomplete operation ι in β is the consecutive subsequence of β starting at ι, and including all actions of β after ι. We extend the definition of complete and incomplete operations in the obvious way when dealing with execution fragments of A. The interval of a complete operation (ι, ρ) in an execution fragment α of A is defined as $s_\iota \iota \ldots \rho s_\rho$, where s_ι is the state immediately preceding ι in α, and s_ρ is the state immediately following ρ in α. We define the interval of an incomplete operation ι in α similarly.

Let T be a variable type, and consider $\beta \in traces(A)$. Let a *linearization* of β be a sequence of actions obtained as follows:

1. For each completed operation ϕ, insert a *linearization point* $*_\phi$ somewhere within ϕ's interval.

2. Select a subset Φ of the incomplete operations.

3. For each operation $\phi \in \Phi$, select a corresponding response.

4. For each operation $\phi \in \Phi$, insert a *linearization point* $*_\phi$ somewhere after ϕ's invocation.

5. For each completed operation ϕ, move the invocation and response actions of ϕ (in that order) to the linearization point ϕ_*. (That is, "shrink" the interval of the operation ϕ to its linearization point.) Also, for each operation $\phi \in \Phi$, put the invocation of ϕ, followed by the selected response, at ϕ_*. Finally, remove all invocations of incomplete operations $\phi \notin \Phi$.

We say β *satisfies the atomicity property* for T if there exists a linearization of β such that the sequence produced by the above procedure is a trace of the underlying variable type T.

2.5 *ABD* Algorithm

Attiya, Bar-Noy and Dolev present an algorithm in [2] for simulating a single-writer/multi-reader shared register in a message passing network. In [9], this algorithm is extended to simulate a multi-writer/multi-reader shared register, and handle dynamic sets of users. Since the main ideas relevant to our work from these algorithms is the way reads are performed, we shall refer to the extended MWMR algorithm as *ABD* in the rest of this thesis.

A MWMR register has the same semantics as atomic replicated data. Therefore, *ABD* can be used as a data replication algorithm. *ABD* performs quorum based replication. However, unlike other quorum based replication algorithms such as weighted voting [5], *ABD* does not need any separate concurrency control mechanism. This reduces the complexity of *ABD* compared to other replication algorithms, and improves its fault tolerance and performance.

We now briefly describe the *ABD* algorithm. Assume that there is a group of replica processes, and a quorum system pair defined over the replicas. Users of *ABD* read and write values of the data being replicated at the replicas. Each write to the data is marked with a tag, consisting of a natural number and the ID of the user who originates the write. Tags are ordered lexicographically. Each replica stores a value of the data, and the tag of the write that wrote the value.

To do a write operation on the data, a user first reads the tags from a read quorum of replicas. Then the user picks a tag higher than any tag it read, and writes its value and that tag to a write quorum of replicas.

To read the data, a user first reads the tags and values from a read quorum of replicas. We call this the *query phase* of the read. Next, the user picks the value marked by the highest tag, and *writes* this value and tag to a write quorum of replicas. When the write finishes, the user returns the value it picked. We call the writes performed by the user the *propagation phase* of the read.

Chapter 3

Problem Statement

In this chapter, we define the notion of a fault-tolerant algorithm for maintaining strongly consistent (*i.e.*, atomic) replicated data.

3.1 Read/Write Variable Type

A replicated data algorithm may be accessed concurrently by multiple users, and the algorithm may keep many copies of the data internally. Yet externally the algorithm should look like a single multi-writer/multi-reader atomic register. We begin by defining the variable type of an atomic register. Let $REG(V, v_0)$ be the variable type of an atomic register with values in V and initial value v_0. The invocations to $REG(V, v_0)$ are *read* and $write(v), v \in V$. The responses are $read\text{-}ok(v), v \in V$ and *write-ok*. The transition function f of $REG(V, v_0)$ is defined by: $f(read, v) = (read\text{-}ok(v), v), v \in V$, and $f(write(w), v) = (write\text{-}ok, w), v, w \in V$.

3.2 Client/Server Read/Write Object

We now describe an object whose external interface is that of a MWMR atomic register. The object is accessed through a set $\mathcal{C}$ of *client* proxies. Clients accept external invocations to read and write the data, and output the appropriate response. Clients coordinate with a finite set $\mathcal{S}$ of *servers* to return consistent values of the data. Servers are internal to the algorithm, and their input actions should not be directly invoked by external users.

More formally, let $\mathcal{C}$, $\mathcal{S}$, and V be sets, where $\mathcal{S}$ is finite. We say an I/O automaton A is a $(\mathcal{C}, \mathcal{S}, V)$*-read/write object* if its external signature is of the following form. For every $i \in \mathcal{C}$, A has input actions $read_i$ and $write(v)_i, v \in V$, and output actions $read\text{-}ok(v)_i, v \in V$ and $write\text{-}ok_i$. We refer to $read_i$ ($write(*)_i$) as an *invocation* at i, and $read\text{-}ok(*)_i$ ($write\text{-}ok_i$) as the *corresponding response* at i. For every $i \in \mathcal{C} \cup \mathcal{S}$, A has an input action $fail_i$. A may have other input and output actions in addition to invocations, responses, and fail actions. Figure 3-1 shows the external signature of A.

Signature

Input
$\mathbf{read}_i, i \in \mathcal{C}$
$\mathbf{write(v)}_i, i \in \mathcal{C}, v \in V$
$\mathbf{fail_i}, i \in \mathcal{C} \cup \mathcal{S}$
(other input actions)

Output
$\mathbf{read\text{-}ok(v)}_i, i \in \mathcal{C}, v \in V$
$\mathbf{write\text{-}ok}_i, i \in \mathcal{C}$
(other output actions)

Figure 3-1: External signature of a $(\mathcal{C}, \mathcal{S}, V)$-read/write object.

Let $\mathcal{C}$, $\mathcal{S}$ and V be some sets, and let A be a $(\mathcal{C}, \mathcal{S}, V)$-read/write object. Define

$$UA(A) = \bigcup_{i \in \mathcal{C}} \left(\{read_i, \textit{write-ok}_i\} \cup \bigcup_{v \in V} \{\textit{read-ok}(v)_i, write(v)_i\} \right)$$

We say $UA(A)$ is the set of *user actions* of A. $UA(A)$ is the subset of the clients' actions by which other objects (users) interact with A.

For the remainder of this section, fix some sets $\mathcal{C}$, $\mathcal{S}$ and V, and fix A to be a $(\mathcal{C}, \mathcal{S}, V)$-read/write object.

A is guaranteed to behave correctly only if users access it in the "right" way. The only conditions we impose on a user of A are that its interface matches the interface of A, and that when a user invokes an action on A, it waits for the action's response to occur before invoking another action.

Formally, we say an automaton U is a *user for* A if the following are true of U. First, the outputs of U are all the invocations of A, and the inputs of U are all the corresponding responses of A. Second, U must *preserve well-formedness* for A, in the sense that $\forall \beta \in traces(A \times U)$, and for all $i \in \mathcal{C}$, U does not make an invocation at i until it has received the corresponding response to any previous invocation U made at i in β.

We say that $\beta \in traces(A \times U)$ is *well-formed* if for all $i \in \mathcal{C}$, β alternates between invocations and responses at i, starting with an invocation.

Lastly, we define the type of failures A can tolerate and still behave correctly. This will be a collection $\mathcal{F}$ of sets, where each set in $\mathcal{F}$ represents a set of $fail_*$ actions that A can tolerate. Formally, we say that $\mathcal{F}$ is a *failure pattern* for A if $\mathcal{F} \subseteq 2^{\mathcal{C} \cup \mathcal{S}}$. For example, if $\mathcal{F} = 2^{\mathcal{C}}$, then A will behave correctly when any set of $fail_i, i \in \mathcal{C}$ occur, but no $fail_i, i \in \mathcal{S}$ occurs.

3.3 Fault-tolerant Replicated Data Algorithm

The previous section defined the interface of a $(\mathcal{C}, \mathcal{S}, V)$-read/write object. We can imagine that the set of traces exhibited by an object is generated by some underlying algorithm. In fact, we can identify the *object* with an *algorithm* generating its traces. In this section, we define the kind of traces a $(\mathcal{C}, \mathcal{S}, V)$-read/write object must exhibit to qualify it as a fault-tolerant replicated data algorithm. For clarity, we divide the definition into two parts. The first part defines consistency properties of the traces, and the second part defines fault-tolerance properties of the traces.

Definition 3.3.1 *Let A be a $(\mathcal{C}, \mathcal{S}, V)$-read/write object for some sets $\mathcal{C}$, $\mathcal{S}$ and V, and let $v_0 \in V$. A is a* strongly consistent replica control algorithm (srca) for (V, v_0) *if, for any user U of A, the following hold:*

- Well-formedness: $\forall \beta \in traces(A \times U)$, *$\beta$ is well-formed.*
- Atomicity: $\forall \beta \in traces(A \times U)$, *$\beta | UA(A)$ satisfies the atomicity property for $REG(V, v_0)$.*

The first part of this definition says that A correctly alternates between receiving an invocation from a user and sending a response. The second part of the definition says that the sequence of user actions in a trace of $A \times U$ is the behavior of an atomic register with range V and initial value v_0.

We now define what it means for a strongly consistent replica control algorithm to be fault tolerant.

Definition 3.3.2 *Let V be a set, and let $v_0 \in V$. Let A be a srca for (V, v_0), and let $\mathcal{F}$ be a failure pattern for A. A is an* $\mathcal{F}$-fault-tolerant srca ($\mathcal{F}$-srca) for (V, v_0) *if for any user U of A, we have*

- Liveness: $\forall \beta \in fairtraces(A \times U)$, *if there exists an $F \in \mathcal{F}$ such that all $fail$ events in β occur at endpoints in F, then every invocation at a non-failing endpoint from $\mathcal{C}$ in β has a response in β.*

This definition says that A is an $\mathcal{F}$-srca for (V, v_0) if it satisfies the well-formedness and atomicity requirements of a srca for (V, v_0), and is also guaranteed to be responsive if faults occur only at a set of endpoints in $F \in \mathcal{F}$.

Sometimes we wish to consider replica control algorithms which are only guaranteed to be correct when composed with certain users. In this case, we make the following definition.

Definition 3.3.3 *Let V be a set, and let $v_0 \in V$. Let A be a srca for (V, v_0), and let $\mathcal{F}$ be a failure pattern for A. Let U be a user for A. We say A is an* $\mathcal{F}$-srca for user *U if A satisfies the well-formedness, atomicity and liveness conditions of Definitions 3.3.1 and 3.3.2, when composed with the user U.*

In the remainder of this thesis, we fix a V and fix a $v_0 \in V$. For technical reasons, assume that $|V| = \infty$. We refer to an $\mathcal{F}$-srca for (V, v_0) as simply an $\mathcal{F}$-srca in the remainder of the thesis.

Chapter 4

Computational Model

In this chapter, we discuss two computational models in which to implement an $\mathcal{F}$-srca. We first discuss an architecture for an $\mathcal{F}$-srca. Then we describe two models for communication in this architecture. The first models an asynchronous message-passing network, while the second models a shared memory system. In Chapter 5, we will describe our algorithm in terms of an asynchronous network, because it more closely resembles the environment we intend to run the algorithm in. In Chapter 8, we will prove some lower bound results in the shared memory model, because it is simpler to work with.

4.1 General Architecture

Fix sets $\mathcal{C}$, $\mathcal{S}$ and V for the remainder of this chapter, and fix A to be a $(\mathcal{C}, \mathcal{S}, V)$-read/write object. We consider an architecture where there is an I/O automaton corresponding to each $i \in \mathcal{C} \cup \mathcal{S}$. Formally, for each, for each $i \in \mathcal{C}$ (resp. $i \in \mathcal{S}$), there is an automaton C_i, called a *client* (resp. S_i, called a *server*). Let $C = \prod_{i \in \mathcal{C}} C_i$, $S = \prod_{i \in \mathcal{S}} S_i$, and let $A = C \times S$.

We now describe two models for communication between components in this architecture.

4.2 Network Model

4.2.1 Model Definition

In the asynchronous network model, components of A communicate through reliable, FIFO channels. We allow communication only between a client and a server, or between two servers (for uniformity, we allow a server to communicate with itself). We do not allow two clients to communicate. The reason for this restriction is that we do not want clients to rely on other clients in order for A to work correctly. That is, we want A to work correctly even when there is only a single client running.

Formally, we say an automaton N is a *reliable network* for A if N can be described in the following way. Let $\mathcal{N} = (\mathcal{C} \times \mathcal{S}) \cup (\mathcal{S} \times \mathcal{I})$, and let $(i, j) \in \mathcal{N}$. The *channel*

$N_{i,j}, (i,j) \in \mathcal{N}$
Signature

Input	Output
$\mathbf{send}(m)_{i,j}, m \in \mathcal{M}$	$\mathbf{recv}(m)_{i,j}, m \in \mathcal{M}$

Figure 4-1: $N_{i,j}$ signature.

between i and j is $N_{i,j}$. $N = \prod_{(i,j)\in\mathcal{N}} N_{i,j}$.

Let $\mathcal{M}$ be an arbitrary message alphabet. The messages that can be sent through a channel come from $\mathcal{M}$. Figure 4-1 gives the signature of channel $N_{i,j}$. i sends message $m \in \mathcal{M}$ to j by invoking $send(m)_{i,j}$. j receives m from i when $recv(m)_{i,j}$ occurs.

The network is asynchronous, so we assume no bound on the delay of a channel. However, we require that all channels make the standard guarantees of message integrity, FIFO ordering, no duplication, and eventual (reliable) delivery. See [8], Chapter 14, for formal definitions of these properties. One example of a network with these properties is a network running TCP.

4.2.2 $\mathcal{F}$-srca in the Network Model

Definition 4.2.1 *Let N be a reliable network for A. A is an* $\mathcal{F}$-srca in the network model *if*

1. $A \times N$ *is an* $\mathcal{F}$*-srca.*
2. $\forall (i,j) \in \mathcal{C} \times \mathcal{S}\ \exists \mathcal{M}_{i,j} \subseteq \mathcal{M}\ \forall m \in \mathcal{M}_{i,j} : send(m)_{i,j} \in out(C_i) \wedge recv(m)_{j,i} \in in(S_i)$.
3. $\forall (i,j) \in \mathcal{S} \times \mathcal{I}\ \exists \mathcal{M}'_{i,j} \subseteq \mathcal{M}\ \forall m \in \mathcal{M}'_{i,j} : send(m)_{i,j} \in out(S_i) \wedge (recv(m)_{j,i} \in in(C_i) \vee recv(m)_{j,i} \in in(S_i))$.
4. $out(C) \cap in(S) = out(S) \cap in(C) = \emptyset$.

Thus, A is an $\mathcal{F}$-srca in the network model if $A \times N$ is an $\mathcal{F}$-srca, the only means of communication between components is using the network N, and the messages they send to each other come from $\mathcal{M}$.

4.2.3 Cost Measures

The cost of an $\mathcal{F}$-srca is measured in terms of its time and communication complexity. For an $\mathcal{F}$-srca in the networks model, the time complexity of the algorithm is the time between an invocation by a user and the response, for either a user read or write action. We will assume an upper bound on the delay of the message channels, and we will assume local processing by clients and servers takes no time. We will also assume that the time to transfer a message is proportional to the size of the message.

The message complexity of an operation is the total amount of messages sent in the operation, defined as the sum of the sizes of the messages sent. These assumptions are more carefully defined in Chapter 7.1, when we analyze the cost of a particular $\mathcal{F}$-srca in the networks model.

4.3 Atomic Servers Model

In this model, each server automaton is an atomic object. Clients communicate with the servers by invoking actions and receiving responses from the servers. We still assume that clients don't communicate with each other. In addition, we now assume that servers don't communicate with each other. The atomic servers model is similar to the shared memory model, where the servers play the role of the shared memory. However, because the servers are atomic objects, they can be accessed by concurrent clients.

To formally define the atomic servers model, let $\mathcal{M}'$ and W be sets, and let $w_0 \in W$. We first define a read/modify variable type $RM(W, w_0, \mathcal{M}')$. The domain of $RM(W, w_0, \mathcal{M}')$ is W, and the initial value is w_0. The invocations to $RM(W, w_0, \mathcal{M}')$ are $read$ and $modify(m), m \in \mathcal{M}'$. The responses are $\textit{read-ok}(v), v \in W$, and $\textit{modify-ok}$. The distinction between the read and modify operations is that a read cannot change the value of the object, while a modify can change the value arbitrarily. Formally, let $g : W \times \mathcal{M}' \rightarrow W$. Then the transition function f of $RM(W, w_0, \mathcal{M}')$ is defined by $f(read, v) = (\textit{read-ok}(v), v)$, and $f(modify(m), v) = (\textit{modify-ok}, g(m, v))$.

A read/modify variable is similar to a register. The read operation of the two variables work the same way. A $modify(m)$ operation on a read/modify variable differs from a $write(v)$ operation on a register in that $modify(m)$ sets the value of the RM variable to $g(m, v)$, where v is the previous value of the RM variable, whereas $write(v)$ simply sets the value of the register to v. Thus, the RM variable is a generalization of a register.

For each $j \in \mathcal{S}$, fix sets $\mathcal{M}'_j$ and W_j, and let $(w_0)_j \in W_j$. S_j has input actions $read_j$ and $modify(m)_j, m \in \mathcal{M}'_j$, and output actions $\textit{read-ok}(v)_j, v \in W_j$ and $\textit{modify-ok}_j$. We call $read_j$ $(write(*)_j)$ an *invocation* at j, and $\textit{read-ok}(*)_j$ $(\textit{modify-ok}_j)$ the *corresponding response* at j. Define

$$SA(A)_j = \bigcup_{i \in \mathcal{C}} \left(\{read_{i,j}, \textit{modify-ok}_{i,j}\} \cup \bigcup_{v \in W_j} \{\textit{read-ok}(v)_{i,j}\} \cup \bigcup_{m \in \mathcal{M}'_j} \{modify(m)_{i.j}\} \right)$$

$SA(A)_j$ is the set of actions the clients use to interact with the server j.

Definition 4.3.1 *A is an* $\mathcal{F}$-srca in the atomic servers model *if*

1. *A is an $\mathcal{F}$-srca.*

2. *For all $i \in \mathcal{C}$ and $j \in \mathcal{S}$, we have*

   (a) $out(C_i) \cap in(S_j) = \{read_{i,j}\} \cup \bigcup_{m \in \mathcal{M}'_j} \{modify(m)_{i,j}\}$.

(b) $in(C_i) \cap out(S_j) = \{modify\text{-}ok_{i,j}\} \cup \bigcup_{v \in W_j} \{read\text{-}ok(v)\}$.

3. $\forall \beta \in traces(A) \forall i \in \mathcal{S}, \beta | SA(A)_i$ *satisfies the atomicity property for* $RM(W_i, w_0, \mathcal{M}'_i)$.

Thus, A is a $\mathcal{F}$-srca in the atomic servers model if A if a $\mathcal{F}$-srca, the only communication between the clients and servers is invoking read and modify operations, and the servers behave like read/modify atomic objects.

One example of an $\mathcal{F}$-srca in the atomic servers model is an $\mathcal{F}$-srca in which the clients are programs, and the servers are hard disks. Overlapping operations at each hard disk may execute in some arbitrary order, but the operations at each disk are linearizable, and so each disk behaves like a read/modify atomic object.

4.3.1 Cost Measures

We defined the atomic servers model in order to prove some lower bounds using the model. The lower bound in Chapter 8.1 shows that an $\mathcal{F}$-srca in the atomic servers model must perform some minimum number of $modify$ actions. However, it does not say anything about the parameters those $modify$ actions take, nor the size of the parameters. Thus, the cost measure we consider in Chapter 8.1 is only the *number* of $modify$ actions an $\mathcal{F}$-srca takes in a read or write operation.

The lower bound in Chapter 8.2 considers the minimum amount of storage a server in an $\mathcal{F}$-srca must have. Instead of defining a cost measure by concretely specifying the type of data stored at a server, Chapter 8.2 defines a more abstract notion of storage based on the *fault tolerance* of the data, and the number of different values of the data that can be read at a certain point in an execution. The definitions are slightly involved, and we refer the reader to Chapter 8.2 for an in depth discussion.

Note that both memory cost measures which we have defined for the atomic servers model are abstract, in the sense that they don't refer to the size of specific data structures. It is possible to define more concrete cost measures. For example, we can define the communication cost of an operation as the sum of the sizes of the values read and written during the operation using $read$ and $modify$ actions. However, it is more difficult to reason about such measures in lower bound proofs, which is why we only consider the abstract measures.

4.4 Relationship Between the Models

The motivation for considering the two lower bounds in Chapter 8 was to show that some of the constructions used in the algorithm we describe in Chapter 5 are necessary. Since we prove the lower bounds in the atomic servers model, while our algorithm works in the network model, we should show some relationship between the two models for the lower bounds to carry through to the network model. However, we don't consider a formal transformation between algorithms for the atomic servers model into algorithms for the network model. Instead, we note that the ideas in the lower bound proofs carry over from the atomic servers model to the network model.

For example, looking ahead to Chapter 8.1, instead of proving that at least f $modify$ actions must occur during some read operation of an $\mathcal{F}$-srca in the atomic servers model, we can use a very similar line of reasoning to show that at least f servers must change their state during some read operation of an $\mathcal{F}$-srca in the network model, which in turn implies that a reading client must sometimes send out at least f write messages during its read. The interested reader can also adapt the statement and proof of the second lower bound for the network model.

Chapter 5

LDR Algorithm

5.1 Overview

In this chapter, we describe the *Layered Data Replication (LDR)* algorithm, an efficient $\mathcal{F}$-srca in the network model. The idea of *LDR* is to replicate the data at arbitrary locations, then use directories to find the up-to-date replicas. The problem is coordinating the information at the directories with the actual set of up-to-date replicas, and performing the coordination efficiently. *LDR* uses ideas from the *ABD* algorithm described in section 2.5, but has more efficient read operations, and equally efficient writes. Chapter 7 analyzes the costs of *LDR*, and also compares them to the costs of *ABD*.

LDR replicates one piece of data, which we will call x for the remainder of this thesis. To replicate multiple data items, we can run a separate instance of *LDR* for each item. *LDR* uses the network model described in Section 4.2, and is composed of a set of client automata and a set of server automata communicating over a reliable, asynchronous network. *LDR* refines the network model by dividing the set of server automata into non-empty sets of *directory* and *replica* automata. The replicas are used to store values of x. The directories store which replicas have the newest value of x. That is, each directory stores a set of replica names. The main steps for a read operation are for a client to read some directories to find the set of up-to-date replicas, then write this set to some directories, and then return the value read from one of those replicas. To write a value, a client first gets a tag, then writes the new value at a sufficiently large set of replicas, and then writes at some directories which set of replicas it just updated.

In *LDR*, a write to x can be done at any set of replicas, and a read can be done at any up-to-date replica. But reads and writes about the set of up-to-date replicas can be done only at read and write quorums of directories, from a quorum system pair defined over the directories. This is so that the client reads and writes (of the set of up-to-date replicas) to the directories will be atomic, which in turn allows client reads and writes to x to be atomic. The precise way in which this occurs is explained in Chapter 6, when we prove the correctness of *LDR*.

Below, we first describe the architecture and definitions used in *LDR*. Then we

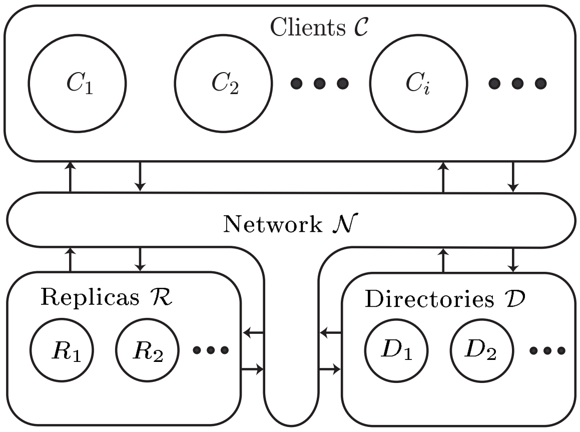

Figure 5-1: *LDR* architecture.

describe the algorithms for the clients, replicas and directories.

5.2 Architecture

LDR is an $\mathcal{F}$-srca in the network model. For the rest of this thesis, we fix a set $\mathcal{C}$ to be the set of clients in *LDR*, and fix a finite set $\mathcal{S}$ to be the set of servers. We also fix $\mathcal{R}$ and $\mathcal{D}$ to be nonempty subsets of $\mathcal{S}$ such that $\mathcal{S} = \mathcal{R} \cup \mathcal{D}$. For each $i \in \mathcal{R}$ (resp., $i \in \mathcal{D}$), there is an automaton R_i, called a *replica* (resp., D_i, called a *directory*). Let $R = \prod_{i\in\mathcal{R}} R_i$, $D = \prod_{i\in\mathcal{D}} D_i$, and $S = R \times D$. The *LDR* architecture is shown in Figure 5.2. Figure 5-2 (resp., 5-3, 5-4) gives the signature of client C_i (resp., replica R_i, directory D_i).

Now, we specify the failure pattern $\mathcal{F}$ that *LDR* tolerates. For the rest of this thesis, we fix a quorum system pair $(\mathcal{Q}_R, \mathcal{Q}_W)$ over $\mathcal{D}$. We also fix a natural number f, such that $2f + 1 \leq |\mathcal{R}|$. The fault-tolerance properties of *LDR* are stated with respect to the quorum system pair and f. Let $\mathcal{F}$ be the collection of all sets F such that:

1. There are at most f different $i \in \mathcal{R}$ such that $fail_i \in F$.

2. There exist sets $Q_1 \in \mathcal{Q}_R, Q_2 \in \mathcal{Q}_W$, such that $\forall i \in Q_1 \cup Q_2 : fail_i \notin F$.

$\mathcal{F}$ consists of all sets of failures in which at most f replicas fail, some read and write quorum of directories never fail, and any set of clients may fail. *LDR* is an $\mathcal{F}$-srca for this failure pattern.

5.3 Definitions

We use tags to order the writes performed by clients. Let $T = \mathbb{N} \times \mathcal{C}$, where $\mathbb{N}$ is the set of natural numbers. We assume that there is a total order on $\mathcal{C}$, and order

T lexicographically. Let t_0 be an arbitrary value which we define to be less than all $t \in T$.

Next, we describe the messages sent by components of *LDR*. Recall that we fixed V as the set of values of x. Define

$$\mathcal{M}_{CD} = \bigcup_{i \in \mathbb{N}, S \subseteq \mathcal{R}, t \in T} \{\langle rread, i\rangle, \langle rwrite, S, t, i\rangle, \langle wread, i\rangle, \langle wwrite, S, t, i\rangle\}$$

$$\mathcal{M}_{DC} = \bigcup_{i \in \mathbb{N}, S \subseteq \mathcal{R}, t \in T} \{\langle rread\text{-}ok, S, t, i\rangle, \langle rwrite\text{-}ok, i\rangle, \langle wread\text{-}ok, t, i\rangle, \langle wwrite\text{-}ok, i\rangle\}$$

$$\mathcal{M}_{CR} = \bigcup_{v \in V, t \in T, i \in \mathbb{N}} \{\langle read, t, i\rangle, \langle write, v, t, i\rangle \langle secure, t, i\rangle\}$$

$$\mathcal{M}_{RC} = \bigcup_{v \in V, t \in T, i \in \mathbb{N}} \{\langle read\text{-}ok, v, t, i\rangle, \langle write\text{-}ok, i\rangle\}$$

$$\mathcal{M}_{RD} = \bigcup_{r \in \mathcal{R}, t \in T} \langle write, r, t\rangle$$

$$\mathcal{M}_{RR} = \bigcup_{v \in V, t \in T} \langle gossip, v, t\rangle$$

$$\mathcal{M}_{CC} = \mathcal{M}_{DD} = \mathcal{M}_{DR} = \emptyset$$

These represent the set of messages that one group of automata sends to another. For example, $\mathcal{M}_{CD}$ is the set of messages that clients send to directories, and $\mathcal{M}_{DC}$ is the set of messages that directories send to clients. Note that clients don't send messages to other clients, and directories don't send messages to other directories, nor to replicas.

To explain the nomenclature of the messages, the *read*(*-ok*) and *write*(*-ok*) messages do what their names imply. The *secure* and *gossip* messages are explained in Section 5.5. The *rread*(*-ok*) and *rwrite*(*-ok*) are used by clients and directories during reads by clients, while *wread*(*-ok*) and *wwrite*(*-ok*) are used during client writes.

Lastly, we define a *latest value* of x after a finite execution fragment α of *LDR*.

Definition 5.3.1 *Let α be a finite execution fragment of* LDR. *A* latest value of x after α *is either the value of the last completed write in α, or the value of an incomplete write in α. If there are no completed writes in α, then it is the value of any incomplete write, or v_0. If there are no completed or uncompleted writes in α, then it is v_0.*

Note that in general, there may be several latest value of x after α, if there are several incomplete (*i.e.*, ongoing) writes in α.

$C_i, i \in \mathcal{C}$

Signature

Input
- $\mathbf{read}_i$
- $\mathbf{write}(v)_i, v \in V$
- $\mathbf{recv}(m)_{j,i}, (m \in \mathcal{M}_{DC} \wedge j \in \mathcal{D}) \vee (m \in \mathcal{M}_{RC} \wedge j \in \mathcal{R})$
- $\mathbf{fail}_i$

Output
- $\mathbf{read\text{-}ok}(v)_i, v \in V$
- $\mathbf{write\text{-}ok}_i$
- $\mathbf{send}(m)_{i,j}, (m \in \mathcal{M}_{CD} \wedge j \in \mathcal{D}) \vee (m \in \mathcal{M}_{CR} \wedge j \in \mathcal{R})$

State

$acc \subseteq \mathcal{I}$, initially $\emptyset$
$phase \in \{idle, rdr, rdw, rrr, rok, wdr, wdw, wrs, wok\}$, initially $idle$
$tag \in T \cup \{t_0\}$, initially t_0
$val \in V$, initially v_0
$msg[j] \in \mathcal{M}_{CD}, \forall j \in \mathcal{D}$, initially all $\perp$
$msg[j] \in \mathcal{M}_{CR}, \forall j \in \mathcal{R}$, initially all $\perp$
$mid \in \mathbb{N}$, initially 0

Figure 5-2: C_i signature and state.

$R_i, i \in \mathcal{R}$

Signature

Input
- $\mathbf{recv}(m)_{j,i}, (m \in \mathcal{M}_{CR} \wedge j \in \mathcal{C}) \vee (m \in \mathcal{M}_{RR} \wedge j \in \mathcal{R})$
- $\mathbf{fail}_i$

Output
- $\mathbf{send}(m)_{i,j}, (m \in \mathcal{M}_{RC} \wedge j \in \mathcal{C}) \vee (m \in \mathcal{M}_{RD} \wedge j \in \mathcal{D}) \vee (m \in \mathcal{M}_{RR} \wedge j \in \mathcal{R})$

Internal
- $\mathbf{gossip}_i$
- $\mathbf{gc}_i$

State

$data \subseteq V \times (T \cup \{t_0\}) \times \{0, 1\}$, initially $\{(v_0, t_0, 1)\}$
$msg[j] \in \mathcal{M}_{RC}, \forall j \in \mathcal{C}$, initially all $\perp$
$msg[j] \in \mathcal{M}_{RD}, \forall j \in \mathcal{D}$, initially all $\perp$
$msg[j] \in \mathcal{M}_{RR}, \forall j \in \mathcal{R}$, initially all $\perp$

Figure 5-3: R_i signature and state.

$D_i, i \in \mathcal{D}$

Signature

Input
- $\mathbf{recv}(m)_{j,i}, (m \in \mathcal{M}_{CD} \wedge j \in \mathcal{C}) \vee (m \in \mathcal{M}_{RD} \wedge j \in \mathcal{R})$
- $\mathbf{fail}_i$

Output
- $\mathbf{send}(m)_{i,j}, m \in \mathcal{M}_{DC} \wedge j \in \mathcal{C}$

State

$utd \subseteq \mathcal{R}$, initially $\mathcal{R}$
$tag \in T \cup \{t_0\}$, initially t_0
$msg[j] \in \mathcal{M}_{DC}, \forall j \in \mathcal{C}$, initially all $\perp$

Figure 5-4: D_i signature and state.

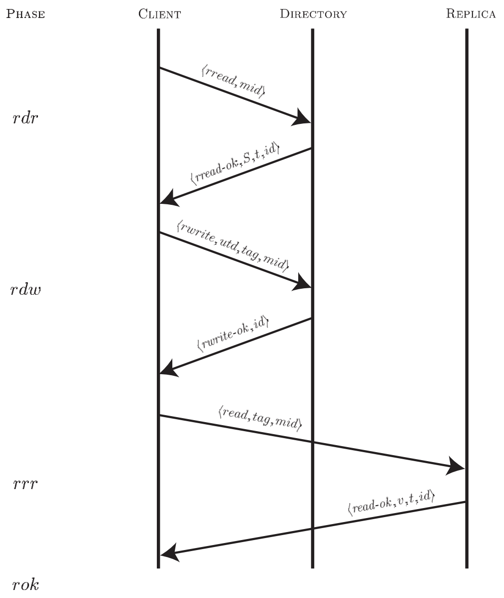

Figure 5-5: Client read operation.

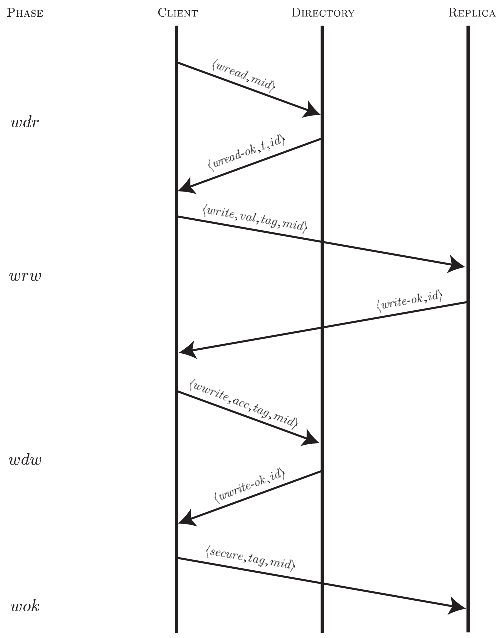

Figure 5-6: Client write operation.

5.4 Client Algorithm

The clients receive user invocations to read and write to x. The transitions of a client C_i are given in Figure 5-7. In this code, C_i marks each message it sends with an integer mid in the last coordinate of the message. This number is echoed by the recipient in its reply, in the last coordinate id of the reply. This lets C_i determine the recency of a response it receives. The id in a response that C_i receives is always less than or equal to i's current mid. If $id < mid$ for a response, then that response is out of date, and C_i ignores it.

5.4.1 Reads

A read by C_i follows four phases: rdr, rdw, rrr and rok. Each of the first three phases corresponds to a round of communication. During the last phase, the client returns a value to the user, but does no communication. That is, there is no cost associated with the last phase.

The first phase, rdr, stands for *read-directories-read*; rdw stands for *read-directories-write*; rrr stands for *read-replicas-read*; rok stands for *read-ok*. We now describe what happens in each phase. The sequence of interactions between clients, replicas and directories is shown in Figure 5-5.

When C_i first receives a $read_i$ invocation, it sends a message $\langle rread, mid\rangle$ to every directory, then enters phase rdr. Here, C_i is trying to find a set of replicas with a latest value of x. During phase rdr, directories acknowledge C_i's read request with messages of the form $\langle \textit{rread-ok}, S, t, mid\rangle$, where S is a set of replicas, and t is a tag. i waits for a read quorum of directories to acknowledge the read, then chooses the (S, t) pair with the largest t among the acks. The S that C_i chooses will be a set of replicas with a latest value of x, where latest value is defined as in Def. 5.3.1; the corresponding t will be the tag for that value of x. C_i sets (utd, tag) equal to (S, t), sends every directory a message $\langle rwrite, utd, tag, mid\rangle$ to write (utd, tag), and enters phase rdw. utd stands for *up-to-date*, and represents the set of replicas that C_i will read the value of x from. The reason that C_i writes (utd, tag) to the directories is to ensure that reads starting after C_i's current read will read a value with a tag at least as high as t. During phase rdw, C_i waits for a write quorum of directories to acknowledge its write with messages of the form $\langle rwrite\text{-}ok, mid\rangle$. After this occurs, C_i tries to read the value of x by sending a read message $\langle read, tag, mid\rangle$ to the replicas in utd, and enters phase rrr. Note that in C_i's message $\langle read, tag, mid\rangle$, C_i tells the replicas that it wants to read a value of x with tag tag. This is because, in general, each replica will store multiple values of x with different tags. The reason a replica does this is explained in Section 5.5. When one of the replicas acknowledges C_i's read with a message $\langle read\text{-}ok, v, t, id\rangle$, C_i takes the value v returned by the replica, and responds to the user with $read\text{-}ok(v)_i$.[1]

It is possible to combine phases rdw and rrr. That is, the client can write to the directories and read from a replica in parallel. The combined phase finishes when the client receives acknowledgments from a write quorum of directories, and receives a value from a replica. Then, the replica enters phase rok. Combining the rdw and rrr phases can decrease the latency of a read. However, for clarity of exposition, we have decided to separate the phases. Chapter 6 proves the correctness of *LDR* assuming the phases are separate. It is easy to adapt the proof to account for combining the phases.

5.4.2 Writes

A write by C_i also follows four phases: wdr, wrw, wdw and wok. There is a round of communication corresponding to each of the first three phases. The final phase, wok, corresponds to a one-way communication from the clients to the replicas. Thus, there are three and a half rounds of communication for a write. At the end of this section, we will describe a way to reduce the communication during the write operation to three rounds, leaving some communication to be performed lazily after the write completes. While this optimization doesn't affect the communication complexity of the write, it does reduce its latency.

The first phase, wdr, stands for *write-directories-read*; wrw stands for *write-replicas-write*; wdw stands for *write-directories-write*; wok stands for *write-ok*. Figure 5-6 shows the interaction between clients, replicas and directories in these phases.

When C_i first receives a $write(v)_i$ invocation to write value v to x, it sends a message $\langle wread, mid\rangle$ to every directory, then enters phase wdr. Here, C_i is trying to find a tag large enough to mark its write as being the latest. During phase wdr,

[1] Note that the replica's reply $\langle read\text{-}ok, v, t, id\rangle$ includes a tag t which C_i discards. The tag t is included in the message to simplify *LDR*'s correctness proof. If actually implementing *LDR*, the tag does not need to be included.

directories acknowledge C_i's read with messages of the form $\langle \textit{wread-ok}, t, id \rangle$, where t is a tag. C_i waits for a read quorum of directory acknowledgments, then chooses the highest tag $t = (n, i')$ from among them. Here, n is a natural number, and $i' \in \mathcal{C}$ is a client ID. Tag t is the tag of a latest value of x, so C_i chooses a larger tag for its write, by setting $tag = (n+1, i)$. Then C_i sends messages of the form $\langle wwrite, val, tag, mid \rangle$ to all the replicas to write (val, tag), where $val = v$, and enters phase wrw. C_i waits for at least $f+1$ replicas to acknowledge the write, to ensure that the value it writes survives even if f replicas fail. When C_i receives a set acc of at least $f+1$ replica acknowledgments, C_i sends a message to all the directories to write (acc, tag), using message $\langle wwrite, acc, tag, mid \rangle$, and enters phase wdw. Here, C_i is informing the directories that the replicas in acc have the most up-to-date value of x. When a write quorum of directories acknowledge i's write with message $\langle \textit{wwrite-ok}, id \rangle$, i sends a message $\langle secure, tag, mid \rangle$ to all the replicas in acc, and responds to the user with $\textit{write-ok}_i$. The secure message tells the replicas that a write quorum of directories know about a write with a tag at least as large as tag, and so the replicas will never in the future need to return a value of x with tag less than tag. This allows the replicas to garbage-collect all values with tag less than tag.

It is possible to return from the write as soon as the client receives acknowledgments from a write quorum of directories in phase wdw, and before the client sends secure messages to the replicas. The client still sends the secure messages to replicas after returning. The proof of correctness in Chapter 6 can be easily adapted to accommodate this change. But again, for clarity of exposition, we do not include this optimization in the pseudocode in Figure 5-7.

5.4.3 Other Actions

The only other input action C_i can receive is $fail_i$. If C_i receives $fail_i$, then it stops taking any more locally-controlled steps.

5.5 Replica Algorithm

5.5.1 State

The replicas store values of x, to which clients read and write. However, instead of storing one value of x, each replica stores a set of values of x. Each value of x has an associated tag, indicating the recency of the value, and an associated security bit, indicating whether the write of the value succeeded (*i.e.*, whether that write received a write quorum of directory acknowledgments during phase wdw). Thus, each replicas stores a set of value-tag-bit triples $\subseteq V \times (T \cup \{t_0\}) \times \{0, 1\}$, called *data*. If $(*, t, 1) \in data$, we say t is *secured*. If $(*, t, 0) \in data$, we say t is *unsecured*. Securing the data and garbage-collection are discussed further in Section 5.5.2.

input $\mathbf{read}_i$
Effect:
  $mid \leftarrow mid + 1$
  for all $j \in \mathcal{D}$ do
    $msg[j] \leftarrow \langle rread, mid\rangle$
  $phase \leftarrow rdr$

input $\mathbf{write(v)}_i$
Effect:
  $val \leftarrow v$
  $mid \leftarrow mid + 1$
  for all $j \in \mathcal{D}$ do
    $msg[j] \leftarrow \langle wread, mid\rangle$
  $phase \leftarrow wdr$

input $\mathbf{fail}_i$
Effect:
  stop taking locally-controlled steps

output $\mathbf{read\text{-}ok(v)}_i$
Precondition:
  $(val = v) \wedge (phase = rok)$
Effect:
  $phase \leftarrow idle$

output $\mathbf{write\text{-}ok}_i$
Precondition:
  $phase = wok$
Effect:
  $phase \leftarrow idle$

output $\mathbf{send(m)}_{i,j}$
Precondition:
  $msg[j] = m$
Effect:
  $msg[j] \leftarrow \perp$

input $\mathbf{recv(m)}_{j,i}$ where $(m = \langle rread\text{-}ok, S, t, id\rangle)$
Effect:
  if $(phase = rdr) \wedge (id = mid)$ then
    $acc \leftarrow acc \cup \{j\}$
    if $(t > tag)$ then
      $tag \leftarrow t$
      $utd \leftarrow S$
    if $(\exists Q \in \mathcal{Q}_R : Q \subseteq acc)$ then
      $mid \leftarrow mid + 1$
      for all $j \in \mathcal{D}$ do
        $msg[j] \leftarrow \langle rwrite, utd, tag, mid\rangle$
      $acc \leftarrow \emptyset$
      $phase \leftarrow rdw$

input $\mathbf{recv(m)}_{j,i}$ where $(m = \langle rwrite\text{-}ok, id\rangle)$
Effect:
  if $(phase = rdw) \wedge (id = mid)$ then
    $acc \leftarrow acc \cup \{j\}$
    if $(\exists Q \in \mathcal{Q}_W : Q \subseteq acc)$ then
      $mid \leftarrow mid + 1$
      for all $j \in utd$ do
        $msg[j] \leftarrow \langle read, tag, mid\rangle$
      $acc \leftarrow \emptyset$
      $phase \leftarrow rrr$

input $\mathbf{recv(m)}_{j,i}$ where $(m = \langle read\text{-}ok, v, t, id\rangle)$
Effect:
  if $(phase = rrr) \wedge (id = mid)$ then
    $val \leftarrow v$
    $tag \leftarrow t$
    $phase \leftarrow rok$

input $\mathbf{recv(m)}_{j,i}$ where $(m = \langle wread\text{-}ok, t, id\rangle)$
Effect:
  if $(phase = wdr) \wedge (id = mid)$ then
    $acc \leftarrow acc \cup \{j\}$
    if $(t > tag)$ then
      $tag \leftarrow t$ $//tag = (n, i')$
    if $(\exists Q \in \mathcal{Q}_R : Q \subseteq acc)$ then
      $mid \leftarrow mid + 1$
      $tag \leftarrow (n + 1, i)$
      for all $j \in \mathcal{R}$ do
        $msg[j] \leftarrow \langle write, val, tag, mid\rangle$
      $acc \leftarrow \emptyset$
      $phase \leftarrow wrw$

input $\mathbf{recv(m)}_{j,i}$ where $(m = \langle write\text{-}ok, id\rangle)$
Effect:
  if $(phase = wrw) \wedge (id = mid)$ then
    $acc \leftarrow acc \cup \{j\}$
    if $(|acc| > f)$ then
      $mid \leftarrow mid + 1$
      for all $j \in \mathcal{D}$ do
        $msg[j] \leftarrow \langle wwrite, acc, tag, mid\rangle$
      $acc \leftarrow \emptyset$
      $phase \leftarrow wdw$

input $\mathbf{recv(m)}_{j,i}$ where $(m = \langle wwrite\text{-}ok, id\rangle)$
Effect:
  if $(phase = wdw) \wedge (id = mid)$ then
    $acc \leftarrow acc \cup \{j\}$
    if $(\exists Q \in \mathcal{Q}_W : Q \subseteq acc)$ then
      $mid \leftarrow mid + 1$
      for all $j \in \mathcal{R}$ do
        $msg[j] \leftarrow \langle secure, tag, mid\rangle$
      $acc \leftarrow \emptyset$
      $phase \leftarrow wok$

Figure 5-7: C_i transitions.

Reason for Storing a Set of Values of x

We now give some intuition why a replica stores a set of values of x instead of a single value. Suppose we want an $\mathcal{F}$-srca that tolerates f replica faults, and suppose each replica only stored one value of x (and possibly other metadata). To make reads fast, the client does not write any values of x during a read operation. Then, suppose that the last complete write to x in some execution of the algorithm wrote the value v to $f+1$ replicas. Consider a client that writes a new value v'. When a replica currently storing v receives a request by the client to write v', it must overwrite v with v'. Otherwise, the write for v' will fail, which would violate the liveness requirement of the algorithm. Now, suppose this overwriting occurs at $f-1$ of the replicas storing v, and then the client writing v' fails. Then, each of v and v' is stored at fewer than f replicas. Since the algorithm tolerates f replica faults, a client who reads must be able to return a response even if it does not hear from up to f replicas. So, by delaying the messages from replicas storing either v or v', we can control whether a reading client returns v or v'. In particular, we can force three sequential reads to return v, v', and v, in that order. But this violates the atomicity of the algorithm. Therefore, the replicas cannot just store one value of x.

In Section 8.2, we formalize and extend the above argument, and prove a theorem that if clients do not write values of x during a read, and we allow an infinite number of concurrent client writes, then the amount of storage at each replica must be unbounded.[2] Since clients do not write values of x during a read in *LDR*, and we place no bound on the number of concurrent writers, this theorem shows that replicas in *LDR* need to have unbounded storage, and justifies why each replica stores a set of values of x.

5.5.2 Transitions

The transitions of replica R_i are given in Figure 5-8. In the transitions, we use a function maxst$(data)$ (maxst stands for *max-secured-tag*), which returns the secured $(value, tag)$ pair with the largest tag in $data$. More precisely,

$$\text{maxst}(data) = \begin{cases} (v,t) & ((v,t,1) \in data) \wedge ((v',t',1) \in data \Rightarrow t \geq t') \\ (v_0,t_0) & \nexists v,t : (v,t,1) \in data \end{cases}$$

We will use the usual subscript notation to extract coordinates from vectors. That is, if $\text{maxst}(data) = (v,t)$, then $\text{maxst}(data)_1 = v$ and $\text{maxst}(data)_2 = t$.

Replica R_i can read, write, gossip and garbage-collect values. We describe these actions below.

[2] In fact, we show a more fine-grained bound, which roughly says that for any $\mathcal{F}$-srca in which clients don't write values of x during reads, the number of concurrent writes allowed cannot exceed the total storage capacity of all the replicas.

Reads

When R_i receives a $\langle read, t, mid\rangle$ message, it is being asked to return a value of x with tag t. If either $(v, t, 0)$ or $(v, t, 1)$ exists in *data*, then R_i returns v and t.[3] Otherwise, R_i must have garbage-collected v (it can be shown that R_i must have stored v in *data* at some point in the past). In this case, R_i returns the largest *secured* value and tag in *data*, *i.e.*, $\text{maxst}(data)$. We will show in Chapter 6 that even though R_i may not return the value corresponding to the tag the client is looking for, the value R_i returns can always be linearized within the execution, and does not violate atomicity. There is also the question why R_i does not simply return the value with the largest, possibly unsecured tag in *data*. Roughly, the reason why R_i returns the value for the largest secured tag is that it must be sure the value has been written to at least $f+1$ replicas, and also that a write quorum of directories know this fact. Otherwise, when a later read reads the directories, it might choose a tag smaller than the tag chosen by the current read, and return a value earlier than the one the current read returned. The security of the tag indicates both that $f+1$ replicas know the value for the tag, and a write quorum of directories know this fact. This argument is formalized in Chapter 6.

Writes

Writing is simple. When R_i receives a $\langle write, v, t, mid\rangle$ message, it appends $(v, t, 0)$ to *data* and returns an acknowledgment to the client requesting the write. Note that v is stored as an unsecured value.

Gossip

Gossiping spreads secured values of x to additional replicas. This way, there are more replicas for clients to read from, which increases the fault tolerance of the data, and makes reads faster by allowing clients to read from a closer replica, to which it may have a faster network connection. To gossip, R_i chooses a secured (v, t) (if any exists), and sends a $\langle gossip, v, t\rangle$ message to the other replicas.

If R_i receives a gossip message $\langle gossip, v, t\rangle$, it adds $(v, t, 1)$ to its *data*. Then it sends a write message $\langle wwrite, \{i\}, t\rangle$ to the directories, to tell the directories that it has become a replica for (v, t). This message may be ignored by the directories if (v, t) is actually out of date, *i.e.*, a value with tag greater than t has been written.

Garbage Collection

R_i can garbage-collect values in *data* that it knows are obsolete. When a writing client finishes its write, *i.e.*, when it is about to enter its *wok* phase, it informs all the replicas of this fact with a $\langle secure, t, mid\rangle$ message. If R_i receives this message, then it knows that it never needs to return a value with tag less than t in the future, since

[3] In fact, it would be correct to return any secured value with tag greater than t. But we choose for the read to return exactly v, as it somewhat simplifies the proof.

input $\mathbf{recv(m)}_{j,i}$ where $(m = \langle read, t, mid \rangle)$
Effect:
  if $\exists v : (v, t, *) \in data$ then
    $(v', t') \leftarrow$ choose $\{v \mid (v, t, *) \in data\}$
    $msg[j] \leftarrow \langle read\text{-}ok, v', t', mid \rangle$
  else
    $(v', t') \leftarrow \mathrm{maxst}(data)$
    $msg[j] \leftarrow \langle read\text{-}ok, v', t', mid \rangle$

input $\mathbf{recv(m)}_{j,i}$ where $(m = \langle write, v, t, mid \rangle)$
Effect:
  $data \leftarrow data \cup \{(v, t, 0)\}$
  $msg[j] \leftarrow \langle write\text{-}ok, mid \rangle$

input $\mathbf{recv(m)}_{j,i}$ where $(m = \langle gossip, v, t \rangle)$
Effect:
  $data \leftarrow data \cup \{(v, t, 1)\} \backslash \{(v, t, 0)\}$
  for all $j \in \mathcal{D}$ do
    $msg[j] \leftarrow \langle wwrite, \{i\}, t \rangle$

input $\mathbf{recv(m)}_{j,i}$ where $(m = \langle secure, t, mid \rangle)$
Effect:
  if $\exists v : (v, t, 0) \in data$ then
    for all $v : (v, t, 0) \in data$ do
      $data = data \cup (v, t, 1) \backslash \{(v, t, 0)\}$

input $\mathbf{fail}_i$
Effect:
  stop taking locally-controlled steps

output $\mathbf{send(m)}_{i,j}$
Precondition:
  $msg[j] = m$
Effect:
  $msg[j] \leftarrow \perp$

internal $\mathbf{gossip}_i$
Precondition:
  $\exists v, t, : (v, t, 1) \in data$
Effect:
  $(v', t') \leftarrow$ choose $\{(v, t) \mid (v, t, 1) \in data\}$
  for all $j \in \mathcal{R}$ do
    $msg[j] \leftarrow \langle gossip, v', t' \rangle$

internal $\mathbf{gc}_i$
Precondition:
  $\exists v, t : (v, t, 1) \in data$
Effect:
  $t \leftarrow$ choose $\{t' \mid (v, t', 1) \in data\}$
  for all $v', t' : ((v', t', *) \in data) \wedge (t' < t)$ do
    remove $(v', t', *)$ from $data$

Figure 5-8: R_i transitions.

at least $f + 1$ replicas have a value with tag at least as large as t, and the directories know this fact. Then R_i can garbage-collect all values with tag less than t.

Specifically, it R_i receives a $\langle secure, t, mid \rangle$ message, then if it has a value v with tag t in $data$, it marks that value as secure, by adding $(v, t, 1)$ to $data$, and removing $(v, t, 0)$ from $data$, if needed. If it doesn't have a value with tag t in $data$, then it just adds $(v, t, 1)$ to $data$.

At any point in its execution, R_i can choose to garbage-collect old values. It does this by finding some secured value $(v, t, 1) \in data$, and then removing all values with tag less than t from $data$. If a client ever asks to read a value with tag less than t, then R_i can instead return a value with tag t or higher.

Other Actions

Lastly, if R_i receives a $fail_i$ action, it stops taking any more locally-controlled steps.

5.6 Directory Algorithm

A directory stores the set of replicas that it thinks has a latest value of x, and the tag for that value. Each directory D_i has a variable $utd \subseteq R$, where utd stands for *up-to-date*. utd represents a set of replicas with a latest value of x. D_i also has a variable $tag \in T$, which is the tag associated with that latest value. That is, all the replicas input $\mathbf{recv(m)}_{j,i}$ where $((m = \langle rread, mid\rangle) \vee (m = \langle wread, mid\rangle))$

Effect:

  if $(m = \langle rread, mid\rangle)$ then

    $msg[j] \leftarrow \langle rread\text{-}ok, utd, tag, mid\rangle$

  else

    $msg[j] \leftarrow \langle wread\text{-}ok, tag, mid\rangle$

input $\mathbf{fail}_i$

Effect:

  stop taking locally-controlled steps

output $\mathbf{send(m)}_{i,j}$

Precondition:

  $msg[j] = m$

Effect:

  $msg[j] \leftarrow \perp$

input $\mathbf{recv(m)}_{j,i}$ where $((m = \langle rwrite, S, t, mid\rangle) \vee (m = \langle wwrite, S, t, mid\rangle))$

Effect:

  if $(t = tag)$ then

    $utd \leftarrow utd \cup S$

  else if $(t > tag)$ then

    if $|S| \geq f + 1$ then

      $utd \leftarrow S$

      $t \leftarrow tag$

  if $(m = \langle rwrite, S, t, mid\rangle)$ then

    $msg[j] \leftarrow \langle rwrite\text{-}ok, mid\rangle$

  else

    $msg[j] \leftarrow \langle wwrite\text{-}ok, mid\rangle$

Figure 5-9: D_i transitions.

in utd have a value with tag tag, unless some replicas in utd have garbage-collected that value. D_i allows clients to read and write to utd and tag.

5.6.1 Read

When D_i receives a $\langle rread, mid\rangle$ message, it just returns the current value of utd and tag by sending a message $\langle rread\text{-}ok, utd, tag, mid\rangle$. D_i does essentially the same thing when it receives a $\langle rread, mid\rangle$. The only difference is it responds with message $\langle wread\text{-}ok, utd, tag, mid\rangle$.

5.6.2 Write

When D_i receives a $\langle rwrite, S, t, mid\rangle$ message, where S is a set of processes and t is a tag, it first checks if $t > tag$. If $t < tag$, then the write is out of date, and D_i does nothing but return a write acknowledgment to the sender (this unblocks the sender, who is waiting for an ack). Otherwise, if $t = tag$, then S is a set of replicas which now have up-to-date values of x, and so D_i adds S to utd. If $t > tag$, then S is a set of replicas with a newer value of x than the replicas in D_i's current utd. Then D_i checks that $|S| > f$, and if so, updates its utd and tag to S and t, respectively. If $|S| \leq f$, the message is ignored.[4] In all cases, D_i returns an acknowledgment $\langle rwrite\text{-}ok, id\rangle$ to the sender. D_i does essentially the same thing when it receives a $\langle rwrite, S, t, mid\rangle$ message. The only difference is D_i responds with message $\langle wwrite\text{-}ok, id\rangle$.

Note that D_i needs to make sure that $|S| > f$ before setting utd to S because D_i can only tell clients about utd's which have at least one nonfailed replica, *i.e.*, utd's such that $|utd| > f$. $|S|$ may be less than or equal to f in the following scenario:

[4] We can also store all the sets with tag $t > tag$ which D_i receives, and set utd to be their union when the union contains more than f replicas. But for simplicity, we just discard sets which are too small.

There is a write using a tag greater than $D_i.tag$, which wrote to a write quorum of directories *not* containing D_i, and which was also secured at some replicas. One of the secured replicas gossips to another replica r, and then r sends a message to D_i informing D_i it is now up-to-date. Note that in this situation, D_i does not need to update *utd* to $S = \{r\}$, since a write quorum of directories already know about the up-to-date replicas.

The proof of correctness in Chapter 6 avoids the complicated situation described above. To prove that *LDR* is correct (specifically, that *LDR* guarantees liveness), the proof requires only that $|D_i.utd| > f$ at all times. Since this is true in the initial state of D_i, and since D_i checks to ensure this condition each time it changes *utd*, it is always true.

Chapter 6

LDR Correctness

In this chapter, we prove that *LDR* is an $\mathcal{F}$-srca in the network model, where $\mathcal{F}$ is defined as in Section 5.2. We need to verify that *LDR* satisfies the conditions of Definition 4.2.1. We recall the four requirements of an $\mathcal{F}$-srca in the network model:

1. $LDR \times N$ is an $\mathcal{F}$-srca.
2. $\forall (i,j) \in \mathcal{C} \times \mathcal{S}\ \exists \mathcal{M}_{i,j} \subseteq \mathcal{M}\ \forall m \in \mathcal{M}_{i,j} : \mathit{send}(m)_{i,j} \in \mathit{out}(C_i) \wedge \mathit{recv}(m)_{j,i} \in \mathit{in}(S_i)$.
3. $\forall (i,j) \in \mathcal{S} \times \mathcal{I}\ \exists \mathcal{M}'_{i,j} \subseteq \mathcal{M}\ \forall m \in \mathcal{M}'_{i,j} : \mathit{send}(m)_{i,j} \in \mathit{out}(S_i) \wedge (\mathit{recv}(m)_{j,i} \in \mathit{in}(C_i) \vee \mathit{recv}(m)_{j,i} \in \mathit{in}(S_i))$.
4. $\mathit{out}(C) \cap \mathit{in}(S) = \mathit{out}(S) \cap \mathit{in}(C) = \emptyset$.

Conditions 2, 3 and 4 describe the required interface of *LDR*. We can satisfy conditions 2 and 3 by the appropriate choice of sets $\mathcal{M}_{i,j}$ and $\mathcal{M}'_{i,j}$. For example, to satisfy condition 2 for $i \in \mathcal{C}$ and $j \in \mathcal{R}$, we choose $\mathcal{M}_{i,j} = \mathcal{M}_{CR}$. Clearly, all of the conditions in 2 and 3 can be satisfied in a similar way. Condition 4 can be verified by inspection of the client, replica and directory signatures in Figures 5-2, 5-3 and 5-4, respectively.

For the rest of this chapter, fix N to be a reliable network for *LDR*, where a reliable network is as defined in Section 4.2. Also fix U to be a user for *LDR*, as defined in Section 3.2. We concentrate on proving condition 1 of Definition 4.2.1, *i.e.*, that $LDR \times N$ is an $\mathcal{F}$-srca. We must show that $LDR \times N$ satisfies the well-formedness and atomicity conditions of Definition 3.3.1, and the liveness condition of Definition 3.3.2. We start by showing well-formedness. Then we show liveness, and finally, atomicity.

6.1 Well-formedness

We show that $LDR \times N$ satisfies the well-formedness condition of Def. 3.3.1. Let $\beta \in \mathit{traces}(LDR \times N \times U)$. We see by inspection of the client transitions in Figure 5-7 that a client outputs at most one response for each user invocation. Thus, since

U preserves well-formedness for LDR, β is well-formed. The details of the argument are omitted.

6.2 Liveness

Now we show that $LDR \times N$ satisfies the liveness condition of Def. 3.3.2. Recall that LDR tolerates the failure pattern $\mathcal{F}$, consisting of any number of client failures, up to f replica failures, and any number of directory failures, as long as some read and write quorum of directories never fail. We will show that LDR is live by showing that a read or write operation by a non-failing client cannot block forever. This is because the only time a client read or write blocks is when the client is waiting to receive acknowledgments from some replicas or directories. Since a sufficient number of replicas and directories always stay alive, they will send acknowledgments to unblock the client.

We first prove two lemmas, then prove the liveness theorem.

The first lemma says that the utd at any directory always contains at least $f+1$ replicas.

Lemma 6.2.1 *Let s be any state of an execution $\alpha \in execs(LDR \times N \times U)$, and let $i \in \mathcal{D}$. Then $|(s|D_i).utd| \geq f+1$.*

Proof. The lemma holds in the initial state s_0 of α, since $(s_0|D_i).utd = \mathcal{R}$ for all $i \in \mathcal{D}$, and $|R| \geq f+1$. Also, whenever D_i changes its utd, then either $|D_i.utd| \geq f+1$ already and D_i adds an element to its utd, or D_i first checks that a set S has size at least $f+1$, before setting $D_i.utd$ to S. Thus, $D_i.utd$ always has at least $f+1$ elements. □

The next lemma says that any utd of replicas which a client tries to read from contains at least $f+1$ replicas.

Lemma 6.2.2 *Let s be any state of $\alpha \in execs(LDR \times N \times U)$, and let $i \in \mathcal{C}$. If $(s|C_i).phase = rrr$, then $|(s|C_i).utd| \geq f+1$.*

Proof. By inspection of C_i's transitions in Figure 5-7, we see that C_i always reads utd from some directory in phase rdr. Lemma 6.2.1 shows that the utd at every directory always has size at least $f+1$. Thus, when C_i enters phase rrr, we have $|C_i.utd| \geq f+1$. □

We now state the theorem that every fair execution of $LDR \times N \times U$ is live.

Theorem 6.2.3 *Let $\alpha \in fairexecs(LDR \times N \times U)$, and suppose there exists $F \in \mathcal{F}$ such that all $fail$ events in α occur at endpoints in F. Then every invocation at a non-failing client has a corresponding response in α.*

Proof. Let $i \in \mathcal{C}$ be a non-failing client. We first show that if $read_i$ occurs in α, then $\textit{read-ok}(*)_i$ occurs later in α. We do this by showing that $C_i.phase$ takes on values rdr, rdw, rrr, and rok in order, and C_i eventually outputs $\textit{read-ok}(*)_i$ after $C_i.tag = rok$.

After $read_i$ occurs, C_i sends $\langle rread, mid\rangle$ messages to all the directories, and sets $phase$ to rdr. In phase rdr, C_i waits to receive $\langle \textit{rread-ok}, S, t, id\rangle$ messages from a read quorum of directories. Since a read quorum of directories is always alive, C_i eventually receives these messages from some read quorum of directories. Then it sends $\langle rwrite, utd, tag, mid\rangle$ messages to all the directories, and sets $phase$ to rdw. In phase rdw, C_i waits to receive $\langle \textit{rwrite-ok}, id\rangle$ from a write quorum of directories. Since a write quorum of directories is always alive, C_i eventually receives these messages from some write quorum of directories. Then C_i sends $\langle read, tag, mid\rangle$ messages to all the replicas in utd, and sets $phase$ to rrr.

By Lemma 6.2.2, the set of replicas utd that C_i sends $\langle read, tag, mid\rangle$ messages to in phase rrr has size at least $f+1$. Since at most f replicas fail in α, there must be a non-failing replica, say $j \in utd$, which receives C_i's read message. If $R_j.data$ contains $(v, tag, *)$, for some v, R_j sends $\langle \textit{read-ok}, v, mid\rangle$ to C_j. Otherwise, R_j sends $\langle \textit{read-ok}, \mathrm{maxst}(data)_1, \mathrm{maxst}(data)_2, mid\rangle$. Note that $\mathrm{maxst}(data)_1$ is always defined. Thus, in all cases, R_j eventually responds to C_i. After receiving some replica's response, C_i sets val to v and sets $phase$ to rok. After this, $\textit{read-ok}(v)_i$ becomes the only enabled action of C_i, and by the fairness of α, this action will eventually occur. Thus, every $read_i$ in α has a corresponding response occurring later in α.

To show that every $write(*)_i$ action in α has a response $\textit{write-ok}_i$ in α, we can use a similar argument as above to show that when $write(*)_i$ occurs, C_i sets $C_i.phase$ to wdr, wrw, wdw, and wok in order, and $\textit{write-ok}_i$ eventually occurs after $C_i.phase = wok$. The details are omitted. $\square$

6.3 Atomicity

It remains to show that LDR satisfies the atomicity condition of Def. 3.3.1. Recall that $UA(LDR \times N)$ is the interface between the user U and $LDR \times N$, consisting of the $read(\textit{-ok})$ and $write(\textit{-ok})$ invocations and responses:

$$UA(LDR \times N) = \bigcup_{i \in \mathcal{C}} \left(\{read_i, \textit{write-ok}_i\} \cup \bigcup_{v \in V} \{\textit{read-ok}(v)_i, write(v)_i\} \right)$$

The following theorem says that the traces of LDR, when projected onto the user actions, satisfy the atomicity property for an atomic register.

Theorem 6.3.1 *Let $\beta \in traces(LDR \times N \times U) \,|\, UA(LDR \times N)$. Then β satisfies the atomicity property for $REG(V, v_0)$.*

For the rest of this chapter, fix an arbitrary $\alpha \in execs(LDR \times N \times U)$, and let β' be the trace corresponding to α, *i.e.*, the subsequence of α consisting only of the actions of α. Let $\beta = \beta' | UA(LDR \times N)$ be the subsequence of β' consisting only of the user actions. We will prove that β satisfies the atomicity property for $REG(V, v_0)$. Then, since α was arbitrary, any trace in $traces(LDR \times N \times U) \,|\, UA(LDR \times N)$ satisfies the atomicity property.

To prove that β satisfies atomicity, we show that β satisfies the conditions in Lemma 13.16 of [8], by defining a partial order on the complete operations in β that satisfies some properties. By Lemma 13.10 of [8], it suffices to assume that β contains only complete operations. In the following, we first make some definitions, then prove some basic facts about β, and then prove β satisfies Lemma 13.16.

6.3.1 Definitions

Let Φ be the set of complete operations in trace β. Recall from Section 2.4 that a complete operation in β is a pair consisting of an invocation event and the following corresponding response event. The interval of a complete operation in β is the consecutive subsequence of β starting with the invocation of the operation and ending with the response. Also, recall that we make similar definitions for a complete operation in α, and the interval of the operation in α. We now defining some convenient notation.

Let π be an event in β'. Denote the state immediately preceding π in α by s_π, and the state immediately following π in α by s'_π. We also use the same notation to denote the states of α preceding and following an event π in β.

Let $\phi = \pi_1 \dots \pi_n$ be an operation in β, where each π_i is an event. Define $\mathrm{ex}(\phi) = \pi_1 s'_{\pi_1} \dots s_{\pi_n} \pi_n s'_{\pi_1}$ to be the execution fragment of α corresponding to ϕ. Note that we don't include the state preceding π_1 in $\mathrm{ex}(\phi)$, but do include the state following π_n.

We now define a complete operation by a client.

Definition 6.3.2 *Let $\phi = (\iota, \rho)$ be a complete operation in β. We say ϕ is a* complete operation by C_i, *where $i \in \mathcal{C}$, if $\iota \in in(C_i)$.*

Next we define a function that assigns tags to operations. Recall that T is the set of tags.

Definition 6.3.3 *Define $\lambda : \Phi \to T \cup \{t_0\}$ by the following:*

1. *If $\phi \in \Phi$ is a read, and s is any state in $\mathrm{ex}(\phi)$ such that $(s|C_i).phase = rok$, then $\lambda(\phi) = (s|C_i).tag$.*

2. *If $\phi \in \Phi$ is a write, and s is any state in $\mathrm{ex}(\phi)$ such that $(s|C_i).phase = wok$, then $\lambda(\phi) = (s|C_i).tag$.*

Note that, if ϕ is a read, $\lambda(\phi)$ equals the tag of the value returned by ϕ. If ϕ is a write, $\lambda(\phi)$ equals the tag of the value written by ϕ. It is easy to verify that λ is well-defined. Indeed, the value of $(s|C_i).tag$ doesn't change when $(s|C_i).phase = rok$ (wok), for any s in the interval of ϕ. Moreover, $(s|C_i).tag$ is defined in every state of α.

We now define a partial order $\prec$ on Φ as follows:

Definition 6.3.4 *Let $\phi, \psi \in \Phi$.*

1. *If ϕ is a write and ψ is a read, define $\phi \prec \psi$ if $\lambda(\phi) \leq \lambda(\psi)$.*
2. *Otherwise, define $\phi \prec \psi$ if $\lambda(\phi) < \lambda(\psi)$.*

6.3.2 Lemmas

We now prove some lemmas useful for the proof of atomicity.

The first lemma says that the *tag* at a directory never decreases. This can be easily verified by inspection, and we omit the proof.

Lemma 6.3.5 *Let $i \in \mathcal{D}$, and let s, s' be two states of α, such that s' occurs after s. Then $(s'|D_i).tag \geq (s|D_i).tag$.*

The next lemma says that for any replica, the maximum secured tag at the replica does not decrease in any step of α.

Lemma 6.3.6 *Let $i \in \mathcal{R}$, and let π be an event in α. Then* $\text{maxst}((s_\pi|R_i).data)_2 \leq \text{maxst}((s'_\pi|R_i).data)_2$.

Proof. If $\pi \neq gc_i$, then, by inspection of the R_i's transitions in Figure 5-8, we see that $R_i.data$ can only have more secured values as a result of π. In particular, every action except gc_i either does not change $R_i.data$, or adds a secured value to it. Thus, the lemma holds.

If $\pi = gc_i$, then $\text{maxst}((s_\pi|R_i).data)$ cannot be garbage-collected by π. Thus, $\text{maxst}((s_\pi|R_i).data) \in (s'_\pi|R_i).data$, and so the lemma holds in this case as well. $\square$

A corollary of the above lemma is that the maximum secured tag at a replica never decreases in α.

Corollary 6.3.7 *Let $i \in \mathcal{R}$, and let s, s' be two states of α such that s precedes s'. Then* $\text{maxst}((s|R_i).data)_2 \leq \text{maxst}((s'|R_i).data)_2$.

The next lemma says that if a replica receives a write or gossip message with tag t, then later receives a message to read a value with tag t, the replica returns a value with tag at least t.

Lemma 6.3.8 *Let $i \in \mathcal{R}$, and suppose an event $\tau = recv(\langle write, *, t, * \rangle)_{*,i}$ or $\tau' = recv(\langle gossip, *, t \rangle)_{*,i}$ occurs before event $\pi = recv(\langle read, t, * \rangle)_{*,i}$ in β. If i responds to π with $\pi' = send(\langle read\text{-}ok, *, t', * \rangle)_{i,*}$, then $t' \geq t$.*

Proof. If $(*, t, *) \in (s_{\pi'}|R_i).data$, then R_i will respond with $\pi' = send(\langle read\text{-}ok, *, t, * \rangle)_{i,*}$, and so the lemma holds.

Otherwise, R_i responds with $send(\langle read\text{-}ok, *, t', * \rangle)_{i,*}$, where $t' \neq t$, and $(*, t') = \text{maxst}((s_{\pi'}|R_i).data)$. Since $(*, t, *)$ was added to $R_i.data$ by τ or τ', but $(*, t, *) \notin (s_{\pi'}|R_i).data$, this implies that $(*, t, *)$ was removed from $R_i.data$ by a gc_i event μ after τ and τ', and before π'. Let $t'' = \text{maxst}((s'_\mu|R_i).data)_2$. Then $t < t''$, since μ removed $(*, t, *)$ from $R_i.data$ but kept $(*, t'', *)$. Also, $t'' \leq t'$, by Corollary 6.3.7. Thus $t < t'$, and the lemma holds. $\square$

The next lemma says that if a replica is in the *utd* of some directory with tag t, then that replica must have previously either received a write message tagged with t from a client, or received a gossip message tagged with t from a replica. This means that the *utd* and *tag* of directories contain correct information.

Lemma 6.3.9 *Let s be any state of α, and let $j \in \mathcal{R}$, and $k \in \mathcal{D}$. Suppose $j \in (s|D_k).utd$ and $t = (s|D_k).tag$. Then one of the following is true:*

1. $t = t_0$.

2. *$\exists i \in \mathcal{C}$ such that the event $recv(\langle write, *, t, * \rangle)_{i,j}$ occurred in α before s.*

3. *$\exists i \in \mathcal{R}$ such that the event $recv(\langle gossip, *, t \rangle)_{i,j}$ occurred in α before s.*

Proof. If $t = t_0$, then we are done. So suppose $t \neq t_0$. Let $C_{i'}$ be a writing client which wrote (utd', t) to D_k, *i.e.*, let $i' \in \mathcal{C}$ be such that $recv(\langle wwrite, utd', t, * \rangle)_{i',k}$ occurred in α before s. $C_{i'}$ must exist, since $t \neq t_0$, and D_k only changes $D_k.tag$ when it receives a $\langle wwrite, *, *, * \rangle$ message from some client. Furthermore, we must have $utd' \subseteq (s|D_k).utd$. This is because the only way that $D_i.utd$ can change without $D_i.tag$ changing is if elements are added to $D_i.utd$ with the same tag.

Now, there are two possibilities, either $j \in utd'$, or $j \in ((s|D_k).utd) \backslash utd'$.

Consider the former case first. Since $C_{i'}$ sent $\langle wwrite, utd', t, * \rangle$ in phase *wdw*, it must have previously done $send(\langle write, *, t, * \rangle)_{*,i'}$ during phase *wrw*, and received acknowledgments $\langle write\text{-}ok, * \rangle$ from a set utd' of replicas. Since $j \in utd'$, R_j must have acknowledged $C_{i'}$, and so R_j must have received $C_{i'}$'s $\langle write, *, t, * \rangle$ message before s. Thus, $recv(\langle write, *, t, * \rangle)_{i',j}$ occurred before s.

In the second case, D_k must have added j to $D_k.utd$ with tag t, *i.e.*, $recv(\langle wwrite, \{j\}, t \rangle)_{j,k}$ must have occurred before s. So, R_j must have done $send(\langle wwrite, \{j\}, t \rangle)_{j,k}$ before s. But R_j only does $send(\langle wwrite, \{j\}, t \rangle)_{j,k}$ if a $recv(\langle gossip, *, t \rangle)_{i,j}$ occurred previously, for some $i \in \mathcal{R}$. Thus, if $j \in ((s|D_k).utd) \backslash utd'$, the third case of the lemma holds. □

The next lemma says that if a client reads a certain tag t from the directories during its *rdr* phase, then it will return a value with tag at least as large as t.

Lemma 6.3.10 *Let ϕ be a complete read operation by C_i, and let t be the greatest value of $C_i.tag$ during phase rdr of ϕ. That is, $t = \max_{t'}\{t' \mid \exists s \text{ a state} : (s \text{ is in the interval of } \phi) \wedge ((s|C_i).tag = t') \wedge ((s|C_i).phase = rdr)\}$. Then $\lambda(\phi) \geq t$.*

Proof. If $t = t_0$, then since any tag t that a replica returns in a $\langle read\text{-}ok, *, t, * \rangle$ message is at least t_0, the value of $C_i.tag$ during phase *rok* of ϕ is at least t_0, and so the lemma holds.

Suppose that $t > t_0$, and let S be a set of replicas corresponding to t which C_i read. That is, during phase *rdr* of ϕ, the event $recv(\langle rread\text{-}ok, S, t, * \rangle)_{j,i}$ occurred, for some $j \in \mathcal{D}$.[1] Let $\pi = send(\langle rread\text{-}ok, S, t, * \rangle)_{i,j}$ be the send event corresponding to the

[1] S may not be unique, as a replica which received a gossip message may have been added to the *utd*'s of some directories but not others.

$recv(\langle \mathit{rread\text{-}ok}, S, t, *\rangle)_{j,i}$ event. Then $(s'_\pi|D_j).utd = S$, and $(s'_\pi|D_j).tag = t$. So, by Lemma 6.3.9, every replica $j \in S$ must have received a $\langle write, *, t, *\rangle$ or $\langle gossip, *, t\rangle$ message from a client or replica before state s'_π.

In phase rrr of ϕ, C_i will try to read from all the replicas in S by doing $send(\langle read, t, *\rangle)_{i,k}$, for all $k \in S$. Since $|S| \geq f+1$ by Lemma 6.2.1, one of the replicas R_k eventually replies with $send(\langle \mathit{read\text{-}ok}, *, t', *\rangle)_{k,i}$. As we argued above, R_k must have received a $\langle write, *, t, *\rangle$ or $\langle gossip, *, t\rangle$ message earlier. Then, by Lemma 6.3.8, we have $t' \geq t$. After receiving the reply from R_k, C_i sets $C_i.tag$ to t', and sets $C_i.phase = rok$. Thus, $\lambda(\phi) = t'$, and so $\lambda(\phi) \geq t$. □

Finally, we give a lemma that says that when a read operation ϕ completes, there is a write quorum of directories all of which have *tag* at least as high as $\lambda(\phi)$.

Lemma 6.3.11 *Let ϕ be a complete read operation in α, and let s be any state of α after ϕ finishes. Then $\exists Q \in \mathcal{Q}_W\ \forall j \in Q : (s|D_j).tag \geq \lambda(\phi)$.*

Proof. It suffices to prove this lemma when s is the state after ϕ finishes, since by Lemma 6.3.5, if the lemma is true for s, it is true for any state later than s.

Let t be the highest value of $C_i.tag$ during phase rdr of ϕ. By Lemma 6.3.10, $t \leq \lambda(\phi)$. We consider the only two possible cases: either C_i asked to read t from some replicas, and a replica returned a value with tag t. Or, C_i asked to read t from some replicas, but a replica returned a value with tag higher than t.

If the first case holds, then the claim is true because C_i propagates $t = \lambda(\phi)$ to a write quorum of directories during phase rdw of ϕ. Then, a write quorum of directories have tag at least as high as $\lambda(\phi)$ after ϕ finishes.

Suppose the second case holds, and consider the replica R_k from which C_i read the value tagged by $\lambda(\phi)$. Since C_i asked to read a value tagged by t from R_k, but R_k returned a value tagged by $\lambda(\phi)$, then $\lambda(\phi) = \mathrm{maxst}((s'|R_k).data)_2$ for some state s' before s. Thus, tag $\lambda(\phi)$ was secured at R_k before s. Let s'' be the first state in α in which there exists some replica R_l which has secured $\lambda(\phi)$ in $R_l.data$. Clearly, s'' occurs no later than s. By inspection of the pseudo-code in Figures 5-7, 5-8, and 5-9, we see that R_l must have received a $\langle secure, \lambda(\phi), *\rangle$ message before s''. Also by inspection, we see that only clients can send *secure* messages. Thus, there must be a client writing a value with tag $\lambda(\phi)$ which started securing the value before state s. This client must have completed its wdw phase before s, since it can only send out *secure* messages after it receives a write quorum of acknowledgments for its $\langle wwrite, *, \lambda(\phi), *\rangle$ messages in phase wdw. Therefore, there exists a write quorum of directories with tag $\lambda(\phi)$ before s, and they have a tag at least as high as $\lambda(\phi)$ in s. □

6.3.3 Proof of Atomicity

We now prove Theorem 6.3.1, which states that any trace of $LDR \times N \times U$, projected onto the user actions of $LDR \times N$, satisfies the atomicity property.

Proof of Theorem 6.3.1. Recall that $\beta = \beta'|UA(LDR \times N)$, where β' is the trace corresponding to $\alpha \in execs(LDR \times N \times U)$. Also recall the definition of $\prec$ in Def. 6.3.4. We will prove β and $\prec$ satisfy Lemma 13.16 of [8]. Then, since α was arbitrary, this implies that any trace in $traces(LDR \times N \times U)|UA(LDR \times N)$ satisfies the atomicity property.

It suffices to show β satisfies conditions 2, 3, and 4 of Lemma 13.16, because condition 1 follows automatically. We show these conditions in the following 3 lemmas. Let ϕ and ψ be two complete operations in β.

Lemma 6.3.12 *(Condition 2) If the response event for ϕ precedes the invocation event for ψ, then $\phi \not\prec \psi$.*

Proof. We consider four cases.
Case 1: ϕ and ψ are both writes. Let $Q_1 \in \mathcal{Q}_W$ be the quorum ϕ writes to during its wdw phase, and let $Q_2 \in \mathcal{Q}_R$ be the quorum ψ reads from during its wdr phase. Then, since $\lambda(\phi)$ is the tag ϕ uses for its write, and $\lambda(\phi)$ is written to every directory in Q_1 by ϕ before ψ starts, we have $\forall D \in Q_1 : D.tag \geq \lambda(\phi)$ before ψ starts. By the quorum intersection property, $\exists i \in Q_1 \cap Q_2$. By Lemma 6.3.5, the tag at D_i never decreases. Thus, ψ reads a tag at least as large as $\lambda(\phi)$ in its wdr phase, and will choose a tag greater than $\lambda(\phi)$ during its wrw phase. Thus, $\lambda(\psi) > \lambda(\phi)$, and $\phi \not\prec \psi$.

Case 2: ϕ is a write and ψ is a read. By the same argument as in case 1, we have that ψ reads a tag at least as high as $\lambda(\phi)$ during its rdr phase. By Lemma 6.3.10, ψ returns a value with tag at least as high as the highest tag it reads during its rdr phase. Thus $\lambda(\psi) \geq \lambda(\phi)$, and $\phi \not\prec \psi$.

Case 3: ϕ is a read and ψ is a write. By Lemma 6.3.11, we know after ϕ finishes, there is a write quorum of directories with tag at least as high as $\lambda(\phi)$. Since the tags at these directories never decrease, C_j will read a tag at least as high as $\lambda(\phi)$ when it reads a read quorum of directories in phase wdr of ψ. Thus, C_j will tag its write with a tag greater than $\lambda(\phi)$, and so $\lambda(\psi) > \lambda(\phi)$, and $\phi \not\prec \psi$.

Case 4: ϕ and ψ are both reads. By the same argument as in case 3, we know that C_j will read a tag at least as high as $\lambda(\phi)$ during the rdr phase of ψ. Then, by Lemma 6.3.10, ψ returns a value with at least this tag, and so $\lambda(\psi) \geq \lambda(\phi)$, and $\phi \not\prec \psi$.

This proves the lemma for all possible cases of ϕ and ψ, so the lemma holds. $\square$

Lemma 6.3.13 *(Condition 3) A write operation is totally ordered with respect to any other operation.*

Proof. Let ϕ be a write operation, and let ψ be any other operation. Suppose first that ψ is a write operation. If ϕ and ψ are operations by different clients, then $\lambda(\phi)$ and $\lambda(\psi)$ will differ in their second coordinates, which is the ID of the process performing the operation. If they are operations by the same client, then ϕ must finish before ψ starts, or vice versa, since any execution of LDR is well-formed, as we argued in Section 6.1. Then by the same argument as we made in case 1 in the proof of Lemma 6.3.12, ϕ and ψ choose different tags. Thus, either $\phi \prec \psi$ or vice versa.

Now suppose ψ is a read. Then either $\lambda(\phi) \leq \lambda(\psi)$ or $\lambda(\phi) > \lambda(\psi)$. By the definition of $\prec$, $\phi \prec \psi$ in the first case, and $\psi \prec \phi$ in the second case. Thus, ϕ is ordered with respect to any other operation. $\square$

Lemma 6.3.14 *(Condition 4) The value returned by each read operation is the value written by the last preceding write operation according to $\prec$ (or the default value of x, if there is no such write).*

Proof. Let ϕ be a read operation. If there was no write preceding ϕ, then no replica changes the value of its *data* variable. Since *data* initially contains only $(v_0, t_0, 1)$ at all the replicas, then any read ϕ can only return the default value of x, v_0.

Otherwise, let ψ be the write preceding ϕ according to $\prec$, *i.e.*, $\psi = \max_{\prec}\{\omega \mid \omega$ is a write $\wedge\ \omega \prec \phi\}$. Let ω be the write which wrote the value that ϕ returned, *i.e.*, ω is a write such that $\lambda(\phi) = \lambda(\omega)$. We claim $\omega = \psi$. Indeed, since $\lambda(\phi) = \lambda(\omega)$, we have $\omega \prec \phi$. And if $\omega \prec \omega'$, where ω' is a write, then $\lambda(\phi) = \lambda(\omega) < \lambda(\omega')$, so that $\phi \prec \omega'$. Thus, ω is the largest write preceding ϕ, and so $\omega = \psi$. $\square$

Chapter 7

Performance Analysis

In this chapter, we analyze the communication and time complexity of *LDR*. We also compare the performance of *LDR* with that of *ABD*. The reason we compare *LDR* with *ABD* is that these two algorithms, unlike most other algorithms described in Chapter 1, do not need mechanisms like distributed locking or atomic broadcast to function correctly; they rely only on a reliable message passing network. Thus, it is possible to analyze the performance of *LDR* and *ABD* using simple properties of the network, such as an upper bound on the time it take to transfer a message of a certain size, without making assumptions on the performance of the locking or broadcast mechanism.

7.1 Communication Complexity

To measure the communication complexity of *LDR*, we count the number of messages that a client sends and receives during an operation, weighted by the size of the messages. We differentiate between two types of messages, *data* messages and *metadata* messages. Data messages are those in which the value of x is sent. Metadata messages are all the remaining messages, *e.g.*, tags, *mid*'s, and *utd*'s. The main assumption is that the size of metadata messages is small compared to the size of data messages. For example, if *LDR* is used in a replicated file system, then the size of the data is the size of a typical file, which may be at least several kilobytes. Meanwhile, the size of the metadata is on the order of bytes. In particular, we assume the size of each metadata item is 1, the size of a set of metadata items equals the cardinality of the set, and the size of x is d, where $d \gg 1$. For example, if the size of *utd* is $f+1$, then the size of the message $\langle \mathit{rread\text{-}ok}, \mathit{utd}, \mathit{tag}, \mathit{mid} \rangle$ is $f+4$, where we assume that *rread-ok*, *tag*, and *mid* each have size 1. Similarly, the size of message $\langle \mathit{read\text{-}ok}, \mathit{val}, \mathit{tag}, \mathit{mid} \rangle$ is $d+3$, where *val* is a value of x. We believe that for many applications of *LDR*, these assumptions are reasonable.

Another assumption we make is that a process only sends the *minimum* number of messages necessary to perform an operation. For example, if a client needs to read *utd* from a read quorum of directories, then it chooses some read quorum, and sends messages *only* to the directories in that quorum. Similarly, if a client needs to read the newest value of x from a set of replicas, it chooses one replica in the set, and sends a read message only to that replica. Note that the pseudocode in Figures 5-7, 5-8 and 5-9 indicates that we send the *maximum* number of messages needed to perform an operation, so that, *e.g.*, to read from a read quorum, a client sends messages to all the directories, and waits to hear back from *any* read quorum. We wrote the pseudocode this way in order to ensure liveness, since we cannot be sure that any particular processes we contact have not failed. In practice, however, the rate of failure is low, so that for example, a client can use timeouts to contact a new quorum if the current quorum doesn't respond. Assuming the network is well-behaved, the client will eventually succeed in hearing from some quorum, and the modified algorithm will still exhibit liveness. To further simplify our analysis, we will assume that whenever one process expects to hear from another process, the latter process eventually responds. In particular, this means that for our analysis, we assume that *no failures occur*. Lastly, we assume that all the read and write quorums of directories have size $f + 1$.

Below, we compute the communication complexity of a *LDR* read and write operation. Then we compute the costs of the operations using *ABD*, and compare the costs.

7.1.1 *LDR* Read

We refer to the pseudocode for a client C_i's read operation in Figure 5-7. In the $read_i$ action, C_i sends a $\langle rread, mid\rangle$ message of size 2 to $f + 1$ directories. During phase rdr, C_i receives messages $\langle rread\text{-}ok, S, t, id\rangle$ of size $f + 4$ from $f + 1$ directories, and also sends messages $\langle rwrite, utd, tag, mid\rangle$ of size $f + 4$ to $f + 1$ directories. In phase rdw, C_i receives messages $\langle rwrite\text{-}ok, id\rangle$ of size 2 from $f+1$ directories, and it sends out a $\langle read, tag, mid\rangle$ message of size 3 to one replica. In phase rrr, C_i receives a $\langle read\text{-}ok, v, t, id\rangle$ message of size $d + 3$ from one replica. Thus in total, C_i sends and receives messages of size $d + 2f^2 + 14f + 18$.

7.1.2 *LDR* Write

When C_i does $write(v)_i$, it first sends a $\langle wread, mid\rangle$ message of size 2 to $f + 1$ directories. During phase wdr, C_i receives messages $\langle wread\text{-}ok, t, id\rangle$ of size 3 from $f + 1$ directories, and also sends messages $\langle write, v, tag, mid\rangle$ of size $d + 3$ to $f + 1$ replicas. In phase wrw, C_i receives messages $\langle write\text{-}ok, id\rangle$ of size 2 from $f+1$ replicas, and sends out $\langle wwrite, acc, tag, mid\rangle$ messages of size $f + 4$ to $f + 1$ directories. In phase wdw, C_i receives messages $\langle wwrite\text{-}ok, id\rangle$ of size 2 from $f + 1$ directories, and sends out $\langle secure, tag, mid\rangle$ messages of size 3 to $f + 1$ replicas. Thus in total, C_i sends and receives messages of size $(f + 1)d + f^2 + 20f + 19$.

7.1.3 *ABD* Read

Recall the description of the *ABD* algorithm in Section 2.5. Based on that description, we can write pseudo-code implementing the *ABD* algorithm in a similar way to the

	LDR	*ABD*	Ratio (asympt.)
Read	$d + 2f^2 + 14f + 18$	$(2f + 2)d + 10f + 10$	$1/(2f + 2)$
Write	$(f + 1)d + 2f^2 + 20f + 19$	$(f + 1)d + 10f + 10$	1

Figure 7-1: *LDR* and *ABD* communication complexity.

pseudo-code implementing *LDR*. We do not present the *ABD* pseudo-code in this thesis, and will only discuss which messages are sent by *ABD*.

Let C_i be a client in *ABD*, where C_i plays the same role as it does in *LDR*. First, C_i sends $\langle rread, mid\rangle$ messages of size 2 to $f + 1$ servers (directories), to read the tag and values stored at the servers. The servers respond with $f + 1$ messages $\langle rread\text{-}ok, val, tag, mid\rangle$, where val is the value of the data. Each such message has size $d + 3$. Then C_i sends messages $\langle rwrite, val, tag, mid\rangle$ of size $d + 3$ to $f + 1$ servers to write back the value. Finally, C_i receives $f + 1$ messages $\langle rwrite\text{-}ok, mid\rangle$ of size 2. Thus, the total communication is $(2f + 2)d + 10f + 10$.

7.1.4 *ABD* Write

When C_i does $write(v)_i$ in *ABD*, it first sends $\langle wread, mid\rangle$ messages of size 2 to $f + 1$ servers, to read their tags. The servers respond with $f + 1$ messages $\langle wread\text{-}ok, tag, mid\rangle$. Then C_i sends out messages $\langle wwrite, v, tag, mid\rangle$ of size $d + 3$ to $f + 1$ servers. Finally, the servers send back $f + 1$ messages $\langle wwrite\text{-}ok, mid\rangle$, of size 2. Thus, the total communication is $(f + 1)d + 10f + 10$.

7.1.5 Comparison of *LDR* and *ABD*

Figure 7-1 summarizes the communication cost of *LDR* and *ABD* read and write operations. It also gives the ratio of the cost of a *LDR* operation and an *ABD* operation, in the limit that $d \to \infty$, and f is constant. Note that the cost of a *LDR* write is the same asymptotically as that of an *ABD* write. On the other hand, the cost of a *LDR* read is $\frac{1}{2f+2}$ the cost of a *ABD* read. Also note that the communication costs of a *LDR* read and write are asymptotically optimal. That is, any data replication algorithm must read at least one copy of the replicated data, as *LDR* does. And, to ensure the data survives the failure of f replicas, the algorithm must write the data to at least $f + 1$ replicas, as *LDR* does.

The reason a *LDR* read is more efficient than an *ABD* read is because *LDR* does not perform the expensive *read-propagation* phase of an *ABD* read, in which the value of x is read from, then written back, to a quorum of servers. This is because any value that a client reads from a replica in *LDR* is guaranteed to be written at at least $f + 1$ replicas, so the value does not need to be propagated. The efficiency of *LDR*'s read operation is significant because the workload of most data replication services contains far more reads than writes.

7.2 Time Complexity

To analyze the time complexity of *LDR*, we make similar assumptions as in Section 7.1. That is, we assume that the size of the data is d, the size of each metadata item is 1, $d \gg 1$, and $d \gg f$. We assume that the time to transmit a message across the network is proportional to the size of the object. In particular, we assume that it takes time d to transmit the data from one process to another, and it takes time 1 to transmit one item of metadata. We also assume that if a process sends a message to a group of other processes, it sends those messages in parallel, so that the time to send all the messages is equal to the size of the largest message in the group. For example, if a client needs to send $\langle rwrite, utd, tag, mid \rangle$ messages to a quorum of directories, where utd has size $f+1$, then the client performs all the sends in parallel, so that it takes time $f+4$ to send the messages to all the directories. We again assume that processes don't fail, and that they respond instantly to messages they receive which require acknowledgments.

Below, we compute the time complexity of a *LDR* read and write operation. Then we compute the time of an *ABD* read and write operation, and compare the costs to that of *LDR*.

7.2.1 *LDR* Read

When a client C_i performs $read_i$, it first sends a $\langle rread, mid \rangle$ message of size 2 to some directories. Then in phase rdr, C_i receives some messages $\langle rread\text{-}ok, S, t, it \rangle$ of size $f+4$, and sends out a $\langle rwrite, utd, tag, mid \rangle$ message of size $f+4$. In phase rdw, C_i receives some messages $\langle rwrite\text{-}ok, id \rangle$ of size 2, and sends out a message $\langle read, tag, mid \rangle$ of size 3. Finally, in phase rrr, C_i receives a message $\langle read\text{-}ok, v, t, id \rangle$ of size $d+3$. Thus, the total time for the read operation is $d + 2f + 18$.

7.2.2 *LDR* Write

When C_i performs $write(v)_i$, it first sends a $\langle wread, mid \rangle$ message of size 2 to some directories. Then in phase wdr, C_i receives some messages $\langle wread\text{-}ok, t, it \rangle$ of size 3, and sends out some messages $\langle write, val, tag, mid \rangle$ of size $d+3$. In phase wrw, C_i receives some messages $\langle write\text{-}ok, id \rangle$ of size 2, and sends out some messages $\langle wwrite, acc, tag, mid \rangle$ of size $f+4$. Finally, in phase wdw, C_i receives some messages $\langle wwrite\text{-}ok, mid \rangle$ of size 2, and sends out some messages $\langle secure, tag, mid \rangle$ of size 3. Thus, the total time for the write operation is $d + f + 19$.

7.2.3 *ABD* Read

We will consider the implementation of an *ABD* read described in Section 7.1.3. First, client C_i sends some $\langle rread, mid \rangle$ message of size 2. Then, it receives some $\langle rread\text{-}ok, val, tag, mid \rangle$ messages of size $d+3$, and sends out some $\langle rwrite, val, tag, mid \rangle$ messages of size $d+3$. Lastly, it receives some messages $\langle rwrite\text{-}ok, mid \rangle$ of size 2. Thus, the total time for the read operation is $2d + 10$.

	LDR	*ABD*	Ratio (asympt.)
Read	$d + 2f + 18$	$2d + 10$	1/2
Write	$d + f + 19$	$d + 10$	1

Figure 7-2: *LDR* and *ABD* time complexity.

7.2.4 *ABD* Write

We consider the implementation of an *ABD* write described in Section 7.1.4. First, client C_i sends some $\langle wread, mid \rangle$ message of size 2. Then, it receives some $\langle wread\text{-}ok, tag, mid \rangle$ messages of size 3, and sends out some messages $\langle wwrite, v, tag, mid \rangle$ of size $d+3$. Finally, it receives some messages $\langle wwrite\text{-}ok, mid \rangle$ of size 2. Thus, the total time for the write operation is $d + 10$.

7.2.5 Comparison of *LDR* and *ABD*

Figure 7-2 summarizes the read and write time complexity of *LDR* and *ABD*, and gives the ratio of the costs in the limiting case of $d \to \infty$, and f fixed. For both the *LDR* read and write, the time of the operation is dominated by the time to read and write the data. In this sense, the time complexity of *LDR* is optimal when the size of the data is large.

As with the communication complexity, the time complexity of a *LDR* read is less than that of an *ABD* read, this time by a factor of 2. Again, this comes from the fact that *LDR* doesn't perform the read-propagation phase of *ABD*. That is, *LDR* only reads the data, but doesn't write it back.

c

Chapter 8

Lower Bounds

In this chapter, we prove two lower bounds on the inherent costs of any $\mathcal{F}$-srca. We prove these results in the atomic servers model, described in Chapter 4.3, instead of the network model. Since the atomic servers model is simpler, it allows us to prove the lower bounds more easily, and also shows more clearly why they arise. It is not difficult to adapt the lower bounds into corresponding results in the network model. However, we omit the formal translation in this thesis.

The first lower bound says that for any $\mathcal{F}$-srca in the atomic servers model tolerating the failure of up to f servers, reading clients must sometimes write to up to f servers. This means, for example, that there does not exist an $\mathcal{F}$-srca in which a reading client only reads from one server and returns the result.

The second lower bound says that given any $\mathcal{F}$-srca in the atomic servers model, if reading clients don't write copies of x during a read, then servers need to have storage proportional to the number of concurrently writing clients. Note that the precondition for the second lower bound is consistent with the conclusion of the first lower bound. The first lower bound says that a client must write *something*, *e.g.*, some metadata, to the servers during some read. It does not say that clients must write the value of x. The second lower bound says that if the reading client never writes the value of x, then the servers need potentially large storage. For example, *LDR* is an $\mathcal{F}$-srca in which clients write (to directory servers) during a read, but never write values of x. Then, the second lower bound implies that replicas (servers) need to keep copies of all the values being concurrently written. In *LDR*, this is implemented by having replicas store the values of all concurrent writes in their *data* variable.

Together, these two lower bounds justify some of the constructions we used in *LDR*. They say, for example, that we have not been too profligate in allowing our clients to write during a read, or in allowing our replicas to store a list of values of x.

Below, we first give the definition, statement and proof of the first lower bound, then do the same for the second lower bound.

8.1 Write on Read Necessity

We prove that for any $\mathcal{F}$-srca in the atomic servers model tolerating the failure of f servers, a client must sometimes write to at least f servers during a read. In the atomic servers model, it is easy to define the meaning of "a client must write to at least f servers". We consider a write to be any operation which may change the state of a server, and a read as an operation which cannot change a server's state. Recall from Chapter 4.3 that the only interface to a server in the atomic servers model are its $read_{*,*}$ and $modify(*)_{*,*}$ actions. Then, we can say that client C_i writes to at least f servers during a read if it invokes $modify(*)_{i,j}$, for at least f different S_j, during the read.

The intuition for this lower bound is that for any $\mathcal{F}$-srca, the value of x at certain points in an execution of the $\mathcal{F}$-srca is ambiguous, *i.e.*, it is possible for a reading client to return different values for x. In this situation, a reading client must write to some servers to record which value of x it returned. Since any server to which it writes may later fail, the client must write to at least f servers, to ensure that later readers know which value it returned.[1]

We will first give some definitions to formalize the lower bound, then state the lower bound and give its proof.

8.1.1 Definitions

Definition 8.1.1 *Let A be an $\mathcal{F}$-srca, where $\mathcal{F} = \{F \,|\, (F \subseteq 2^{\mathcal{I}}) \wedge (|F \cap \mathcal{S}| \leq f)\}$. Then we say A is an* f-srca.

Thus, an f-srca is a $\mathcal{F}$-srca tolerating the failure of up to f servers. Given a user U of A, we can similarly define an f-srca for U as an $\mathcal{F}$-srca for U tolerating up to f server failures.

In the remainder of this section, let A be an f-srca, for some f, and let $\alpha \in execs(A)$.

Definition 8.1.2 *Let $\phi = (\iota, \rho)$ be a complete operation by C_i in α. Define the* value *of ϕ, written $\chi(\phi)$, by the following:*

1. *If $\rho = read\text{-}ok(v)_i$, then $\chi(\phi) = v$.*
2. *If $\iota = write(v)_i$, then $\chi(\phi) = v$.*

Thus, the value of ϕ is the value read by ϕ if ϕ is a read, or the value written by ϕ if ϕ is a write.

Definition 8.1.3 *Let ϕ be a complete operation by C_i in α, and let τ be the interval of ϕ in α.*

[1] It might seem that because up to f servers may fail, the client should write to $f+1$ servers. However, our lower bound does not imply this stronger claim. The *ABD* algorithm writes to $f+1$ servers during a read. Thus, it is an open question whether there exists an f-srca which writes to only f servers during a read.

1. *The set of servers* written by ϕ *is* $\Delta(\phi) = \{S_j \mid modify(*)_{i,j}$ *occurs in* $\tau\}$.

2. *The set of servers* read by ϕ *is* $\Gamma(\phi) = \{S_j \mid read\text{-}ok(*)_{j,i}$ *occurs in* $\tau\}$.

Thus, the set of servers written by ϕ is the set of servers at which C_i invokes $modify(*)_{i,*}$. The set of servers read by ϕ is the set of servers which respond to C_i's read with $read\text{-}ok(*)_{*,i}$.

Definition 8.1.4 *Let A be an f-srca, let $\alpha \in finexecs(A)$, and let $i \in \mathcal{C}$. Then $\alpha' = \alpha \cdot \pi_1 s_1 \ldots \pi_n s_n$ is a* read extension of α by C_i *if*

1. $\alpha' \in exec(A)$.

2. $\pi_1 s_1 \ldots \pi_n s_n$ *is the interval of a complete read operation by* C_i.

3. $\forall i, 1 \leq i \leq n : \pi_i \in acts(C_i)$.

If α' is a read extension of α by C_i, we write $\alpha \sqsubseteq_i^R \alpha'$.

Thus, a read extension of α by C_i is an execution consisting of α followed by a complete read by C_i, such that no other clients take steps during the duration of C_i's read. We define a *write extension of α by C_i* similarly, and we write $\alpha \sqsubseteq_i^W \alpha'$ if α' is a write extension of α by C_i.

8.1.2 Theorem

Theorem 8.1.5 *Let A be an f-srca, and assume $|\mathcal{C}| > 2$. There exists $\alpha \in execs(A)$ and a complete operation ϕ in α such that $|\Delta(\phi)| \geq f$.*

Proof. The proof is by contradiction. We first give an outline of the proof. Assume that there exists an f-srca A such that no read operation of an execution of A writes to more than $f-1$ replicas. Then we consider an execution α of A consisting of a write operation ϕ writing value $v_1 \neq v_0$. Recall from Chapter 2.1.1 that $\alpha(i)$ denotes the length $2i+1$ prefix of α. Consider the shortest prefix $\alpha(i^*)$ of α such that if we pause ϕ and start a read, the read can return v_1. Let α_1 be the execution $\alpha(i^*)$ appended with the read returning v_1. Also, let p be the server that changed its state from state s_{i^*-1} to s_{i^*}, if such a server exists. By atomicity, any read starting after α_1 must return v_1 or a newer value. In particular, a read that doesn't read from any server in $\Gamma(\phi)$ and doesn't read p must still return v_1 or a newer value. But such a read can't distinguish between states s_{i^*-1} and $\alpha_1.lstate$, so the read can also occur following $\alpha(i^*-1)$, when it must return v_0. This is a contradiction, and shows that A doesn't exist. Figure 8-1 shows the executions considered in the proof. We now give the details of the proof.

Let A be an f-srca, and let s_0 be an initial state of A. Let α be an execution of A consisting of a write of value $v_1 \neq v_0$ by C_w. That is, $\alpha = s_0 \pi_1 s_1 \ldots \pi_n s_n$ is a write extension of s_0 by C_w,[2] and $\pi_1 = write(v_1)_w$. The following lemma says that there exists an $i > 0$ and a read extension of $\alpha(i)$ which returns v_1.

[2]So the only client taking steps is C_w

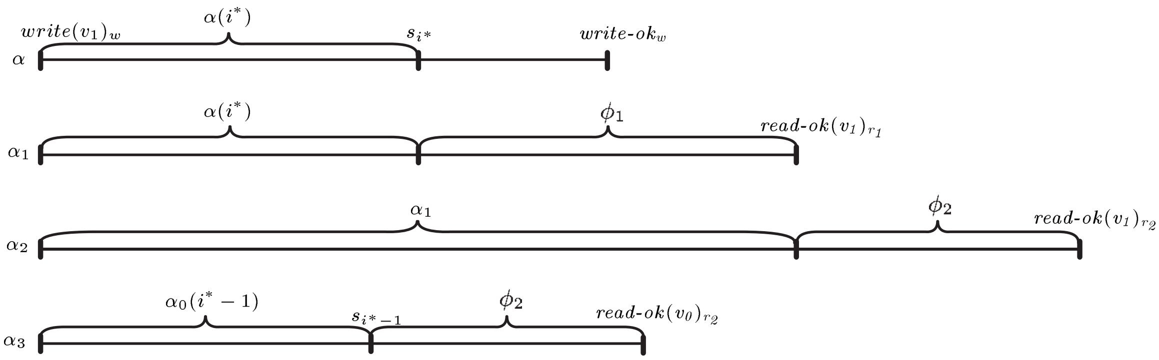

Figure 8-1: Proof of Theorem 8.1.5

Lemma 8.1.6 $\exists i > 0\ \exists j \in \mathcal{C}\ \exists \alpha' \in execs(A) : (\alpha(i) \sqsubseteq_j^R \alpha') \wedge (\alpha'.lact = read\text{-}ok(v_1)_j)\}$.

Proof. By the atomicity of A, we know that any read extension of $\alpha(0) = s_0$ must return v_0, so $i > 0$. Also by atomicity, any read extension of α must return v_1. Thus, we also have $i \neq n$. □

Let i^* be the minimum i for which there exists a read extension of $\alpha(i)$ by a client returning v_1. Note that by definition, for any $i < i^*$, all read extensions of $\alpha(i)$ return v_0. Let α_1 be a read extension of $\alpha(i^*)$ by C_{r_1} returning v_1, for some client $C_{r_1} \neq C_w$. That is, let $\alpha_1 = \alpha(i^*)\phi_1$, where ϕ_1 is the interval of a read operation by C_{r_1}. Also, let S_p be the server, if any, which changed its state from state s_{i^*-1} to s_{i^*}. That is, choose S_p such that $s_{i^*-1}|S_p \neq s_{i^*}|S_p$, if such an S_p exists. For convenience, if no server changed its state from s_{i^*-1} to s_{i^*}, we set S_p to be an arbitrary server. Note that there can be at most one server which changed its state from s_{i^*-1} to s_{i^*}, since at most one server changes its state between any two consecutive states. The following lemma says that there is read extension $\alpha_1\phi_2$ of α_1, by a client other than r_1 or w, such that ϕ_2 does not read from any process written to by ϕ_1, nor from S_p.

Lemma 8.1.7 *There exists a read extension $\alpha_1\phi_2$ of α_1 by C_{r_2}, where $r_2 \notin \{r_1, w\}$, such that $\Gamma(\phi_2) \cap (\Delta(\phi_1) \cup \{S_p\}) = \emptyset$.*

Proof. Let $F = \Delta(\phi_1) \cup \{S_p\}$. Since by the assumption on A, $|\Delta(\phi_1)| \leq f - 1$, we have $|F| \leq f$. Consider any read extension $\alpha_1\phi_2$ of α_1 by C_{r_2}. During ϕ_2, we delay the responses from all the servers in F indefinitely, while allowing all other servers to respond immediately when they receive an invocation. In ϕ_2, it seems to C_{r_2} that the processes in F have failed. However, since A tolerates the failure of up to f processes, ϕ_2 must eventually return, without ever reading from a process in F. Thus, we have $\Gamma(\phi_2) \cap (\Delta(\phi_1) \cup \{S_p\}) = \emptyset$. □

Fix a ϕ_2 with the properties described in Lemma 8.1.7. The next lemma says that ϕ_2 is a read extension of $\alpha(i^* - 1)$.

Lemma 8.1.8 $\alpha(i^* - 1)\phi_2 \in execs(A)$.

Proof. We consider the servers which may have changed their state from state s_{i^*-1} to state $\alpha_1.lstate$. From state s_{i^*-1} to s_{i^*}, only S_p can change its state. From state s_{i^*} to $\alpha_1.lstate$, only the servers in $\Delta(\phi_1)$ can change their state, by the definition of Δ. Thus, the servers which changed state from s_{i^*-1} to $\alpha_1.lstate$ are a subset of $\Delta(\phi_1) \cup \{S_p\}$. Since $\Gamma(\phi_2) \cap (\Delta(\phi_1) \cup \{S_p\}) = \emptyset$, ϕ_2 doesn't read from any server which changed its state from s_{i^*-1} to $\alpha_1.lstate$. Thus, states s_{i^*-1} to $\alpha_1.lstate$ look identical to ϕ_2. Therefore, since ϕ_2 is a read extension of α_1 starting from state $\alpha_1.lstate$, it is a valid read extension of $\alpha(i^* - 1)$ starting from state s_{i^*-1}. Thus, $\alpha(i^* - 1)\phi_2 \in execs(A)$. $\square$

We can now finish the proof of Theorem 8.1.5. By the definition of i^*, all read extensions of $\alpha(i^* - 1)$ must return v_0. However, by Lemma 8.1.8, ϕ_2 is a read extension of $\alpha(i^* - 1)$, and ϕ_2 returns v_1. This is a contradiction, and shows that A does not exist. Therefore, for any f-srca A, there exists an execution of A in which a reading client must write to at least f servers. $\square$

8.2 Proportional Storage Necessity

Recall from Chapter 1 that we informally define a *selfish* replication algorithm as one in which readers don't write the value of x, and writers only write their own value, and no other values. In the second lower bound, we prove that in any selfish f-srca, with $f > 0$, the servers need to have storage proportional to the number of concurrent writers.

We first explain why this lower bound holds only for $f > 0$. In fact, for a 0-srca, *i.e.*, an algorithm which doesn't tolerate any server failures, there is a trivial algorithm which uses only constant storage, independent of the number of concurrent writers. Namely, the algorithm always reads and writes to one server. That server needs only to store one copy of x at all times. A write is completed as soon as it takes a single atomic step, *i.e.*, write at the server. But, if the algorithm tolerates server failures, then a writing client must write to more than one server. If the writer fails in the middle of the write, its value is left in an ambiguous state. It is in this situation that the lower bound arises.

In the rest of this chapter, we formally define selfish f-srcas. We then discuss the advantages of selfish f-srcas. Then we state the lower bound on selfish f-srcas, and present its proof.

8.2.1 Definitions and Lemmas

We first define some helpful notation. Let $P \subseteq \mathcal{C} \cup \mathcal{S}$, $P = \{p_1, \ldots, p_n\}$. We let $fail_P = fail_{p_1} \cdot fail_{p_2} \cdot \ldots \cdot fail_{p_n}$ be an execution fragment in which all the processes in P fail. Given an f-srca A, $\alpha \in execs(A)$, and a value v, we let $W(v, \alpha) \subseteq \mathcal{C}$ be the set of clients that start to write v in α. That is, $W(v, \alpha) = \{i \in \mathcal{C} \mid write(v)_i \text{ occurs in } \alpha\}$.

In order to define selfish f-srcas, we first define the related notions of erasability, multiplicity, and server-exclusive executions.

Definition 8.2.1 *Let A be an f-srca, $\alpha \in finexecs(A)$, v be a value, and g a natural number. We say v is* g-erasable after α *if*

$$(\exists G \subseteq \mathcal{S} : |G| = g)\ (\forall i \in \mathcal{C} - W(v, \alpha))\ (\forall \alpha' \in execs(A)) :$$
$$(\alpha \cdot fail_G \sqsubseteq_i^R \alpha') \Rightarrow (\alpha'.lact \neq \textit{read-ok}(v)_i)$$

Note that $\alpha \cdot fail_G$ is an extension of α in which a set G of g servers fail, and $\mathcal{C} - W(v, \alpha)$ is the set of servers that do not write v during α. This definition says that v is g-erasable after α if there exists a set of g servers such that if we fail these servers, then no client which has not already started to write v in α can read v. That is, by failing some g servers, we can "erase" the value v from α.

Definition 8.2.2 *Let A be an f-srca, $\alpha \in finexecs(A)$, and let v be a value. The* multiplicity of v after α *is $m(v, \alpha) = \min\{g \mid v$ is g-erasable after $\alpha\}$.*

That is, $m(v, \alpha)$ is the minimum number of servers that need to fail to erase v from α. Intuitively, $m(v, \alpha)$ corresponds to the *number of servers* that v is "written" at after α. In fact, if $m(v, \alpha) = g$, then by failing some g servers, we can erase v from α. Thus, v is not written at more than g servers. On the other hand, no set of $g - 1$ server failures is enough to erase v, so v is written at at least g servers. Thus, if $m(v, \alpha) = g$, then v is written at exactly g servers after α.

Definition 8.2.3 *Let A be an f-srca. An execution $\alpha \in execs(A)$ is* server-exclusive *if, for any server S_j, $j \in \mathcal{S}$, any event π_1 an invocation at S_j in α, and π_2 the corresponding response to π_1 at S_j in α, there is no occurrence of an invocation at S_j between π_1 and π_2.*

This definition says that an execution is server-exclusive if no two clients ever concurrently access the same server in α, *i.e.*, each client has exclusive access to a server during α. We can think of a server-exclusive execution as one in which the servers are replaced by shared objects that return instantaneous responses to invocations, *e.g.*, shared memory.

In a server-exclusive action, every action is either an action by a client, or an action "on behalf" of a client by a server. This is because any server is accessed by one client at a time, so we can attribute the action of that server to a particular client. Based on this fact, we have the following definition:

Definition 8.2.4 *Let α be a server-exclusive execution of an f-srca A, and let π be an action in α. We say client C_i, $i \in \mathcal{C}$* initiated *π if either π is an action by C_i, or π is an action by a server S_j, $j \in \mathcal{S}$, and the last invocation at S_j was by C_i.*

Now, we can formally define a selfish f-srca.

Definition 8.2.5 *Let A be an f-srca. We say that A is* selfish *if for any server-exclusive execution α of A, the following holds: Let π be an action in α initiated by client C_i, $i \in \mathcal{C}$.*

1. *If the last (user) invocation at C_i before π is $read_i$, then $\forall v \in V : m(v, s'_\pi) \leq m(v, s_\pi)$.*

2. *If the last (user) invocation at C_i before π is $write(v)_i$, then $\forall v' \in V \backslash \{v\} : m(v', s'_\pi) \leq m(v', s_\pi)$.*

This definition says that an f-srca is selfish, if any action initiated by a reading client does not increase the multiplicity of any value, and any action initiated by a writing client does not increase the multiplicity of any value besides the value the client is writing. This means that clients don't "help" each other write any value.

A selfish f-srca might be preferable over an unselfish f-srca in some circumstances. Recall from in Chapter 7 in many situations, the time and communication needed to write a value of x is large. Therefore, if we want an f-srca to provide fast reads, we don't want the reads to have to write values of x. Similarly, if we want fast writes, we don't want writes to have to write any value of x other than their own.

Selfish algorithms represent a trade-off between the time and space cost of an f-srca. Indeed, we will show in Theorem 8.2.8 that a selfish f-srca must use storage proportional to the number of concurrent writers. *LDR* is a selfish f-srca. An *LDR* read operation doesn't increase the multiplicity of any value, since, if a value was written at g replicas before a read, then it is written at the same g servers after the write. Similarly, an *LDR* write doesn't increase the multiplicity of any value other than its own. Therefore, by Theorem 8.2.8, the servers (*i.e.*, the directories and replicas) in *LDR* must have storage proportional to the maximum number of concurrent writers. This shows that the fact that replicas store a set of values of x when there are concurrent writes is not a flaw of *LDR*, but (modulo design choices) a necessity.

On the other hand, there exist "unselfish" f-srcas[3] that use an amount of storage independent of the number of concurrent writers. An example of such an algorithm is *ABD*. Also, if we allow writes to write values of x other than their own, but disallow reads from writing values of x[4], then again there exist f-srcas which use storage independent of the number of concurrent writers.

Next, we define the amount of storage that the servers of an f-srca use, and also what it means for the servers to have unbounded storage.

Definition 8.2.6 *Let A be an f-srca. Define $M(A) = \sup_{\alpha \in finexecs(A)}\{\sum_{v \in V} m(v, \alpha)\}$. We say that the servers in A have* storage *s if $M(A) = s < \infty$. If $M(A) = \infty$, we say the servers in A have* unbounded storage.

[3] An f-srca is unselfish if, informally, reads are allowed to write values of x, and writes are allowed to write values of x other than their own.

[4] This might be useful to provide fast response, at the expense of a slower write response.

This defines the storage of the servers as the supremum of the sum of the multiplicities of all values in V, over all finite executions. $M(A)$ is an implementation-independent way to measure the storage used by the servers of A. That is, $M(A)$ corresponds to how much storage is used by servers in A, without explicitly mentioning any data-structures used by the servers. If $M(A) > M(B)$ for two f-srcas A and B, then intuitively, the servers of A can store more information than the servers of B. If $M(A)$ is infinite, then the servers of A must actually have unbounded storage capacity, since they must store an arbitrarily large number of copies of values of x.

Lastly, we define an environment that outputs at most η concurrent writes. This will help us state the relationship between the amount of storage at the servers of an f-srca, and the number of concurrent writers the f-srca allows.

Definition 8.2.7 *Let η be a positive integer. Define $U(A, \eta)$ to be a user for A such that there are at most η write invocations without corresponding responses, in any state of any execution of $A \times U(A, \eta)$. Define $U(A, \infty)$ to be a user for A such that there may be an arbitrary number of write invocations without corresponding responses, in any state of any execution of $A \times U(A, \infty)$.*

8.2.2 Theorem

We now formally state the second lower bound.

Theorem 8.2.8 *Let η and f be two positive integers, and let A be a selfish f-srca for a user $U(A, \eta)$. Then $M(A) \geq f\eta$.*

Proof. This theorem says that if A is a selfish replication algorithm which tolerates up to $f > 0$ server failures, and is guaranteed to be atomic, live and well-formed as long as there are at most $\eta > 0$ concurrent write invocations, then the servers of A must have storage at least as great as $f\eta$.

The proof is by contradiction. Assume that there is an algorithm A that is an f-srca for $U(A, \eta)$ in which the servers have storage less than $f\eta$. Then we will construct an execution α that begins with η concurrent writes. We will ensure that all the values being written are f-erasable. At the same time, we ensure that one of the writes finishes. Then we will extend α with a series of nonoverlapping reads. Since one of the η writes finished, then by the atomicity of A, no read can return the initial value of x. For each of the reads, we will select one of the written values v, and delay all the messages from a set of f servers so as to erase v, from the point of view of the read. Then, the read must return some value other than v. Using this procedure, we can make each read return a different value than the previous read. Thus, if there are $\eta + 1$ reads, one of the reads must return an older value than a preceding read, which violates the atomicity of A. This contradiction shows that A doesn't exist. We now give the details of the proof.

Let A be an f-srca for $U(A, \eta)$, where $\eta, f > 0$, and assume $M(A) < f\eta$. Let W be a set of η writer clients, all writing distinct values different from v_0, and let s_0 be an initial state of A. Consider the following procedure, call it G, for generating an execution of A.

`Procedure` G
$\alpha \leftarrow s_0$
while no $w \in W$ is finished {
  if $\exists w \in W$ with action π enabled, and π is not an invocation at a server
    $\alpha \leftarrow \alpha\pi s'_\pi$
  else, choose a $w \in W$ with invocation π at server S_j enabled, such that the following holds: if we extend α to α', by running π and then letting S_j run until it outputs a response to π, then $\forall v \in V \backslash \{v_0\} : m(v, \alpha') \leq f$
    $\alpha \leftarrow \alpha'$
}

G begins with execution $\alpha = s_0$. Then, as long as no client in W finishes its write, G either lets a client take a step that isn't an invocation at a server, or lets a client invoke an action at a server and then runs that server until it outputs a response to the invocation, so long as doing so doesn't increase the multiplicity of any value beyond f.

Let α be an execution generated by G. We prove some properties about α. We first show that α is a server-exclusive execution of A.

Lemma 8.2.9 *$\alpha \in execs(A)$, and α is server-exclusive.*

Proof. Since α begins in a starting state of A, and G only runs processes with actions enabled, α is a valid execution of A.

Execution α is server-exclusive because anytime a client invokes an action at a server, G runs the server until it responds to the invocation. Thus, only one client accesses a server at a time. □

The next lemma says that the multiplicity of every value, except possibly v_0, is at most f after α.

Lemma 8.2.10 $\forall v \in V \backslash \{v_0\} : m(v, \alpha) \leq f$

Proof. We show that for all prefixes α' of α, we have $\forall v \in V \backslash \{v_0\} : m(v, \alpha') \leq f$. This holds for $\alpha' = s_0$, since all reads starting from s_0 must return v_0, which implies that $\forall v \in V \backslash \{v_0\} : m(v, \alpha') = 0$. Suppose the lemma holds for α', and consider any extension $\alpha'\pi s'_\pi$ generated by G. We claim that if π is not an invocation at a server, then the multiplicity of every value stays the same after π. Indeed, let F be any set of servers, and consider the executions $\alpha \cdot fail_F \cdot s'_{fail_F} \cdot \pi \cdot s'_\pi$ and $\alpha \cdot \pi \cdot s'_\pi \cdot fail_F \cdot s'_{fail_F}$. Since no servers or clients observe $fail_F$, then states s'_π in the first execution looks the same as state s'_{fail_F} in the second execution to every client and server, which implies that the multiplicity of every value stays the same after π. If π is not an invocation at a server, then no server changes its state after π, and the multiplicity of every value remains the same. Thus, the lemma holds for $\alpha'\pi s'_\pi$. If π is an invocation at a server, then by the test in the else statement, the multiplicity of every value except v_0 is at most f after $\alpha'\pi s'_\pi$. □

Lastly, we prove that some writer finishes its write in α.

Lemma 8.2.11 *$\exists w \in W : w$ finishes its write after α.*

Proof. We first show that as long as no writer finishes in α, G can extend α to a longer execution. That is, as long as no writer finishes, either the `if` or the `else` condition of G is true. Thus, we assume the `if` condition is false, and show that the `else` condition is true. We claim that there exists a $w \in W$ writing value v, such that $m(v, \alpha) < f$. Indeed, the sum of the multiplicities of all the values being written by the η clients in W is at most $M(A) < f\eta$, so there must exist some value with multiplicity less than f. We next claim that w must have an action, which is an invocation at a server, enabled. Indeed, w must have *some* action π enabled. This is because, by the liveness guarantee of A, w must eventually complete its write, no matter what the other writers are doing, as long as at most f servers fail. Since there are no failures in α, and w is not waiting for a response from a server, it must have some action enabled.

Now, since the `if` condition is false, π must be an invocation at some server S_j. Let α' be an extension of α in which we run π, then run S_j until it outputs a response to π. We claim that $\forall v \in V \backslash \{v_0\} : m(v, \alpha') \leq f$. Indeed, the only server whose state may differ from the end of α to the end of α' is S_j. This is because π can only change the state of S_j, and any action taken by S_j before outputting a response to w can only change its own state, by the definition of the atomic servers model. Since only S_j can change its state from α to α', the multiplicity of any value can increase by at most 1 between α and α'. Furthermore, since A is selfish and α' is a server-exclusive execution of A, then only the multiplicity of w's value v can increase from α to α'. Since $m(v, \alpha) < f$, we have $m(v, \alpha') \leq f$, and $\forall v' \in V \backslash \{v_0\} : m(v', \alpha') \leq f$. Therefore, if the `if` condition of G is false, then the `else` condition of G is true. Thus, as long as no writer finishes, G can extend α to a longer execution.

By the above argument, as long as no writer finishes, α can grow arbitrarily long. A's liveness guarantee states that every write must eventually finish if there are at most f server failures. Then, since there are no server failures in α, at some point α will grow long enough for some writer $w \in W$ to finish. $\square$

We fix an α generated by G in which a writer w finishes writing value v. Observe that any read ϕ starting in any extension of α must return a value different from v_0.[5], since the write of v has finished, and must be linearized before the start of ϕ. We now define a set of read extensions of α, generated by the following procedure:

1. Choose a complete read operation ϕ_0 such that $\alpha \sqsubseteq_j^R \alpha\phi_0$. Set $\alpha_1 = \alpha\phi_0$. Go to step 2.

2. For $i > 0$, let $v_{i-1} = \chi(\phi_{i-1})$. Choose $F_{i-1} \subseteq \mathcal{S}$, $|F_{i-1}| = f$, such that there is no read extension of $\alpha_{i-1} \cdot fail_{F_{i-1}}$ returning v_{i-1}. Go to step 3.

[5]Again, unless another writer writes v_0 after α. However, we will only append reads to α from now on.

3. Choose a complete read operation ϕ_i by C_j such that $\alpha_{i-1} \sqsubseteq_j^R \alpha_{i-1}\phi_i$, and such that $\Gamma(\phi_i) \cap F_{i-1} = \emptyset$. Set $\alpha_i = \alpha_{i-1}\phi_i$. Go back to step 2.

This procedure creates a set of extensions $\{\alpha_i\}_i$ of α, such that for all i, α_i equals α_{i-1} extended by a complete read operation ϕ_i. Note that every α_i is server-exclusive, since every read ϕ_i runs in isolation and has exclusive access to any server. For any i, v_i is the value returned by ϕ_i. F_i is a set of servers such that if these servers fail, then no read extension of α_i returns v_i. We argue why F_i exists. Indeed, after α, the multiplicity of every value except possibly v_0 is at most f. Since A is selfish and every α_i is server-exclusive, then the multiplicity of any value does not increase after ϕ_i, for any i. Therefore, the multiplicity of every value except possibly v_0 is at most f after α_i, and by the definition of multiplicity, there exists a set of at most f servers whose failure erases v from α_i.

Now, given v_{i-1}, the next read ϕ_i is chosen so that it does not read from any server in F_{i-1}.[6] Read ϕ_i exists because A must tolerate the failure of any f servers, so that by delaying all the responses from servers in F_{i-1} indefinitely during ϕ_i, we can ensure that ϕ_i completes without reading from any server in F_{i-1}.

Note that we can execute the above procedure an arbitrary number of times, and generate an arbitrarily large number of ϕ_i's. We now prove some properties about the reads $\{\phi_i\}_i$. We first show that any two consecutive reads return different values.

Lemma 8.2.12 $\forall i : v_i \neq v_{i-1}$.

Proof. Consider the execution extending α_{i-1} in which the servers in F_{i-1} fail following α_{i-1}. Then by the definition of F_{i-1}, there is no read extension of that execution returning v_{i-1}. But since ϕ_i doesn't communicate with any replica in F_{i-1}, it seems to ϕ_i that the replicas in F_{i-1} have failed. Then, since all the values written by processes in P are different, ϕ_i must return $v_i \neq v_{i-1}$. □

Corollary 8.2.13 $\exists i, j > 0 : (j - i > 1) \wedge (v_i = v_j)$.

Proof. Each ϕ_i must return one of the at most η values written during α. Since there are an arbitrary number of ϕ_i's, there must exist i and j such that ϕ_i and ϕ_j return the same value, *i.e.*, $v_i = v_j$. By Lemma 8.2.12, we must have $|i - j| > 1$. □

Now we can finally finish the proof of Theorem 8.2.8. Choose i and j as in Corollary 8.2.13, and choose k such that $i < k < j$. In any linearization of α_j, the write of v_i precedes the write of v_k which precedes the write of v_j. However, we have $v_i = v_j$, so that the write of v_i is the same as the write of v_j. This is a contradiction, and shows that A does not exist. Thus, for any f-srca which works correctly when there are η concurrent writers, the servers must have storage no less than $f\eta$. □

[6] Note that we do not actually fail the servers in F_{i-1}, but only make sure that ϕ_i doesn't communicate with them.

Chapter 9

Conclusions and Future Work

In this thesis, we studied problems related to efficient data replication. We presented *LDR*, a nearly optimal algorithm for replicating large data objects. *LDR* tolerates an arbitrary number of replica server failures, up to the total number of replicas. *LDR* makes minimal assumptions about its environment. It does not rely on distributed locking or group communication, and works in any asynchronous, reliable message-passing network. *LDR* tolerates high latency, and is suitable for implementation in both WAN and LAN settings. In addition to describing the *LDR* algorithm, we formally specified its assumptions and guarantees. We also formally implemented *LDR* in the IOA language, and provided correctness proofs and performance analyses of our implementation. Lastly, we presented two lower bounds on the costs of data replication. The motivation for these lower bounds were certain algorithmic techniques we used in the design of *LDR*. Our lower bounds suggest that these techniques were necessary.

Our work can be extended along several directions. For example, we mentioned in Chapter 5 that it is possible to combine the rdw and rrr phases of a read operation, and that a write can return as soon as it writes to a quorum of directories in phase wdw, before it has sent out the secure messages. These optimizations improve the performance of *LDR*. Also, the performance of *LDR* only meets our lower bounds approximately, and it would be interesting to bridge the gaps. For example, Theorem 8.1.5 states that any read must write to at least f servers, while *LDR* writes to $f + 1$ servers during a read. Also, Theorem 8.2.8 states that servers need at least ηf storage when there are η concurrent writes. *LDR* uses more storage than this if it does not perform prompt garbage-collection, or if *secure* messages are delayed in the network. It may be possible to modify *LDR* to meet the storage bound exactly. Another improvement to *LDR* is to optimize the placement of replicas, *e.g.*, depending on data access patterns. Indeed, since *LDR* stores the data at arbitrary sets of ($\geq f + 1$) replicas, instead of quorums of replicas, we can consider facility-location type algorithms to distribute data to replicas which can service requests with the least cost.

LDR is able to efficiently replicate large data objects because it separately maintains data from metadata, and performs mostly cheap operations on the metadata in order to avoid expensive operations on the data. It seems likely that such a separation can be applied in other distributed algorithms to yield improved performance. The separation technique can also be viewed as a procedure to minimize the amount of synchronization within a distributed algorithm. For example, in the case of data replication that we considered, it suffices for different clients to synchronize with each other using the tags on their data. The main work of the algorithm, writing values of x, can be done on "shadow copies" (*i.e.*, copies local to each client's operation) at the replicas, without any synchronization. It is only at the end of each client's operation that it synchronizes with the others by performing some cheap writes of tags at the directories. We may contrast this with an algorithm like ABD, which performs expensive synchronization by having clients help perform the main work (writing values of x) of each other's operation. That is, ABD synchronizes clients using the data instead of tags on the data, and is therefore inefficient for replicating large data objects. In the case of data replication, it was easy to see that it is cheaper to synchronize on the tags instead of the data. But for more complex distributed algorithms, it would be interesting to develop a theory of how to determine the cheapest way for processes to synchronize with each other. Indeed, every distributed algorithm consists of a "local" part in which a participating process does not need to synchronize with other processes, and a "global" part which requires synchronization among the processes. By minimizing the global part of the algorithm, we can minimize the amount of communication, which is often the most expensive part of a distributed computation, and thereby enhance the performance of the algorithm.

Bibliography

[1] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd international conference on Software engineering*, pages 562–570. IEEE Computer Society Press, 1976.

[2] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995.

[3] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley Longman Publishing Co., Inc., 1987.

[4] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46+, 2001.

[5] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh symposium on Operating systems principles*, pages 150–162, 1979.

[6] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems (TOCS)*, 4(1):32–53, 1986.

[7] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, 1992.

[8] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, March 1996.

[9] Nancy Lynch and Alex Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 272–281, Seattle, Washington, USA, June 1997. IEEE.

[10] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 569–578. ACM Press, 1997.

[11] M. Tamer Ozsu and P. Valduriez. *Principles of distributed database systems.* Prentice-Hall, Inc., 1991.

[12] J.-F. Paris. Voting with witnesses:: A consistency scheme for replicated files. In *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS)*, pages 606–612, Washington, DC, 1986. IEEE Computer Society.

[13] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182. ACM Press, 1995.

[14] R. van Renesse and A. S. Tanenbaum. Voting with ghosts. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*, pages 456–462, Washington, DC, 1988. IEEE Computer Society.

[15] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Morgan Kaufmann Publishers Inc., 2001.