Efficient Replication of Large Data Objects

Rui Fan and Nancy Lynch

MIT Computer Science and Artificial Intelligence Laboratory,
200 Technology Square,
Cambridge, MA USA 02139
{rfan,lynch}@theory.lcs.mit.edu

Abstract. We present a new distributed data replication algorithm tailored especially for large-scale read/write data objects such as files. The algorithm guarantees atomic data consistency, while incurring low latency costs. The key idea of the algorithm is to maintain copies of the data objects separately from information about the locations of up-to-date copies. Because it performs most of its work using only the location information, our algorithm needs to access only a few copies of the actual data; specifically, only one copy during a read and only $f+1$ copies during a write, where f is an assumed upper bound on the number of copies that can fail. These bounds are optimal. The algorithm works in an asynchronous message-passing environment. It does not use additional mechanisms such as group communication or distributed locking. It is suitable for implementation in WANs as well as LANs. We also present two lower bounds on the costs of data replication. The first lower bound is on the number of low-level writes required during a read operation on the data. The second bound is on the minimum space complexity of a class of efficient replication algorithms. These lower bounds suggest that some of the techniques used in our algorithm are necessary. They are also of independent interest.

1 Introduction

Data replication is an important technique for improving the reliability and scalability of data services. To be most useful, data replication should be transparent to the user. Thus, while there exist multiple physical copies of the data, users should only see one logical copy, and user operations should appear to execute atomically on the logical copy.

To maintain atomicity, existing replication algorithms typically use locks [4], embed physical writes to the data within a logical read [3,14], or assume powerful network primitives such as group communication [2]. However, such techniques have adverse effects on performance [8], and practical systems either sacrifice their consistency guarantees [12], or rely on master copies [15] or use very few replicas.

This paper presents an algorithm which deals with the performance penalty of data replication by taking advantage of the fact that, in a typical application requiring replication, such as a file system, the size of the objects being replicated

F.E. Fich (Ed.): DISC 2003, LNCS 2848, pp. 75–91, 2003.

is often much larger than the size of the metadata (such as tags or pointers) used by the algorithm. In this situation, it is efficient to perform more cheap operations on the metadata in order to avoid expensive operations on the data itself.

Our algorithm replicates a single data item supporting read and write operations, and guarantees that the operations appear to happen atomically. The normal case[1] communication cost is nearly constant for a read operation, and nearly linear in f for a write, where f is an upper bound on the number of replica failures. The latency for a read and write are both nearly constant. Here, we measure the communication and latency costs in terms of the number of data items accessed, and ignore the number of metadata items accessed, as the former term is dominant.[2] Our algorithm runs on top of any reliable, asynchronous message passing network. It tolerates high latency and network instability, and therefore is appropriate in both LAN and WAN settings.

The basic idea of the algorithm is to separately store copies of the data in *replica servers*, and information about where the most up-to-date copies are located in *directory servers*. We call this layered replication approach *Layered Data Replication (LDR)*. Roughly speaking, to read the data, a client first reads the directories to find the set of up-to-date replicas, then reads the data from one of the replicas. To write, a client first writes its data to a set of replicas, then informs the directories that these replicas are now up-to-date.

In addition to our replication algorithm, we prove two lower bounds on the costs of replication. The first lower bound shows that in any atomically consistent replication algorithm, clients must sometimes write to at least f replicas during a logical read, where f is the number of replicas that can fail. The second lower bound shows that for a *selfish* atomic replication algorithm, *i.e.*, one in which clients do not "help" each other, the replicas need to have memory which is proportional to the maximum number of clients that can concurrently write. In addition to their independent interest, these lower bounds help explain some of the techniques *LDR* uses, such as writing to the directories during a read, and sometimes storing multiple copies of the data in a replica.

Our paper is organized as follows. Section 2 describes related work on data replication. Section 3 formally defines our model and problem. Section 4 describes the *LDR* algorithm, while Sections 5 and 6 prove its correctness, and analyzes and compares its performance to other replication algorithms. Section 7 presents our lower bounds. Finally, Section 8 concludes.

2 Related Work

There has been extensive work on database replication [4,15,5,2]. Algorithms that guarantee strong consistency usually rely on locking and commit protocols [4]. Practical systems usually sacrifice consistency for performance [12], or rely on master copies [5] or group communication [2]. In our work, we do not consider

[1] *I.e.,* when there are no failures.
[2] The amount of metadata accessed is also not large.

transactions, only individual read/write operations on a single object. Therefore, we can avoid the use of locks, commit protocols and group communication, while still guaranteeing strong consistency.

Directory-based replication is also used in file systems, such as Farsite [1]. However, this system focuses more on tolerating Byzantine failures and providing file-system semantics, and their replication algorithm and analysis is less formal and precise than ours.

Our use of directory servers bears similarities to the *witness replicas* of [16] and the *ghost replicas* of [18]. These replicas store only the version number of the latest write, and are cheap to access. We extend these ideas by storing the locations of up-to-date replicas in directories, allowing *LDR* to access the minimum copies of the actual data. In addition, since our replicas are not used in voting, we can replicate the data in arbitrary sets of (at least $f + 1$) replicas, instead of only in quorums. This allows optimizations on replica placement, which can further enhance *LDR*'s performance. Lastly, while [16] and [18] still need external concurrency control mechanisms, *LDR* does not.

Directory-based cache-coherence protocols [17] are used in distributed shared memory systems, and are in spirit similar to our work. However, these algorithms are not directly comparable to *LDR*, since the assumptions and requirements in the shared memory setting are quite different from ours.

The algorithm used to maintain consistency among the directories is based on the weighted voting algorithm of [7] and the shared memory emulation algorithms of [3] and [14]. One can apply [3] and [14] directly for data replication. However, doing so is expensive, because these algorithms read and write the data to a quorum of replicas in each client read or write operation. The client read operation is especially slow compared to *LDR*, since *LDR* only accesses the data from one replica during a read.

Theorem 10.4 of [9] is similar in spirit to our first bound. It shows that in a wait-free simulation of a single-writer, multi-reader register using single-writer, single-reader registers, a reader must sometimes write. In contrast, our lower bound considers arbitrary processes simulating a multi-reader, multi-writer register, and shows that a reader must sometimes write to at least f processes, where f is the number of processes allowed to fail.

3 Specification

3.1 Model

Let x be the data object we are replicating. x takes values in a set V, and has default value v_0. x can be read and written to. To replicate multiple objects, we run multiple instances of *LDR*. However, we do not support transactions, *i.e.*, single operations that both read and write, or access multiple objects.

LDR is based on the client-server model. Each client and server is modeled by an I/O automaton [13]. The communication between clients and servers is modeled by a standard reliable, FIFO asynchronous network.

3.2 Interface, Assumptions, and Guarantees

The clients receive external input actions to read and write to x. Upon receiving an input, a client interacts with the servers to perform the requested action. To distinguish the input actions that read/write x from the low-level reads and writes which clients perform on the servers, we sometimes call the former *logical* or *client* reads/writes, and the latter *physical* reads/writes.

Let $\mathcal{C}$ be the set of *client endpoints*. For every $i \in \mathcal{C}$, client i has *invocations* (input actions) $read_i$ (resp., $write(v)_i$) to read (resp., write v to) x, and *corresponding responses* (output actions) $read-ok(*)_i$ (resp., $write-ok_i$). The servers are divided into $\mathcal{R}$, the *replica endpoints*, and $\mathcal{D}$, the *directory endpoints*. We assume that $\mathcal{R}$ and $\mathcal{D}$ are finite ($\mathcal{C}$ may be infinite). The interface at a server $i \in \mathcal{D} \cup \mathcal{R}$ consists of $send(m)_{i,j}$ to send message m to endpoint j, $j \in \mathcal{C} \cup \mathcal{D} \cup \mathcal{R}$, and $recv(m)_{j,i}$ to receive m from j.

We assume the crash-fail model, and we model failures by having a $fail_i$ input for every $i \in \mathcal{C} \cup \mathcal{D} \cup \mathcal{R}$. When $fail_i$ occurs, automaton i stops taking any more locally controlled steps.

LDR assumes clients are *well-behaved*. That is, clients do not make consecutive invocations without a receiving a corresponding response in between.

LDR's guarantees are specified by properties of its traces. *LDR*'s liveness guarantee is conditional. Specifically, let $(\mathcal{Q}_R, \mathcal{Q}_W)$ be a *read/write quorum system* over $\mathcal{D}$. That is, $\mathcal{Q}_R, \mathcal{Q}_W \subseteq 2^{\mathcal{D}}$ are collections of subsets of $\mathcal{D}$, with the property that for any sets $Q_1 \in \mathcal{Q}_R$ and $Q_2 \in \mathcal{Q}_W$, $Q_1 \cap Q_2 \neq \emptyset$. Also, let f be any natural number such that $f < |\mathcal{R}|$. Then *LDR* guarantees the following:

Definition 1. *(Liveness) In any infinite trace of* LDR *in which at most f replicas fail, and some $Q_1 \in \mathcal{Q}_R$ and $Q_2 \in \mathcal{Q}_W$ of directories do not fail, every invocation at a nonfailing client has a subsequent corresponding response at the client.*

LDR's safety guarantee says that client read/write operations appear to execute atomically.

Definition 2. *(Atomicity) Every trace of* LDR*, when projected onto the client invocations and corresponding responses, can be linearized to a trace respecting the semantics of a read/write register with domain V and initial value v_0.*

We refer to [10] for a formal definition of atomicity and linearization.

4 The *LDR* Algorithm

The clients, replicas and directories have different state variables and run different protocols. The protocols are shown in Figures 1, 2, and 3, resp., and are described below. Figures 4 and 5 show the schematics of the read and write operations, resp. Both the read and write operations involve the client getting an external input, then contacting some directories and replicas to perform the requested action.

4.1 State

A client has the following state variables. Variable *phase* (initially equal to *idle*) keeps track of where a client is in a read/write operation. Variable $utd \in 2^{\mathcal{R}}$ (initially $\emptyset$) stores the set of replicas which the client thinks are most up-to-date. Variable $tag \in \mathbb{N} \times \mathcal{C}$ (initially t_0[3]) is the tag of the latest value of x the client knows. Variable *mid* (initially 0) keeps track of the latest message the client sent; the client ignores responses with $id < mid$.

A replica has one state variable $data \subseteq V \times T \times \{0, 1\}$, initially $\emptyset$. For each triple in *data*, the first coordinate is a value of x that the replica is storing. The replica may store multiple values of x; the reason why this is done is explained in Section 7.3. The second coordinate is the tag associated with the value. The third coordinate indicates whether the value is *secured*, as explained in Section 4.3.

A directory has a $utd \subseteq \mathcal{R}$ variable, initially equal to $\mathcal{R}$, which stores the set of replicas that have the latest value of x. It also has a variable $tag \in T$, initially t_0, which is the tag associated with that value of x.

4.2 Client Protocol

When client i does a read, it goes through four phases in order: rdr, rdw, rrr and rok.[4] During rdr, i reads (utd, tag) from a quorum of directories to find the most up-to-date replicas. i sets its own *tag* and *utd* to be the (tag, utd) it read with the highest *tag*. During rdw, i writes (utd, tag) to a write quorum of directories, so that later reads will read i's *tag* or higher. During rrr, i reads the value of x from a replica in *utd*. Since each replica may store several values of x, i tells the replica it wants to read the value of x associated with *tag*. During rok, i returns the x-value it read in rrr.

When i writes a value v, it also goes through four phases in order: wdr, wrw, wdw and wok.[5] During wdr, i reads (utd, tag) from a quorum of directories, then sets its *tag* to be higher than the largest *tag* it read. During wrw, i writes (v, tag) to a set *acc* of replicas, where $|acc| \geq f+1$. Note that the set *acc* is arbitrary; it does not have to be a quorum. During wdw, i writes (acc, tag) to a quorum of directories, to indicate that *acc* is the set of most up-to-date replicas, and *tag* is the highest tag for x. Then i sends each replica a *secure* message to tell them that its write is finished, so that the replicas can garbage-collect older values of x. Then i finishes in phase *wok*.

---

[3] Here $t_0 < t, \forall t \in T$, where tags are ordered lexicographically, and T denotes the set of all tags.

[4] The phase names describe what happens during the phase. They stand for *read-directories-read*, *read-directories-write*, *read-replicas-read*, and *read-ok*, resp.

[5] As for a read, *wdr* stands for *write-directories-read*, *wrw* for *write-replicas-write*, *wdw* for *write-directories-write*, and *wok* for *write-ok*.

```
input read_i
Effect:
    mid ← mid + 1
    for all j ∈ D do msg[j] ← ⟨read, mid⟩
    phase ← rdr

input write(v)_i
Effect:
    val ← v; mid ← mid + 1
    for all j ∈ D do msg[j] ← ⟨read, mid⟩
    phase ← wdr

input fail_i
Effect:
    stop taking locally-controlled steps

output read-ok(v)_i
Precondition:
    (val = v) ∧ (phase = rok)
Effect:
    phase ← idle

output write-ok_i
Precondition:
    phase = wok
Effect:
    phase ← idle

output send(m)_{i,j}
Precondition:
    msg[j] = m
Effect:
    msg[j] ←⊥

input recv(m)_{j,i} where (m = ⟨read-ok, S, t, id⟩)
Effect:
    if (phase = rdr) ∧ (id = mid) then
      acc ← acc ∪ {j}
      if (t > tag) then
        tag ← t; utd ← S
      if (∃Q ∈ Q_R : Q ⊆ acc) then
        mid ← mid + 1
        for all j ∈ D do msg[j] ← ⟨write, utd, tag, mid⟩
        acc ← ∅; phase ← rdw

input recv(m)_{j,i} where (m = ⟨write-ok, id⟩)
Effect:
    if (phase = rdw) ∧ (id = mid) then
      acc ← acc ∪ {j}
      if (∃Q ∈ Q_W : Q ⊆ acc) then
        mid ← mid + 1
        for all j ∈ utd do msg[j] ← ⟨read, tag, mid⟩
        acc ← ∅; phase ← rrr

input recv(m)_{j,i} where (m = ⟨read-ok, v, t, id⟩)
Effect:
    if (phase = rrr) ∧ (id = mid) then
      val ← v; tag ← t; phase ← rok

input recv(m)_{j,i} where (m = ⟨read-ok, t, id⟩)
Effect:
    if (phase = wdr) ∧ (id = mid) then
      acc ← acc ∪ {j}
      if (t > tag) then
        tag ← t              //tag = (n, i′)
      if (∃Q ∈ Q_R : Q ⊆ acc) then
        mid ← mid + 1; tag ← (n + 1, i)
        for all j ∈ R do msg[j] ← ⟨write, val, tag, mid⟩
        acc ← ∅; phase ← wrw

input recv(m)_{j,i} where (m = ⟨write-ok, id⟩)
Effect:
    if (phase = wrw) ∧ (id = mid) then
      acc ← acc ∪ {j}
      if (|acc| > f) then
        mid ← mid + 1
        for all j ∈ D do msg[j] ← ⟨write, acc, tag, mid⟩
        acc ← ∅; phase ← wdw
    else if (phase = wdw) ∧ (id = mid) then
      acc ← acc ∪ {j}
      if (∃Q ∈ Q_W : Q ⊆ acc) then
        mid ← mid + 1
        for all j ∈ R do msg[j] ← ⟨secure, tag, mid⟩
        acc ← ∅; phase ← wok
```

Fig. 1. Client C_i transitions.

```
input recv(m)_{j,i} where (m = ⟨read, t, mid⟩)
Effect:
    if ∃v : (v, t, ∗) ∈ data then
      (v′, t′) ← choose {v | (v, t, ∗) ∈ data}
      msg[j] ← ⟨read-ok, v′, t′, mid⟩
    else
      (v′, t′) ← maxst(data)
      msg[j] ← ⟨read-ok, v′, t′, mid⟩

input recv(m)_{j,i} where (m = ⟨write, v, t, mid⟩)
Effect:
    data ← data ∪ {(v, t, 0)}
    msg[j] ← ⟨write-ok, mid⟩

input recv(m)_{j,i} where (m = ⟨gossip, v, t⟩)
Effect:
    data ← data ∪ {(v, t, 1)}\{(v, t, 0)}
    for all j ∈ D do
      msg[j] ← ⟨write, {i}, t⟩

input recv(m)_{j,i} where (m = ⟨secure, t, mid⟩)
Effect:
    if ∃v : (v, t, 0) ∈ data then
      for all v : (v, t, 0) ∈ data do
        data = data ∪ (v, t, 1)\{(v, t, 0)}

input fail_i
Effect:
    stop taking locally-controlled steps

output send(m)_{i,j}
Precondition:
    msg[j] = m
Effect:
    msg[j] ←⊥

internal gossip_i
Precondition:
    ∃v, t, : (v, t, 1) ∈ data
Effect:
    (v′, t′) ← choose {(v, t) | (v, t, 1) ∈ data}
    for all j ∈ R do
      msg[j] ← ⟨gossip, v′, t′⟩

internal gc_i
Precondition:
    ∃v, t : (v, t, 1) ∈ data
Effect:
    t ← choose {t′ | (v, t′, 1) ∈ data}
    for all v′, t′ : ((v′, t′, ∗) ∈ data) ∧ (t′ < t) do
      remove (v′, t′, ∗) from data
```

Fig. 2. Replica R_i transitions.

```
input recv(m)_{j,i} where (m = ⟨read, mid⟩)
Effect:
    msg[j] ← ⟨read-ok, utd, tag, mid⟩

input fail_i
Effect:
    stop taking locally-controlled steps

output send(m)_{i,j}
Precondition:
    msg[j] = m
Effect:
    msg[j] ←⊥

input recv(m)_{j,i} where (m = ⟨write, S, t, mid⟩)
Effect:
    if (t = tag) then
      utd ← utd ∪ S
    else if (t > tag) then
      if |S| ≥ f + 1 then
        utd ← S
        t ← tag
    msg[j] ← ⟨write-ok, mid⟩
```

Fig. 3. Directory D_i transitions.

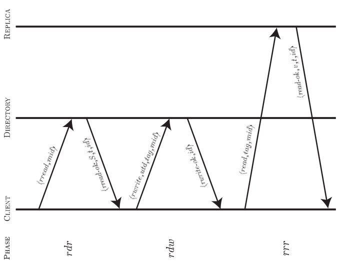

Fig. 4. Client read operation.

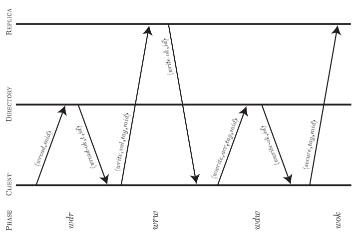

Fig. 5. Client write operation.

4.3 Replica Protocol

The replicas respond to client requests to read and write values of x. Replicas also garbage-collect out of date values of x from *data*, and gossip among themselves the latest value of x. The latter is an optimization to help spread the latest value of x, so that clients can read from a nearby replica.

When replica i receives a message to write value/tag (v, t), i just adds $(v, t, 0)$ to *data*. The 0 in the third coordinate indicates v is not a secured value. When i is asked to read the value associated with tag t, i checks whether it has $(v, t, *)$[6] in *data*. If so, i returns (v, t). Otherwise, i finds the secured triple with the largest tag in *data*, *i.e.*, the $(v', t', 1)$ with the highest tag t' among all triples with third coordinate equal to 1, and returns (v', t'). When i is asked to secure tag t, i checks whether $(*, t, 0)$ exists in *data*, and if so, sets the third coordinate of the triple to 1.

When i garbage-collects out of date values of x, it finds a secured value $(v, t, 1)$ in *data*, and then removes all triples $(v', t', *)$ with $t' < t$ from *data*.

[6] The $*$ indicates the last coordinate can be a 0 or 1.

When i gossips, it finds a secured value $(v, t, 1)$ in *data*, and sends (v, t) to all the other replicas. When i receives a gossip message for (v, t), it adds $(v, t, 1)$ to *data*.

4.4 Directory Protocol

The directories' only job is to respond to client requests to read and write *utd* and *tag*.

When directory i gets a message to read *utd* and *tag*, it simply returns (utd, tag). When i is asked to write (S, t) to *utd* and *tag* (S is a set of replicas and t is a tag), i first checks that $t \geq tag$. If not, then the write request is out of date, and i sends an acknowledgment but does not perform the write. If $t = tag$, i adds S to *utd*. If $t > tag$, i checks whether $|S| \geq f + 1$, and if so, sets *utd* to S.

5 Correctness

In this section, we show that *LDR* satisfies the liveness and atomicity properties of Defns. 1 and 2, resp. A more detailed proof can be found in the full paper [6].

5.1 Liveness

Consider an execution in which some read and write quorum of directories do not fail, and no more than f replicas fail. Then a client never blocks waiting for a response from a quorum of directories. The client also does not block waiting to read from a set *utd* of replicas, since we can easily check that $|utd| \geq f + 1$ always. Therefore, every invocation at a nonfailing client always has a response in the execution.

5.2 Atomicity

To prove the atomicity condition, we show that a trace of *LDR* satisfies Lemma 13.10 of [13]. In [13], it is shown that an algorithm satisfying Lemma 13.10 implements the semantics of an atomic register. The lemma requires us to define a partial order $\prec$ on the operations in a trace of *LDR*. Let ϕ be a complete operation in a trace, *i.e.*, an invocation and its corresponding response. If ϕ is a read, define $\lambda(\phi)$ to be the tag associated with the value returned by ϕ.[7] If ϕ is a write, define $\lambda(\phi)$ to be the tag of the value written by ϕ. We define $\prec$ as follows:

Definition 3. *Let ϕ and ψ be two complete operations in an execution of* LDR.

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 $\lambda(\phi) < \lambda(\psi)$.*

[7] Recall that when ϕ reads from a replica in phase *rrr*, the replica returns $(*, t)$. Then, we set $\lambda(\phi) = t$.

Before proving *LDR* satisfies Lemma 13.10, we first prove some lemmas. The first lemma says that if a client i asks to read a value with tag t from a replica, then the replica returns a value with tag $\geq t$ to the client.

Lemma 1. *Let ϕ be a complete read operation by client i, and let t be the maximum tag which i read during the rdr phase of ϕ. Then, $\lambda(\phi) \geq t$.*

We briefly argue why this lemma is true. Suppose i read (S, t) from a directory during rdr. Then S is the set of replicas that i asks to read from during rrr. For every replica in S, either $(*, t, *)$ still exists in the replica's *data*, or it was garbage-collected. In the first case, the replica returns $(*, t)$, so $\lambda(\phi) = t$. In the second case, the replica must have secured a value with tag $t' > t$ in *data*. The replica returns $(*, t')$, so $\lambda(\phi) > t$.

The next lemma states that after a read finishes, a write quorum of directories have *tag* at least as high as the tag of the value the read returned.

Lemma 2. *Let ϕ be a complete read operation in an execution of* LDR*. Then after ϕ finishes, there exists a write quorum of directories with tag $\geq \lambda(\phi)$.*

We argue why the lemma holds. Let t be the largest tag i read during the rdr phase of ϕ. If $\lambda(\phi) = t$, then i writes $(*, t)$ to a write quorum of directories during rdw, before the end of ϕ, and the lemma is true. Otherwise, by the previous lemma, $\lambda(\phi) > t$. This means i tried to read a value with tag t at a replica, but the replica returned a value with a larger tag. Hence, the latter value was secure at the replica, which implies an earlier client had finished its phase wdw while writing that value. That client wrote $\lambda(\phi)$ to a write quorum of directories during its phase wdw, before the end of ϕ, and so the lemma holds.

We can now prove the relation $\prec$ we defined earlier satisfies Lemma 13.10 of [13]. For lack of space, we prove only the most interesting condition in the lemma, the second. The condition is that if an operation ϕ completes before operation ψ begins, then $\psi \not\prec \phi$.

To see this, we consider the four cases where ϕ and ψ are various combinations of reads or writes. If ϕ and ψ are both writes, then ϕ writes $\lambda(\phi)$ to a write quorum of directories before it finishes. Since the read quorum ψ reads from intersects with ϕ's write quorum, ψ will use a larger tag than ϕ, and $\psi \not\prec \phi$. If ϕ is a write and ψ is a read, then by similar reasoning, ψ returns a value with tag at least as large as $\lambda(\phi)$, and the condition again holds. When ϕ is a read and ψ is a write, by Lemma 2, a write quorum of directories have *tag* at least as high as $\lambda(\phi)$ after ϕ finishes, so ψ uses a larger tag than $\lambda(\phi)$, and the condition holds. Lastly, when both ϕ and ψ are reads, then ψ will try to read a value with tag at least as high as $\lambda(\phi)$ from the replicas. By Lemma 1, $\lambda(\psi) \geq \lambda(\phi)$, and so $\psi \not\prec \phi$.

Combining Sections 5.1 and 5.2, we have shown:

Theorem 1. LDR *satisfies the liveness and atomicity properties of Definitions 1 and 2, resp.*

6 Performance Analysis

We analyze the communication and time complexity of *LDR*, and show that these costs are nearly optimal when the size of the data is large compared to the size of the metadata.

We first describe a modification to the client algorithm. Currently, when a client wants to contact a set of directories or replicas, it sends messages to a superset of that set, in case some directories or replicas have failed. However, in practice failures are rare, and so it suffices for the client to send messages to exactly those directories or replicas it wants to contact. This technique greatly improves performance, and in general, does not decrease fault-tolerance. We analyze *LDR* for this optimized implementation.

6.1 Communication Complexity

A basic assumption which *LDR* makes is that the size of the data, *i.e.*, values of x, is much larger than the size of metadata *LDR* uses, such as tags and *utd*'s. Therefore, we also assume it is much more costly to transfer data than metadata. In particular, we assume that the communication cost to transfer one value of x is d, while the cost to transfer one unit of metadata is 1. We assume $d \gg 1$, and also that $d \gg f^2$, where f is the number of replica failures *LDR* tolerates.[8] Lastly, we assume all read and write quorums have size $f + 1$. As an example of our cost measure, it costs $d + 3$ to transfer the message $\langle read - ok, v, t, id \rangle$, where v is a value of x. With this measure, the communication cost of an *LDR* read operation adds up to $d + 2f^2 + 14f + 18$, and the cost of an *LDR* write operation adds up to $(f + 1)d + f^2 + 20f + 19$.

When $d \gg 1$ and $d \gg f^2$, the cost of an *LDR* read is dominated by the d term. However, any replication algorithm must read at least one value of the data during a read. Therefore, the communication complexity of a read for any replication algorithm is $\geq d$ in the worst case. Therefore, for large d, the communication complexity of an *LDR* read is asymptotically optimal. Also, in any replication algorithm tolerating the failure of up to f replicas, the data must be written to at least $f + 1$ replicas. Therefore, the worst case communication complexity of a write for any replication algorithm is $\geq (f + 1)d$[9]. Therefore, *LDR* also has asymptotically optimal write communication complexity.

6.2 Time Complexity

To evaluate the time complexity, we now consider a synchronous communication model. Similar to the communication complexity, we assume that it takes time

[8] This assumption is reasonable, since in practice f is quite small, typically < 4.

[9] In fact, it is not necessary to write a complete copy of the data to each server. For example, by encoding the data, one can write smaller chunks of the encoding to each server, decreasing the total amount of communication. However, as any such optimizations can also be applied to *LDR*, they do not change the optimality of *LDR*'s communication complexity.

d to transfer a value of x, and it takes unit time to transfer a piece of metadata. We also assume that when we send messages to multiple destinations, we can send them in parallel, so that the time required to send all the messages equals the time to send the largest message. Then, the time complexity of an *LDR* read sums to $d + 2f + 18$, and that of a write sums to $d + f + 19$. Any replication algorithm must take at least d time for a read or write, since it has to read or write at least one copy of the data. Thus, for d large, the time complexity of *LDR* is optimal.

6.3 Comparison to Other Algorithms

We now compare *LDR*'s performance with the performance of the algorithm given in [14]. We choose this comparison because [14] has many attributes in common with *LDR*, such as not using locks or group communication. Most other replication algorithms rely on these techniques, which makes comparison to them difficult. *LDR* and [14] are also similar in methodology. In fact, *LDR* uses a modified form of [14] in its directory protocol. However, the two algorithms differ substantially in their performance. Using the measure for communication cost and latency given above, we compute [14]'s read communication cost as $2(f+1)d$ plus "lower order"(compared to d) terms. For large d, this is a factor of $2(f+1)$ larger than *LDR*'s read communication cost. The write communication cost for [14] is $(f+1)d$ plus lower order terms, which is asymptotically the same as *LDR*'s cost. The latency of a read in [14] is approximately $2d$, which is asymptotically twice that of *LDR*. The latency of a write is asymptotically the same in [14] and *LDR*. We note that most replication algorithms have costs similar to that of [14], so that for large d, *LDR* also performs better than those algorithms.

Lastly, we mention that because *LDR* does not store data in quorums of replicas, but rather, in arbitrary sets, *LDR* can take advantage of algorithms which optimize replica placement to further improve performance.

7 Lower Bounds

7.1 Model

We prove our lower bounds in the *atomic servers* model. This computational model is based on the standard client/server model, except that the servers are required to be atomic objects (of arbitrary and possibly different types), which permit concurrent accesses by clients. Each server j's interface consists of $read(-ok)_{i,j}$ and $modify(-ok)_{i,j}$ actions, $\forall i \in \mathcal{C}$. *read* can return any value based on the server's state, but must not change the server's state. *modify* can change the state of the server arbitrarily, and return any value. The clients have input and output actions corresponding to the outputs and inputs, resp., of the servers. Clients and servers communicate by invoking actions and receiving responses from each other, instead of sending messages.

Let f be a natural number. We say an *f-srca*[10] is an algorithm in the atomic servers model which allows clients to read and write a data object, such that the client operations appear to happen atomically, and such that every client invocation has a response, as long as at most f servers fail.

The atomic servers model is similar to the network-based model we implemented *LDR* in, and *LDR* is a network analogue of an f-srca. The lower bounds we prove in the atomic servers model have direct analogues in the network model, which we describe following the proof of each lower bound. The reason we use the atomic servers model is that it simplifies our proofs by removing details, such as message buffering, which are present in the network model; however, it is straightforward to translate the proofs we present to the network model. Therefore, using the atomic servers model does not weaken our lower bounds.

7.2 Write on Read Necessity

Recall that when a client reads in *LDR*, it writes to the directories during phase *rdw*. Similarly, in *ABD* and other replication algorithms, clients also write during reads. Our first lower bound shows that this is inherent: in any f-srca with $f > 0$, clients must write to some servers during a read. More precisely, let ϕ be a complete (read or write) operation by some client i in a trace of an f-srca. We will think of ϕ both as an ordered pair consisting of an invocation and response, and as a subsequence of the trace, beginning at the invocation and ending at the response. We define $\Delta(\phi)$ to be the number of servers j such that $modify(*)_{i,j}$ occurs during (subtrace) ϕ. That is, we count the $modify(*)_{i,*}$ actions occurring during ϕ as writes performed by ϕ. We do this because $modify(*)_{i,j}$ potentially changes the state of server j, and to do so, it must write to j.

The following theorem says that in any f-srca, a read must sometimes write to at least f servers.

Theorem 2. *In any f-srca A, there exists a complete client read operation ϕ in an execution of A such that $|\Delta(\phi)| \geq f$.*

Proof. The intuition for the proof is that during the course of a write operation, the algorithm is sometimes in an ambiguous state, in which another logical read can return either an old value or the new value being written. A reader needs to write to record which value it decided to return, so that later reads can make a consistent decision. Since any server the reader writes to may fail, the reader must write to at least f servers.[11]

Suppose for contradiction there exists an f-srca A, such that for any complete read operation ϕ in any execution of A, $|\Delta(\phi)| < f$. Consider an execution $\alpha = s_0\pi_1 s_1 \ldots \pi_n s_n$ of A starting from initial state s_0, in which a client w_1 writes a value $v_1 \neq v_0$, where v_0 is the default value of x. Let $\alpha(i) = s_0\pi_1 s_1 \ldots \pi_i s_i$ be the length $2i + 1$ prefix of α ($\alpha(0) = s_0$). Let i^* be the smallest i such that there exists a client read starting from s_i, so that if this read runs in isolation (*i.e.*, we

[10] f-srca stands for *f-strongly consistent replica control algorithm.*
[11] We'll see later why the reader writes to f and not $f + 1$ servers.

pause w_1 and only run the read), it may return v_1. Thus, we choose i^* to be the first "ambiguous" point in w_1's write, when a client read can return either v_0 or v_1. Note that all reads starting after $\alpha(i)$, for $i < i^*$, must return v_0. Clearly, $1 \leq i^* \leq n$. Let p_1 be the server, if any, that changed its state from state s_{i^*-1} to s_{i^*}. Note that there can be at most one such server, since only one server can change its state after each action.

Now let α_1 be an execution consisting of $\alpha(i^*)$ appended by a complete logical read operation ϕ_1 returning v_1. Let α_2 be an execution consisting of α_1, appended by another complete logical read operation ϕ_2, such that ϕ_2 does not (physically) read from any server in $\Delta(\phi_1) \cup p_1$. That is, ϕ_2 does not read from any server that ϕ_1 wrote to, nor from p_1. We first argue why ϕ_2 exists. By assumption, $|\Delta(\phi)| < f$, so that $|\Delta(\phi_1) \cup p_1| \leq f$. In ϕ_2, we *delay* processing ϕ_2's read invocations at all the servers in $\Delta(\phi_1) \cup p_1$ indefinitely, so that it looks to ϕ_2 like the servers in $\Delta(\phi_1) \cup p_1$ failed. Since A guarantees liveness when at most f servers fail, ϕ_2 must still terminate, without reading from $\Delta(\phi_1) \cup p_1$. This shows that ϕ_2 exists, and $\alpha(i^*)\phi_1\phi_2$ is a valid execution of A. Note that ϕ_2 returns v_1, since ϕ_2 occurs after ϕ_1, which returns v_1.

We now claim that $\alpha(i^* - 1)\phi_2$ is also a valid execution of A. Indeed, only the servers in $\Delta(\phi_1) \cup p_1$ can change their state from the final state of $\alpha(i^* - 1)$ to the final state of $\alpha\phi_1$.[12] Since ϕ_2 does not read from any server in $\Delta(\phi_1) \cup p_1$, the final state of $\alpha(i^* - 1)$ and $\alpha\phi_1$ look the same to ϕ_2. So, since $\alpha(i^*)\phi_1\phi_2$ is a valid execution of A, $\alpha(i^* - 1)\phi_2$ is also a valid execution. However, all logical reads starting after $\alpha(i^* - 1)$ return v_0, which is a contradiction because ϕ_2 returns v_1. This shows A does not exist, and all f-srca's must sometimes write to f servers during a read.

To translate this lower bound to the standard network model, we say that for any atomic replication algorithm in the network model tolerating f server faults, there exists a read operation in an execution of the algorithm in which at least f servers change their state.

7.3 Proportional Storage Necessity

Recall that a replica in *LDR* sometimes stores several values of x when there are multiple concurrent client writes. Our second lower bound shows that this behavior is not an artifact of *LDR*, but is inherent in a class of efficient replication algorithms we call *selfish* f-srcas. Intuitively, a selfish f-srca is one in which the clients do not "help" each other (much). Helping is a crucial ingredient in implementing lock-free concurrent objects, as in [11]. But helping has adverse effects on performance, since clients must do work for other operations as well as their own. In a selfish f-srca, we only allow clients to help each other with "cheap" operations. In particular, clients can help each other write metadata, such as tags, but cannot help write data (values of x), since we assume the data is

[12] Only p_1 can change its state from s_{i^*-1} to s_{i^*}, and only servers receiving *modify* invocations can change state during ϕ_1.

large and expensive to write. For example, *LDR* is a selfish f-srca, since a reader never writes data, only metadata, and a writer only writes its own value, and does not help write the values of other writes. On the other hand, *ABD* is not a selfish f-srca, because a reader writes values of x during its second phase. The comparison in Section 6.3 shows that a selfish f-srca such as *LDR* can be more efficient than an unselfish one such as *ABD*. We show that a disadvantage of selfish f-srcas is that they require the servers to use storage that is proportional to the number of concurrently writing clients. In the following, we formalize the notions of selfish f-srcas and the amount of storage that the servers use.

To make our result more general, we want an abstract measure of the storage used by the servers, not tied down to a particular storage format. Let α be an execution of an f-srca, and let $v \in V$. We say v is *g-erasable after* α if, by failing some set of g servers after α, we ensure that no later client read can return value v. That is, the failure of some g servers after α is enough to "erase" all knowledge of v. We define the *multiplicity* of v after α, $m(v, \alpha)$, to be the smallest g such that v is g-erasable after α. If $m(v, \alpha) = h$, then intuitively, exactly h servers know about v, and the amount of storage used for v is proportional to h.

We now to formally define selfish f-srcas, trying to capturing the idea that client reads do not write values of x, and client writes only write their own value. Let A be an f-srca. We say an execution of A is *server-exclusive* if at any point in the execution, there is at most one client accessing any server. In a server-exclusive execution, we can easily "attribute" every action to a particular client. If the action is performed by a client, we attribute the action to that client. If the action is performed by a server, then the server must be responding to some client's invocation; we attribute the action to that client. We now define selfish f-srcas as follows:

Definition 4. *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 attributed to client $i \in \mathcal{C}$, let s_π be the state in α before π, and let s'_π be the state in α after π.*

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

This definition says that in a server-exclusive execution of a selfish f-srca, client reads do not increase the multiplicity of any value, and clients writes can only increase the multiplicity of their own value.

Definition 5. *Let A be an f-srca. Define the* storage used by *A, $M(A)$, to be the supremum, over all executions α of A, of $\sum_{v \in V} m(v, \alpha)$.*

Assuming the storage needed for a value of x is large compared to the storage for metadata that the servers use, $M(A)$ is an abstract measure of the amount of storage used by the servers of A.

Lastly, we define an (f, c)-srca as an f-srca which only guarantees liveness and atomicity when there are $\leq c$ concurrent writers in an execution. We now state the second lower bound.

Theorem 3. *Let A be an (f, c)-srca, where f and c are positive integers.*[13] *Then $M(A) \geq fc$.*

Proof. Suppose for contradiction that $M(A) < fc$. The intuition for the proof is that if we run c client writes concurrently, then because $M(A) < fc$, we can ensure none of the values written have multiplicity greater than f. Then, in later client reads, we can delay responses from f servers at a time to ensure that consecutive reads do not return the same value. But eventually some two non-consecutive reads must return the same value. This violates atomicity, and shows that A does not exist.

Let W be a set of c writer clients, all writing distinct values different from v_0. Construct an execution α starting from an initial state of A using to the following procedure:

1. Repeat steps 2 or 3, as long as no $w \in W$ has finished its write.
2. If any $w \in W$ has an action π enabled, and π is not an invocation at a server, extend α by letting w run π.
3. Otherwise, choose a $w \in W$ with invocation π at server j enabled, such that the following holds: if we extend α to α', by running π and then letting server j run until it outputs a response to π, then $\forall v \in V \backslash \{v_0\} : m(v, \alpha') \leq f$. Set $\alpha \leftarrow \alpha'$.

It is easy to see that α is a server-exclusive execution. Also, every value except possibly v_0 has multiplicity at most f after α. This is because when step 2 of the procedure occurs, only client w changes state, and no servers. Therefore, the multiplicity of any value cannot increase. When step 3 occurs, the server j that runs was chosen so that it does not increase the multiplicity of any value beyond f. Lastly, we claim that some $w \in W$ finishes its write after α. To see this, first observe that in any prefix of α, there must exist a value v_w being written by $w \in W$ that has multiplicity $< f$, since there are c values being written, and the sum of all their multiplicities is at most $M(A) < cf$. Then, the above procedure can run w and any server which w invokes, because doing this increases the multiplicity of v_w by at most 1, and leaves the multiplicity of every other value unchanged (because A is selfish). So, as long as no writer is finished, the procedure can extend α to a longer execution. Thus, since algorithm A guarantees liveness, some writer must eventually take enough steps in α to finish. Let α' be the prefix of α up to when the first writer w finishes.

Now we start a sequence of non-overlapping client reads $\{\phi_i\}_i$ after α'. Let read ϕ_i return value v_i. Since w finished writing v_w, by atomicity, no ϕ_i can return v_0 (x's initial value). For each v_i, let F_i be a set of f servers such that, if we fail F_i, no later client read can return v_i. F_i exists, because no value except possibly v_0 has multiplicity greater than f, and no ϕ_i increases the multiplicity of any value. During ϕ_i, we delay the responses from all servers in F_{i-1} indefinitely, so that it seems to ϕ_i like the servers in F_{i-1} failed. Then, since ϕ_i must tolerate f server failures, ϕ_i must finish without (physically) reading from any server in

[13] The theorem does not hold for $f = 0$, as we explain in the full paper [6].

F_{i-1}. Therefore, ϕ_i cannot return v_{i-1}, *i.e.*, two consecutive reads cannot return the same value. Since there are only c values which any client read can return, eventually some $v_i = v_j$, where $j - i > 1$. Choose k such that $i < k < j$. We now have a contradiction because v_k linearizes after v_i, and v_j linearizes after v_k, but $v_i = v_j$. This shows that A does not exist, and $M(A) \geq fc$ for all selfish (f, c)-srcas.

To translate this lower bound to the network model, we say that the servers in any atomic replication algorithm in the network model tolerating f server faults must have storage proportional to the maximum number of concurrently writing clients.

8 Conclusions

In this paper we presented *LDR*, an efficient replication algorithm based on separately replicating data and metadata. Our algorithm is optimal when the size of the data is large compared to the metadata. We also presented two lower bounds. One states that the number of writes necessary within some client read operation equals at least the fault-tolerance of the algorithm. The other states that servers in a selfish replication algorithm need storage proportional to the number of concurrent writers. The separation of data from metadata was the key to *LDR*'s efficiency. We are interested in extending this idea to enhance the performance of other distributed algorithms.

References

1. A. Adya, W. Bolosky, M. Castro, and G. Cermak et al. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the fifth symposium on operating systems design and implementation*, 2002.
2. Y. Amir, D. Dolev, P. Melliar-Smith, and L. Moser. Robust and efficient replication using group communication, 1994.
3. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995.
4. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley Longman Publishing Co., Inc., 1987.
5. Y. Breitbart and H. F. Korth. Replication and consistency: being lazy helps sometimes. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 173–184. ACM Press, 1997.
6. R. Fan. Efficient replication of large data-objects. Technical Report MIT-LCS-TR-886, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA 02139, February 2003.
7. David K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh symposium on Operating systems principles*, pages 150–162, 1979.
8. J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182. ACM Press, 1996.
9. J. Welch H. Attiya. *Distributed Computing.* McGraw Hill International, Ltd., 1998.

10. M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. pages 13–26. ACM Press, 1987.
11. Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
12. R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, 1992.
13. N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, March 1996.
14. N. Lynch and A. 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.
15. E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 126–137. Morgan Kaufmann, 1999.
16. 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.
17. K. Petersen and K. Li. An evaluation of multiprocessor cache coherence based on virtual memory support. In *Proc. of the 8th Int'l Parallel Processing Symp. (IPPS'94)*, pages 158–164, 1994.
18. 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.