Fine-grained Analysis on Fast Implementations of Distributed Multi-writer Atomic Registers

Kaile Huang, Yu Huang, Hengfeng Wei
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, Jiangsu Province, China
mg1933024@smail.nju.edu.cn,{yuhuang,hfwei}@nju.edu.cn

ABSTRACT

Distributed multi-writer atomic registers are at the heart of a large number of distributed algorithms. While enjoying the benefits of atomicity, researchers further explore fast implementations of atomic reigsters which are optimal in terms of data access latency. Though it is proved that multi-writer atomic register implementations are impossible when both read and write are required to be fast, it is still open whether implementations are impossible when only write or read is required to be fast. This work proves the impossibility of fast write implementations based on a series of chain arguments among indistiguishable executions. We also show the necessary and sufficient condition for fast read implementations by extending the results in the single-writer case. This work concludes a series of studies on fast implementations of distributed atomic registers.

CCS CONCEPTS

• **Theory of computation → Distributed algorithms**.

KEYWORDS

atomicity, fast implementation, impossibility result, chain argument

ACM Reference Format:
Kaile Huang, Yu Huang, Hengfeng Wei. 2020. Fine-grained Analysis on Fast Implementations of Distributed Multi-writer Atomic Registers. In *Symposium on Principles of Distributed Computing (PODC '20), August 3–7, 2020, Virtual Event, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3382734.3405698

1 INTRODUCTION

Distributed storage systems employ replication to improve performance by routing data queries to data replicas nearby [1, 2, 19]. System reliability is also improved due to the redundancy of data. However, data replication is constrained by the intrinsic problem of maintaining data consistency among different replicas [27]. The data consistency model acts as the "contract" between the developer and the storage system. Only with this contract can the developers reason about and program over the data items which actually exist as multiple replicas [9, 26].

PODC '20, August 3–7, 2020, Virtual Event, Italy

ACM ISBN 978-1-4503-7582-5/20/08…$15.00
https://doi.org/10.1145/3382734.3405698

Atomicity is a strong consistency model [17, 20, 21]. It allows concurrent processes to access multiple replicas of logically the same data item, as if they were accessing one data item in a sequential manner. This abstraction, usually named an *atomic register*, is fundamental in distributed computing and is at the heart of a large number of distributed algorithms [5, 23]. Though the atomicity model greatly simplifies the development of upper-layer programs, it induces longer data access latency. The latency of read and write operations is mainly decided by the number of round-trips of communications between the reading and writing clients and the server replicas. In the single-writer case, the read operation on an atomic register needs two round-trips of communications [5]. In the multi-writer case, both write and read operations need two round-trips of communications [4, 23].

In distributed systems, user-perceived latency is widely regarded as the most critical factor for a large class of applications [1, 2, 19, 22, 24]. While enjoying the benefits of atomicity, researchers further explore whether we can develop *fast* implementations for atomic registers. Since two round-trips are sufficient to achieve atomicity, fast implementation means one round-trip of communication, which is obviously optimal. In the single-writer case, it is proved that when the number of reading clients exceeds certain bound, fast read is impossible [12]. In the multi-writer case, it is proved impossible when both read and write are required to be fast [12].

This leaves an important open problem when examining the design space of fast implementations of multi-writer atomic registers in a fine-grained manner. Specifically, we denote fast write implementations as W1R2, meaning that the write operation finishes in one round-trip, while the read operation finishes in two round-trips. Similarly, we denote fast read implementations as W2R1 and fast read-write implementations as W1R1. Existing work only proves that fast read-write (W1R1) implementations are impossible. It is still open whether fast write (W1R2) and fast read (W2R1) are impossible. This impossibility result (yet to be proved) underlies the common practice of quorum-replicated storage system design, e.g. the Cassandra data store [19]: when read or write is required to finish in one round-trip, weak consistency has to be accepted.

This work thoroughly explores the design space of fast implementations of multi-writer atomic registers. Specifically, for fast write (W1R2) implementations, we prove that it is impossible to achieve atomicity. The impossibility proof is mainly based on the chain argument to construct the indistinguishability between executions. Unlike the W1R1 case, the chain argument for W1R2 implementations faces two severe challenges:

- (Section 3) Since the read operation has one more round-trip (compared to the W1R1 case) to discover differences

Table 1: Overview of contributions.

Design space	Impossibility	Implementation
W2R2 [23]	$t \geq \frac{S}{2}$	$W \geq 2, R \geq 2, t < \frac{S}{2}$
W1R2 [this work]	$W \geq 2, R \geq 2, t \geq 1$	$\emptyset$
W2R1 [this work]	$R \geq \frac{S}{t} - 2$	$R < \frac{S}{t} - 2$
W1R1 [12]	$W \geq 2, R \geq 2, t \geq 1$	$\emptyset$

between executions, it is more difficult to construct the indistinguishability we need for the impossibility proof. To this end, we combine three consecutive rounds of chain arguments, in order to hide the differences in the executions from the 2-round-trip read operations.

- (Section 4) The first round-trip of a read operation might update information on the servers, thus potentially affecting the return values of other read operations. The effect from the first round-trip of read operations may also break the indistinguishability we try to construct. To this end, we use sieve-based construction of executions to eliminate the effect of the first round-trip of a read operation.

The impossibility proof for W1R2 implementations is the main contribution of this work.

For W2R1 implementations, we prove the impossibility when $R \geq \frac{S}{t} - 2$. When $R < \frac{S}{t} - 2$, we propose a W2R1 implementation. The proof and the implementation are extensions to the results of the single-writer case [12].

The contributions in this work conclude a series of studies on fast implementations of distributed atomic registers. The contributions of this work in light of results in the existing work are outlined in Table 1.

The rest of this work is organized as follows. In Section 2, we describe the preliminaries. Section 3 and Section 4 present the impossibility proof for W1R2 implementations. Section 5 outlines the impossibility proof and the algorithm design of W2R1 implementations. Section 6 discusses the related work. In Section 7, we conclude this work and discuss the future work.

2 PRELIMINARIES

In this section, we first describe the system model and the definition of atomicity. Then we outline the algorithm schema for multi-writer atomic register implementations.

2.1 Atomic Register Emulation in Message-passing Systems

We basically adopt the system model used in [12]. Specifically, a replicated storage system considered in this work consists of three disjoint sets of processes:

- the set Σ_{sv} of *servers*: $\Sigma_{sv} = \{s_1, s_2, \cdots, s_S\}$.
- the set Σ_{rd} of *readers*: $\Sigma_{rd} = \{r_1, r_2, \cdots, r_R\}$.
- the set Σ_{wr} of *writers*: $\Sigma_{wr} = \{w_1, w_2, \cdots, w_W\}$.

Here, S, R and W denote the cardinalities of Σ_{sv}, Σ_{rd} and Σ_{wr} respectively. The readers and the writers are also called *clients*. We are concerned of *multi-writer multi-reader* implementations. Thus we have $W \geq 2$ and $R \geq 2$. In a distributed message-passing system, we also have that $S \geq 2$. The clients and the servers communicate by asynchronous message passing, via a bidirectional reliable communication channel, as shown in Fig. 1. There is no communication among the servers. For the simplicity of presentation, we assume the existence of a discrete global clock, but the processes cannot access the global clock. An implementation $\mathcal{A}$ of a shared register is a collection of automata. Computation proceeds in steps of $\mathcal{A}$. An execution is a finite sequences of steps of $\mathcal{A}$. In any given execution, any number of readers and writers, and t out of S servers may crash.

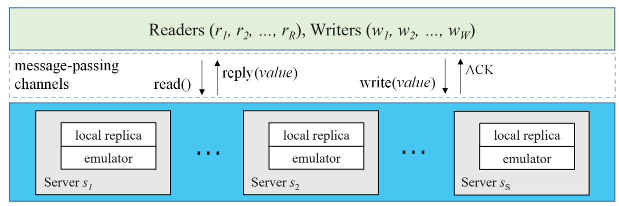

Figure 1: System model of read/write register emulation.

An atomic register is a distributed data structure that may be concurrently accessed by multiple clients, yet providing an "illusion of a sequential register" to the accessing processes. The atomic register provides two types of operations. Only a writer can invoke the write operation $write(v)$, which stores v in the register. Only a reader can invoke the read operation $read()$, which returns the value stored. We are concerned of *wait-free* implementations, where any read or write invocation eventually returns independently of the status of other clients. Due to the locality property of atomicity [17], we consider one single shared register.

We define an *execution* of the clients accessing the shared register as a sequence of events where each event is either the invocation or the response of a read or write operation. Each event in the execution is tagged with a unique timestamp from the global clock, and events appear in the execution in increasing order of their timestamps. For execution σ, we can define the partial order between operations. Let $O.s$ and $O.f$ denote the timestamps of the invocation and the response events of operation O respectively. We define $O_1 \prec_\sigma O_2$ if $O_1.f < O_2.s$. We define $O_1 || O_2$ if neither $O_1 \prec_\sigma O_2$ nor $O_2 \prec_\sigma O_1$ holds. An execution σ is *sequential* if σ begins with an invocation, and each invocation is immediately followed by its matching response. An execution σ is *well-formed* if for each client p_i, $\sigma | p_i$ (the subsequence of σ restricted on p_i) is sequential. Given the notations above, we can define atomicity:

Definition 2.1. A shared register provides *atomicity* if, for each of its well-formed executions σ, there exists a permutation π of all operations in σ such that π is sequential and satisfies the following two requirements:

- [Real-time requirement] If $O_1 \prec_\sigma O_2$, then O_1 appears before O_2 in π.
- [Read-from requirement] Each read returns the value written by the latest preceding write in π.

2.2 Algorithm Schema for Multi-writer Atomic Register Implementations

When studying fast implementations of multi-writer atomic registers, the critical operation we consider is the round-trip of communication between the client and the servers. In each round-trip, the client can *query* all the servers, i.e., collect useful information from the servers. The client can also *update* all the servers, i.e., send useful information to the servers. Upon receiving a *query* request, the server replies the client as required. Upon receiving an *update* request, the server first stores data sent from the client. Then it can reply certain information if necessary, or it can simply reply an ACK. Exemplar implementations can be found in [4, 5, 18, 23, 28].

Tuning the number of round-trips in emulation of a multi-writer atomic register, we have four possible types of implementations [25], as shown in Fig. 2. They are slow read-write implementation (W2R2), fast write implementation (W1R2), fast read implementation (W2R1) and fast read-write implementation (W1R1). Fig. 2 can be viewed as the Hasse Diagram of the partial order among implementations. The partial order relation can be thought of as providing stronger consistency guarantees or inducing less data access latency.

Note that, for atomic register implementations, when 2 round-trips are sufficient, we do not consider implementations employing k round-trips for $k \geq 3$. However, for impossibility of fast implementations, we need to consider the impossibility of W1Rk and WkR1 implementations for $k \geq 3$. The impossibility proofs of W1Rk and WkR1 implementations are principally same with the impossibility proofs of W1R2 and W2R1 implementations, as discussed in Section 3 and Section 5 respectively.

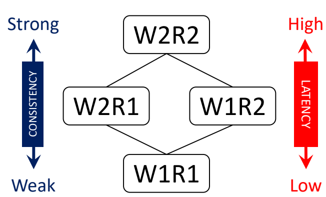

Figure 2: Algorithm schema for multi-writer atomic register implementations.

3 FAST WRITE (W1R2): CHAIN ARGUMENTS FOR IMPOSSIBILITY PROOF

We first present the impossibility proof for fast write (W1R2) implementations. Specifically, we prove the following theorem:

Theorem 1 (W1R2 impossibility). *Let $t \geq 1$, $W \geq 2$ and $R \geq 2$. There is no fast write* (W1R2) *atomic register implementation.*

This impossibility result is proved by chain argument [6], which is also used to prove the impossibility of W1R1 implementations in [12]. The central issue in chain argument is to construct certain indistinguishability between executions. Compared to the impossibility proof of W1R1 implementations, the read operations now have one more round-trip. This "one more round-trip" imposes two critical challenges for constructing the indistinguishability:

(1) Obtaining more information from the second round-trip, the read operations can now "beat" the indistinguishability constructed in the W1R1 case. In our proof, we add one more read operation and construct two more chains of executions, in order to obtain the indistinguishability even when facing two round-trips of read operations.
(2) The first round-trip of a read operation may update information on the servers, thus possibly affecting the return values of other read operations. The effect of the first round-trip may also break the indistinguishability we plan to construct. To cope with this challenge, we propose the sieve-based construction of executions. We sieve all the servers and eliminate those which are affected by the first round-trip of a read operation. On the servers that remain after the sieving, we show that the chain argument can still be successfully conducted.

This section addresses the first challenge and presents the chain argument. In Section 4, we address the second challenge and discuss how to eliminate effects of the first round-trip. Note that the impossibility proof of W1R2 implementations also applies for W1Rk implementations for $k \geq 3$. We can combine the round-trips $2, 3, \cdots, k$ as if they were one single round-trip. The chain argument still applies.

3.1 Overview

It suffices to show the impossibility in a system where $S \geq 3$ [2], $W = 2$, $R = 2$ and $t = 1$. In the proof, we use two write operations W_1 and W_2 (issued by writers w_1 and w_2 respectively) and two read operations R_1 and R_2 (issued by readers r_1 and r_2 respectively). Since $t = 1$, the read operation must be able to return when one server gives no response. When constructing an execution, we say one round-trip in an operation *skips* one server s, if the messages between the client and the server are delayed a sufficiently long period of time (e.g. until the rest of the execution has finished). If one round-trip of communication does not skip any server, we say it is *skip-free*.

In a chain argument, we will construct a chain of executions, where two consecutive executions in the chain differ only on one server. Since in two end executions of the chain the read operations return different values, there must be some "critical server". The change on the critical server results in the difference in the return values. We intentionally let the read operation skip the critical server. This will construct the indistinguishability we need (as detailed in Section 3.2).

In a chain argument, we may also utilize the relation between operations to construct the indistinguishability (as detailed in Section 3.4). Specifically, one operation cannot notice the differences in executions after it has finished. Moreover, the operation cannot notice the differences on one server if it skips this server.

The indistinguishability makes the read operations return the same value in two executions. However, construction of the chain of executions tells us that the two executions should return different values (note that within one execution, two reads must return the

[2] In a replicated system, we have $S \geq 2$. When $S = 2$ while $t = 1$, it is trivial to prove the impossibility.

same value, as required by the definition of atomicity). This leads to contradiction.

To beat the ability of the read operation to employ two round-trips of communications, we need to conduct a series of chain arguments. The proof will be presented in three phases, each phase constructing one chain, as shown in Fig. 3. For the ease of presentation, we assume in the chain argument that the first round-trip of a read operation will not affect the return values of other read operations. In Section 4, we will explain how to lift this assumption.

3.2 Phase 1: Chain α and Critical Server s_{i_1}

To construct chain α, we first construct the "head" and the "tail" executions:

- Head execution α_{head} consists of the following three non-concurrent operations: i) a skip-free $W_1 = write(1)$, which precedes ii) a skip-free $W_2 = write(2)$, which precedes iii) a skip-free $R_1 = read()$. Note that all servers receive the three operations in this order. In α_{head}, R_1 returns 2 (as required by the definition of atomicity).
- Tail execution α_{tail} consists of the same three non-concurrent operations, but the temporal order is W_2, W_1 and R_1. In α_{tail}, R_1 returns 1.

Let $\alpha_0 = \alpha_{head}$. From α_0 we construct α_1, the next execution in the chain, as follows. Execution α_1 is identical to α_0 except that server s_1 receives W_2 first, and then W_1 and R_1. That is, we "swap" two write operations on s_1 and everything else is unchanged. Continuing this "swapping" process, we swap two write operations on s_i in α_{i-1} and obtain α_i, for all $1 \leq i \leq S$. Thus we obtain chain $\alpha = (\alpha_0, \alpha_1, \cdots, \alpha_S)$. Note that R_1 cannot distinguish α_S from α_{tail}. Thus, R_1 returns 1 in α_S, while it returns 2 in α_0.

Since R_1 returns different values in two ends of the chain, there must exist two consecutive executions α_{i_1-1} and α_{i_1} $(1 \leq i_1 \leq S)$, such that R_1 returns 2 in α_{i_1-1} and returns 1 in α_{i_1}. Note that α_{i_1-1} and α_{i_1} differ only on one "critical server" s_{i_1}, i.e., s_{i_1} receives W_1 first in α_{i_1-1}, and receives W_2 first in α_{i_1}. This critical server s_{i_1} will be intentionally skipped to obtain indistinguishability, when constructing chain β in Phase 2 below.

3.3 Phase 2: Chain β Derived from Chain β' and Chain β''

In Phase 2 of our proof, we basically append the second read operation R_2 to executions in chain α and obtain chain β. We actually construct two candidate chains β' and β'', and modify one of them to get chain β, depending on what the return value of R_2 is. Chain β' and β'' stem from execution α_{i_1-1} and α_{i_1} respectively, i.e. two executions pertained to the critical change on the critical server.

Since the read operations consist of two round-trips, we denote the two round-trips of read operation R_i as $R_i^{(1)}$ and $R_i^{(2)}$ $(i = 1, 2)$. We extend execution α_{i_1-1} with the second read operation R_2. We interleave the round-trips of R_1 and R_2 as follows: the four round-trips are non-concurrent and the temporal order is $R_1^{(1)}$, $R_2^{(1)}$, $R_1^{(2)}$ and $R_2^{(2)}$ on all servers s_i $(1 \leq i \leq S)$, as shown in Fig. 3. This execution is named $\beta'_{head} = \beta'_0$. To construct chain β', we will swap $R_1^{(2)}$ and $R_2^{(2)}$ on one server a time. Specifically, for $1 \leq i \leq S$, β'_i is the same with β'_{i-1}, except that server s_i receives $R_1^{(2)}$ first in β'_{i-1}, and receives $R_2^{(2)}$ first in β'_i. The last execution of the chain is $\beta'_{tail} = \beta'_S$.

We then extend execution α_{i_1} in the same way, and get $\beta''_{head} = \beta''_0$. We also do the swapping in the same way and get executions $\beta''_1, \beta''_2, \cdots, \beta''_S$. The only difference between chain β' and β'' is that, chain β' stems from execution α_{i_1-1}, while chain β'' stems from α_{i_1}. Thus, R_1 returns 2 in chain β', while returning 1 in chain β''. This is because, the return value of R_1 is decided by executions α_{i_1-1} and α_{i_1}. Appending the read operation R_2 should not change the return value of an existing read, as required by the definition of atomicity.

The only server which can tell the difference between β' and β'' is s_{i_1}, the critical server in chain α. Now we modify tail executions β'_{tail} and β''_{tail}, in order to obtain the indistinguishability we need. In both tail executions β'_{tail} and β''_{tail}, we let R_2 (both round-trips) skip server s_{i_1}. Thus the (modified) β'_{tail} and β''_{tail} are indistinguishable to R_2, and R_2 returns the same value in both modified tail executions.

To construct chain β, we must start from either β' or β'', and revise the chosen candidate chain into chain β. The criteria for choosing a chain is that the candidate chain must enable us to make the read operations in the two end executions β_0 and β_S have different return values. Without loss of generality, we assume that R_2 returns 1 in both β'_{tail} and β''_{tail} (modified, with R_2 skipping s_{i_1}). In β'_0, since R_1 returns 2, according to the definition of atomicity, we have that R_2 must also return 2 in β'_0. Thus, we choose chain β'.

We modify chain β' to obtain chain β as follows. For every execution in chain β', we let R_2 (both round-trips) skip s_{i_1} and obtain every corresponding execution in chain β. That is, R_2 in chain β' is skip-free while R_2 in chain β skips s_{i_1} (if R_2 returns 2 in both modified β'_{tail} and β''_{tail}, we will choose to revise chain β'', and obtain chain β in the same way). Chain β servers as the basis for construction of chain γ and $\mathbb{Z}$ in Phase 3 of our proof.

3.4 Phase 3: Zigzag Chain $\mathbb{Z}$ Combining Chain β and γ

Given chain β, we have that R_1 and R_2 both return 2 in β_0, while both read operations return 1 in β_S. Now in Phase 3, we will first construct chain $\gamma = (\gamma_0, \gamma_1, \cdots, \gamma_{S-1})$. Then we combine chain β and γ, and obtain the zigzag chain $\mathbb{Z}$, as shown in Fig. 3.

For any two executions x and x' from chain β and γ, we define an equivalence relation: $x \approx x'$ when R_1 and R_2 return the same value in both x and x'. Note that R_1 and R_2 must return the same value in one execution, as required by the definition of atomicity. We will prove that all executions in chain $\mathbb{Z}$ are connected by the '$\approx$' relation, i.e., $\beta_0 \approx \gamma_0 \approx \beta_1 \approx \gamma_1 \approx \cdots \beta_{S-1} \approx \gamma_{S-1} \approx \beta_S$. According to our construction in Phase 1 and 2, we have that $\beta_0 \not\approx \beta_S$. This leads to contradiction.

We first construct the horizontal links in chain $\mathbb{Z}$, i.e., $\forall 0 \leq k \leq S-1, \beta_k \approx \gamma_k$ in Section 3.4.1. Then we construct the diagonal links, i.e., $\forall 0 \leq k \leq S-1, \beta_{k+1} \approx \gamma_k$ in Section 3.4.2.

3.4.1 *Horizontal link from β_k to γ_k.*

We first construct execution γ_k from β_k $(0 \leq k \leq S-1)$. The construction process implies that $\beta_k \approx \gamma_k$. The key behind the process is still constructing certain

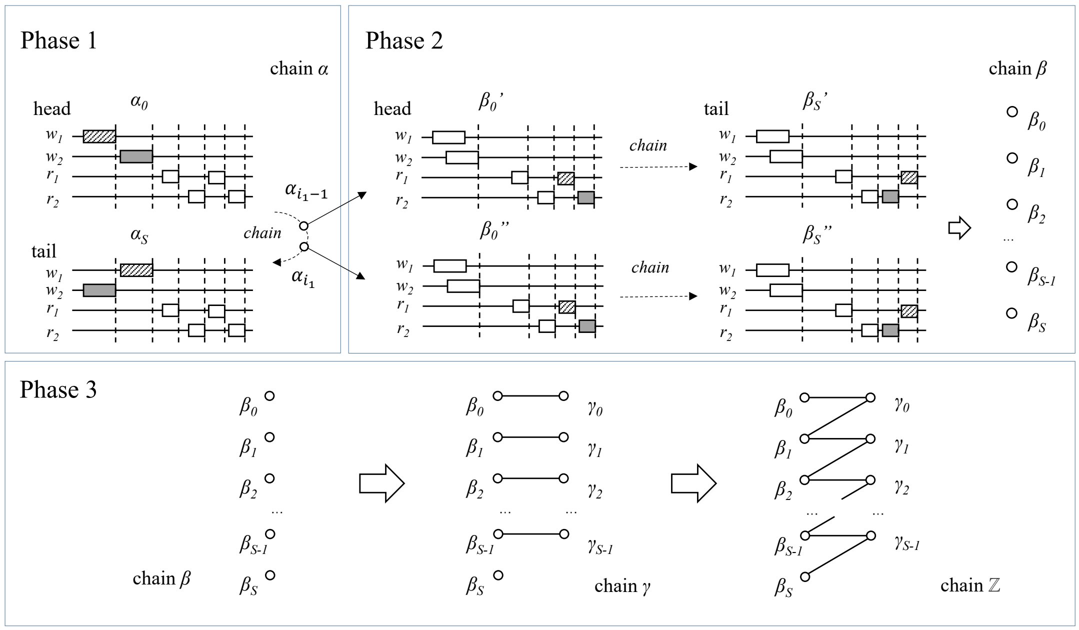

Figure 3: Proof overview.

indistinguishability. When constructing γ_i, we need to utilize two sources of indistinguishability:

(1) When $R_1^{(2)}$ finishes before $R_2^{(2)}$ on some server s_x, and we modify $R_2^{(2)}$ on s_x, $R_1^{(2)}$ will not notice the change (behind its back).
(2) When $R_2^{(2)}$ skips s_x, and we modify R_1 on s_x, R_2 will not notice the change.

The construction is shown in Fig. 4 and Fig. 5, from the reader's view and the server's view respectively.

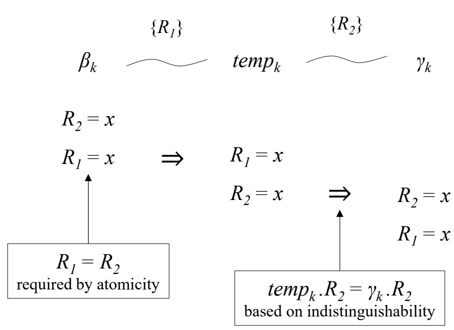

Figure 4: Construction of the horizontal link: the reader's view.

Before the construction of γ_k, we need to review the characteristics of all executions in chain β. For every execution β_k ($0 \le k \le S$), operation R_1 (both round-trips) is skip-free, while R_2 (both round-trips) skips exactly one server s_{i_1} (the critical server obtained from chain α, see Section 3.2). In the construction of γ_k, we will change

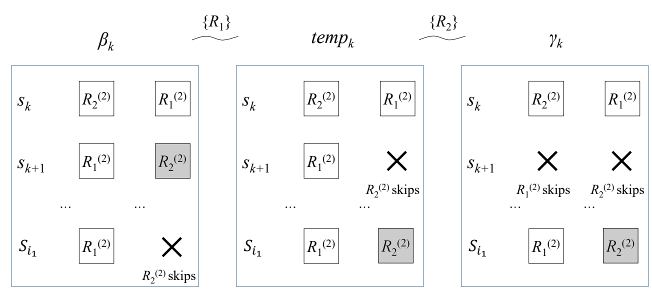

Figure 5: Construction of the horizontal link: the server's view.

the server $R_2^{(2)}$ skips. We will also let $R_1^{(2)}$ skip one server. No other modifications will be made to β_k. Also note that starting from β_0, for $k = 1, 2, \cdots, S$, we do the swapping in s_k and obtain execution β_k. That is, for β_k, s_{k+1} sees $R_1^{(2)}$ and then $R_2^{(2)}$ (not swapped); while s_k sees $R_2^{(2)}$ and then $R_1^{(2)}$ (swapped).

From β_k, we will create γ_k as follows. In construction of γ_k, we only modify $R_1^{(2)}$ and $R_2^{(2)}$, i.e., the first round-trips of both operations are unchanged. In β_k, the swapping takes place on s_k and we will pick the first not-swapped server, i.e., s_{k+1}. Server s_{k+1} finishes $R_1^{(2)}$ before it receives $R_2^{(2)}$. We create a temporary execution $temp_k$ which is the same with β_k except that $R_2^{(2)}$ skips s_{k+1} and does not skip s_{i_1}. The only two servers affected are s_{k+1} and s_{i_1}. Note that here we assume that $k+1 \neq i_1$. The case $k+1 = i_1$ (which is actually simpler) will be discussed separately below. For the two servers affected, we verify the indistinguishability for R_1 (see Fig. 5):

- For s_{k+1}, R_1 cannot see any difference since R_1 finishes first.
- For s_{i_1}, previously $R_2^{(2)}$ skips s_{i_1} (in β_k) and now we add $R_2^{(2)}$ back on s_{i_1} (in $temp_k$). We can intentionally add $R_2^{(2)}$ after $R_1^{(2)}$ on s_{i_1}. Thus R_1 still cannot see any difference.

Thus R_1 cannot distinguish β_k from $temp_k$, and R_1 will return the same value in both executions (see Fig. 4). As required by the definition of atomicity, R_2 will return the same value with R_1, thus returning the same value in both executions. This gives us that $\beta_k \approx temp_k$.

Now we create execution γ_k which is the same with $temp_k$ except that $R_1^{(2)}$ skips s_{k+1} (note that in β_k and $temp_k$, $R_1^{(2)}$ is skip-free). The only change takes place on s_{k+1}. Since $R_2^{(2)}$ skips s_{k+1} in both $temp_k$ and γ_k, R_2 cannot distinguish $temp_k$ from γ_k. Thus we have that R_2 will return the same value in $temp_k$ and γ_k (see Fig. 4). Also as required by the definition of atomicity, R_1 will return the same value in both executions. This gives us $temp_k \approx \gamma_k$.

Finally, combining the two links above (see Fig. 4 and Fig. 5), we have $\beta_k \approx \gamma_k$. Here note that since $R_2^{(2)}$ skips s_{k+1}, it seems unnecessary for R_1 to skip s_{k+1} in γ_k. For the proof till now, it is indeed unnecessary. However, we need to let R_1 skip s_{k+1} here, in order to construct the diagonal link between β_{k+1} and γ_k later in the following Section 3.4.2.

In the proof above, we left out the case $k + 1 = i_1$, which is discussed here. When $k+1 = i_1$, we create γ_k as follows. In β_k, s_{k+1} only receives $R_1^{(2)}$ (since $R_2^{(2)}$ skips $s_{i_1} = s_{k+1}$). We let $R_1^{(2)}$ skip s_{k+1}, and get γ_k. Since $R_2^{(2)}$ skips s_{k+1}, R_2 cannot distinguish β_k from γ_k and will return the same value. As required by the definition of atomicity, R_1 will also return the same value in β_k and γ_k. Thus we still have $\beta_k \approx \gamma_k$ when $k + 1 = i_1$.

3.4.2 Diagonal link from β_{k+1} to γ_k. Now we construct the diagonal link. We will create from β_{k+1} executions $temp'_k$ and γ'_k ($0 \le k \le S - 1$). The construction is principally the same with the construction of the horizontal link. We need to show that $\beta_{k+1} \approx temp'_k \approx \gamma'_k$. As for γ'_k and γ_k, the executions on all servers, together with the order among operations, are the same. It is straightforward to verify that $\gamma'_k \approx \gamma_k$ (so we do not show γ'_k in Phase 3 in Fig. 3). Thus we can obtain the diagonal link, meaning that $\beta_{k+1} \approx \gamma_k$. Now we explain construction of the diagonal link in detail.

First note that in β_{k+1}, the "swapping" (see Section 3.3) takes place in s_{k+1}. Thus s_{k+1} sees $R_2^{(2)}$ first and then $R_1^{(2)}$. We create execution $temp'_k$ which is the same with β_{k+1}, except that $R_1^{(2)}$ skips s_{k+1}. The only difference between $temp'_k$ and β_{k+1} is on s_{k+1}. Since $R_2^{(2)}$ finishes first on s_{k+1}, we have that R_2 cannot distinguish β_{k+1} from $temp'_k$, as shown in Fig. 6. So R_2 will return the same value in β_{k+1} and $temp'_k$. As required by the definition of atomicity, R_1 will also return the same value in β_{k+1} and $temp'_k$. Thus we have $\beta_{k+1} \approx temp'_k$. The construction from the server's view is shown in Fig. 7.

Now we construct execution γ'_k, which is the same with $temp'_k$ except that $R_2^{(2)}$ skips s_{k+1} and does not skip s_{i_1} (see Fig. 7). Similar to the horizontal link case, here we assume that $k + 1 \ne i_1$. We will discuss the simpler case "$k + 1 = i_1$" below. We need to show that

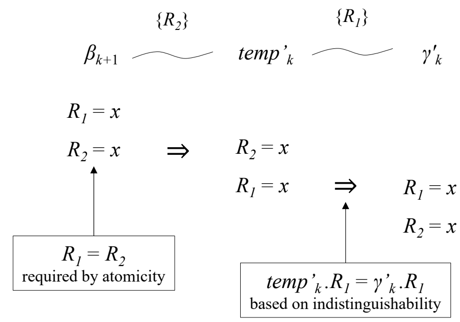

Figure 6: Construction of the diagonal link: the reader's view.

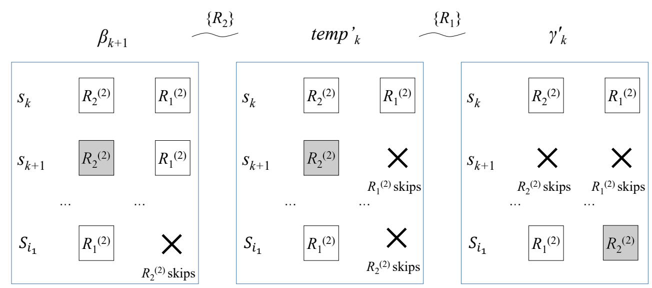

Figure 7: Construction of the diagonal link: the server's view.

R_1 cannot distinguish $temp'_k$ from γ'_k. The differences concern two servers s_{k+1} and s_{i_1}:

- As for s_{k+1}, since $R_1^{(2)}$ skips s_{k+1}, R_1 will not see the difference that $R_2^{(2)}$ skips s_{k+1}.
- As for s_{i_1}, now we add $R_2^{(2)}$ back on s_{i_1}. We can add $R_2^{(2)}$ after $R_1^{(2)}$ on s_{i_1}. Thus R_1 finishes first on s_{i_1}, not being able to distinguish $temp'_k$ from γ'_k.

Thus we have that R_1 returns the same value in $temp'_k$ and γ'_k. As required by the definition of atomicity, R_2 will also return the same value in both executions. This gives us that $temp'_k \approx \gamma'_k$.

It is straightforward to check that behaviors of R_1 and R_2 on every server, as well as the order among operations, in γ_i and γ'_i are the same. Thus we have $\gamma'_i \approx \gamma_i$. Note that here we can see the importance of the seemingly unnecessary change from $temp_k$ to γ_k (in Section 3.4.1): letting $R_1^{(2)}$ skip s_{k+1}. The "unnecessary" skipping of $R_1^{(2)}$ in the horizontal link helps us make γ_k and γ'_k behave principally in the same way. Finally, this gives us $\beta_{k+1} \approx \gamma_k$.

There is still the case "$k + 1 = i_1$" left, which is also simpler. We create γ'_k as follows. In β_{k+1}, s_{k+1} only receives $R_1^{(2)}$. Let $R_1^{(2)}$ skip s_{k+1}, and we will get γ'_k. Since R_2 skips s_{k+1}, R_2 cannot distinguish β_{k+1} from γ'_k and will return the same value. As required by the definition of atomicity, R_1 will return the same value in β_{k+1} and γ'_k too. Thus we still have $\beta_{k+1} \approx \gamma'_k$.

All the horizontal and diagonal links finally connects β_0 and β_S, meaning that R_1 and R_2 return the same value in both executions.

However, according to our construction of the chains, R_1 and R_2 return different values in β_0 and β_S. This leads to contradiction, which finishes our impossibility proof.

4 FAST WRITE (W1R2): SIEVE-BASED CONSTRUCTION OF EXECUTIONS

Informally speaking, it is reasonable to think that the first round-trip of a read operation should not change the information stored on the servers, thus being not able to affect the return values of other read operations. It is because in the first round-trip, the reader knows nothing about what happens on the servers and other clients. It should not "blindly" affect the servers.

Following the intuition above, we prove that in our chain argument in Section 3, if $R_2^{(1)}$ affects certain servers, such servers cannot affect our chain argument. Thus we sieve all the servers and only those which actually decide the return values of R_1 remain. We restrict our chain argument in Section 3 to the remaining servers and can still obtain the contradiction.

Before sieving the servers, we need an abstract model which can capture the essence of the interaction between clients and servers in W1R2 implementations. Only with this abstract model can we discuss what the effect is when we say that the server is affected by the first round-trip of a read. In analogy, the role of this abstract model is like that of the decision-tree model, which is used to derive the lower bound of time complexity for comparison-based sorting algorithms [11]. We name this model the *crucial-info* model and present it in Section 4.1. With the crucial-info model, we discuss in Section 4.2 how we can eliminate the servers which have no effect on the return values of read operations. We further explain how our chain argument can be successfully conducted on servers that remain.

4.1 The Crucial-Info Model

We first present the *full-info* model, which is the basis for presenting the crucial-info model. When considering an atomic register implementation, we only care about the number of round-trips to complete a read or write operation. To this end, we use a *full-info* model, where the server is designed as an append-only log. The server just append everything it receives from the writers and readers in its log (never deleting any information). The clients can send arbitrary information to the servers. The clients can also arbitrarily modify the information stored on the servers. The server itself and the clients can always check the log to decide what data the server holds in any moment in the execution.

When the client queries information from the server, the server just replies the client with all the log it currently has. When the client obtains the full-info logs from multiple servers, it derives from the logs what to do next, .e.g. deciding a return value or issuing another round-trip of communication. Since we only care about the number of round-trips required in an implementation, we assume that the communication channel has sufficient bandwidth and the clients and servers have sufficient computing power. Implementations following this model are called full-info implementations.

This full-info model is for the theoretical analysis on the lower bound of the number of round-trips. Obviously, full-info implementations can be optimized to obtain practically efficient implementations. Since no implementation will use less round-trips than the full-info implementation, we only need to prove that there is no W1R2 full-info implementation of the atomic register. Based on the full-info model, we can refine certain *crucial information* the servers must maintain. Such crucial information must be stored, modified and disseminated among the clients and the servers, as long as the implementation is a correct atomic register implementation. Specifically, in the executions constructed in our impossibility proof (Section 3), when the writer writes the value "1" to the servers, the server must store the crucial information "1". Besides this crucial information, the server can store any auxiliary information it needs, but we are not concerned of such non-crucial information. In analogy, in comparison-based sorting, we only record which elements are compared and what the results are in the decision tree. Other information is not of our concern when deriving the lower bound of the time complexity of comparison-based sorting.

When two writers write "1" and "2", no matter what the temporal relation between the two write operations is, the server receives the crucial information in certain sequential order, and we store this crucial information as "12" or "21". In order to determine the return value, the reader collects the crucial information "12" or "21" from no less than $S - t$ servers. According to the definition of atomicity, the reader needs to infer the temporal relation between the two write operations W_1 and W_2. Then it can decide the return value. In executions we construct in our proof, the only possible relations between W_1 and W_2 are:

- Rel1: W_1 precedes W_2.
- Rel2: W_1 is concurrent with W_2.
- Rel3: W_2 precedes W_1.

In the executions in chain α, β, γ and $\mathbb{Z}$, there are two essential cases for the reader to decide a return value:

- If the reader cannot differentiate Rel1 (or Rel3) from Rel2, then it must return 2 (or 1).
- If two readers both see Rel2, they need to coordinate (through the servers) to make sure that they decide the same return value.

In other cases, the reader can obviously decide what it should return. Note that we only have client-server interaction, i.e., the servers do not communicate with other servers and the clients do not communicate with other clients.

Given the crucial-info model, we can now describe how the first round-trip of a read operation $R_i^{(1)}$ affects another read operation R_j. When $R_i^{(1)}$ affects R_j, $R_i^{(1)}$ must change the crucial information on some servers, while such modified crucial information is obtained by R_j. Note that R_j may be affected (i.e., the indistinguishability is broken) since the crucial information it obtains from the servers changes, but R_j could still decide the same return value even if the crucial information has changed.

In the executions in our proof, the reader only needs to derive the temporal relation between W_1 and W_2. The only crucial information that can be stored on the server is the temporal order between W_1

and W_2 the server sees. The possible values of the crucial information on the server are "12" and "21". The first round-trip of the reader can only affect the server by changing the crucial information from "12" to "21" or vise versa, as long as the implementation correctly guarantees atomicity.

Given the crucial-info model, we can explain how we sieve the serves, as well as how the chain argument can be successfully conducted after the affected servers are eliminated.

4.2 Eliminating the Affected Servers

We conduct the sieving when we append $R_2^{(1)}$ to executions in chain $\alpha = (\alpha_0, \alpha_1, \cdots, \alpha_S)$. From α_0, we append the second read operation R_2, and discuss the effect of $R_2^{(1)}$ on the return value of R_1. Now we have three non-concurrent round-trips $R_1^{(1)}$, $R_2^{(1)}$ and $R_1^{(2)}$, as shown in Phase 2 of Fig. 3. We are concerned of what happens to our chain argument proof (in Section 3) if $R_2^{(1)}$ may affect (the crucial information on) some servers and may potentially affect the return value of R_1 (more specifically, $R_1^{(2)}$), thus breaking the indistinguishability we try to construct.

Considering the effect of $R_2^{(1)}$, we partition all servers Σ_{sv} into two subsets Σ_1 and Σ_2, as shown in Fig. 8. Set Σ_1 contains all servers whose crucial information is affected by $R_2^{(1)}$, while Σ_2 contains all servers whose crucial information is not affected. Without loss of generality, we let $\Sigma_2 = \{s_1, s_2, \cdots, s_x\}$ and $\Sigma_1 = \{s_{x+1}, s_{x+2}, \cdots, s_S\}$. According to our construction of α_0, every server in Σ_2 contains crucial information "12". At first, the crucial information stored on servers in Σ_1 is also "12". According to the crucial-info model, the only effect on the server which can affect the return value of read operations is changing this "12" to "21". So after servers in Σ_1 are affected by $R_2^{(1)}$, their crucial info is changed from "12" to "21". We denote this execution where servers in S_1 are affected by $R_2^{(1)}$ as $\hat{\alpha}_0$.

In $\hat{\alpha}_0$, we have that R_1 must return 2. It is because W_1 precedes W_2 by construction, and in any correct atomic register implementation, read operations after W_2 should return 2. Whatever the effect of $R_2^{(1)}$ is, it should not prevent R_1 from returning 2. For the chain argument, we need to construct the other end of the chain. We still do the swapping one server a time. Execution $\hat{\alpha}_i$ is the same with $\hat{\alpha}_{i-1}$ except for s_i, for $1 \le i \le x$. The crucial information on s_i is "12" in $\hat{\alpha}_{i-1}$, while the crucial information on s_i is "21" in $\hat{\alpha}_i$. We do the swapping one server a time for all servers in Σ_2. The tail execution of chain is $\hat{\alpha}_{tail} = \hat{\alpha}_x$, as shown in Fig. 8. Note that the chain becomes "shorter". Servers in Σ_1 are unchanged, in all executions $\hat{\alpha}_0, \hat{\alpha}_1, \cdots, \hat{\alpha}_x$.

Now we describe the sieving process to eliminate servers in Σ_1 from our chain argument. As for execution $\hat{\alpha}_x$, consider the servers in Σ_1. They do the computation the same way they do in $\hat{\alpha}_0$, i.e., they first contain crucial info "12", then is affected by $R_2^{(1)}$ and change their crucial info to "21". Note that the effect of $R_2^{(1)}$ is "blind" effect because it does not obtain any information from the outside world first. The servers in Σ_1 and the reader r_2 of round-trip $R_2^{(1)}$ will not differentiate $\hat{\alpha}_x$ from $\hat{\alpha}_0$. Thus all servers in Σ_1 behave the same way in both executions, and they will have crucial info "21".

As for servers in Σ_2 in $\hat{\alpha}_x$, after W_1 and W_2, all servers in Σ_2 have crucial information "21". This crucial information should remain "21" after $R_1^{(1)}$ and $R_2^{(1)}$. Assume for contradiction that the crucial information on some server s_y in Σ_2 has been affected by $R_1^{(1)}$ and $R_2^{(1)}$, and is changed from "21" to "12". Combining the behavior of s_y in both $\hat{\alpha}_0$ and $\hat{\alpha}_x$, we find that s_y always end with crucial information "12" after $R_1^{(1)}$ and $R_2^{(1)}$, no matter what the write operations write on the servers. Such servers obviously cannot decide the return value of R_1 and can be safely eliminated. So we can assume that all servers in Σ_2 in $\hat{\alpha}_x$ have crucial information "21" after $R_1^{(1)}$ and $R_2^{(1)}$.

In this way, R_1 will see all servers have crucial info "21" in $\hat{\alpha}_x$, and R_1 must return 1 in execution $\hat{\alpha}_x$. We thus obtain the key property required for the chain argument: in two end executions of the chain $\hat{\alpha} = (\hat{\alpha}_0, \hat{\alpha}_1, \cdots \hat{\alpha}_x)$, R_1 return different values. Note that the length of the chain will not affect our chain argument in Section 3, as long as we have enough servers left for the chain argument. Operation R_1 uses crucial information only from servers in Σ_2. Crucial information on servers in Σ_1 have been affected, and the change in this crucial information will not affect that R_1 returns 2. Since $t = 1$ and servers in Σ_2 can enable a correct atomic register implementation (we have this assumption to derive the contradiction), we have at least 3 servers in Σ_2.

Another threat to clarify is that when constructing chain β', β'' and β, the chains are based on the swapping among all servers, i.e., chain β', β'' and β all have length $S + 1$ even after the sieving. That is to say, the sieving is only conducted on executions in chain α, in order to obtain the critical server s_{i_1}. This raises the potential threat that when constructing $\beta_0, \beta_1, \cdots, \beta_S$, what happens if $R_1^{(1)}$ affects the return value of R_2. Observe that in our proof, we only use the fact that, when R_2 (both round-trips) skips s_{i_1} in executions β'_S and β''_S, R_2 returns the same value. This means that, no matter what the effect of $R_1^{(1)}$ is, R_2 still returns the same values in both β'_S and β''_S, as long as the critical server is skipped. Thus our chain argument can successfully go on as in Section 3.

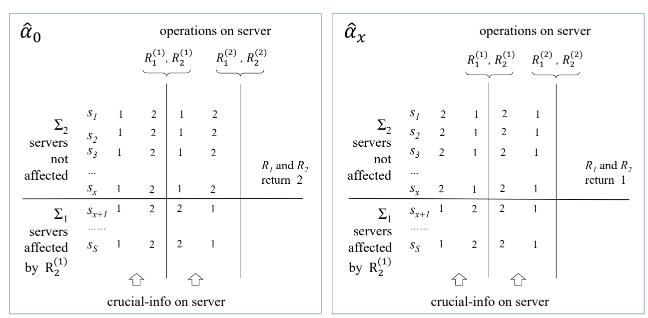

Figure 8: Eliminating servers affected by $R_2^{(1)}$.

5 FAST READ (W2R1): IMPOSSIBILITY AND IMPLEMENTATION

In this section, we discuss the impossibility and implementation of fast read (W2R1) multi-writer atomic registers. The necessary and sufficient condition of a W2R1 implementation is $R < \frac{S}{t} - 2$, which is the same with that of the single-writer case [12]. The impossibility proof and the algorithm design are also obtained by extending their counterparts in the single-writer case.

5.1 Impossibility when $R \geq \frac{S}{t} - 2$

We need to prove that it is impossible to obtain a W2R1 implementation when $R \geq \frac{S}{t} - 2$ in the multi-writer case. It is sufficient to prove that, even there is only one writer and this single writer can employ two round-trips, W2R1 implementations are still impossible.

The proof in the single writer-case does not depend on how many round-trips a write operation has. When we change all write operations in the impossibility proof in the single-writer case to two (or more) round-trips, we let all the two (or more) round-trips of a write operation take place consecutively and precede all other operations, as shown in Fig. 9 (based on Fig. 6 of [12]). The rest of the impossibility proof is not affected.

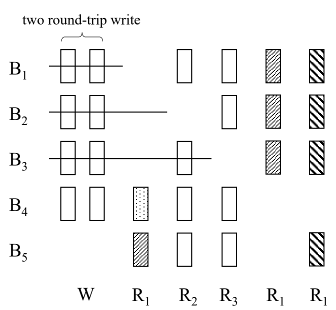

Figure 9: Fast read impossibility.

5.2 Implementation when $R < \frac{S}{t} - 2$

We derive the W2R1 implementation from the single-writer W1R1 implementation in [12]. Our implementation is inspired by how multiple writers are handled in the W2R2 implementation [23], which can also be viewed as a derivation from the single-writer W1R2 implementation [5]. The key change in the design of a multi-writer implementation is that we use (ts, w_i) to denote one value. Here w_i is the writer ID and ts is the version number denoting one value written by w_i. Assuming that the writer IDs are totally ordered, we can thus order all the values from multiple writers using the lexicographical order when we have equal ts values.

The order among write values is further strengthened by the two round-trip write algorithm. Specifically, before writing a value, the writer first queries all the servers and calculates the $maxTS$ in its first round-trip. Then the writer updates value $(maxTS + 1, w_i)$ to all servers in the second round-trip. The two-round-trip write algorithm guarantees that when write operations have the same ts value, they must be concurrent.

As for the the single round-trip read operations, the reader first obtains multiple values from the servers. It also uses the $admissible(\cdot)$ predicate (defined in the single-writer algorithm in [12]) to test all the values obtained. The $admissible(\cdot)$ predicate is designed to guarantee that: i) a read never returns older values than that of a preceding write, and that ii) a read never returns older values than that of a preceding read. Since there are multiple writers, the reader may obtain multiple admissible values and need to return one of them. Since all the values are totally ordered, we simply let the reader return the largest admissible value. That is, when the reader needs to choose from equal ts values, it just chooses the ts value with the largest writer ID.

One potential threat to the correctness of our algorithm is that in the single-writer case, all values are totally ordered on one single-writer and it is trivial to choose the more up-to-date value. However in the multi-writer case, the two round-trip algorithm can order non-concurrent write operations from different writers. But for the concurrent writes, we can only use the (somewhat arbitrary) order among writer IDs. We need to prove that using the writer ID order will not comprise the correctness of our implementation.

Specifically, for two read operations R_1 preceding R_2, the predicate $admissible(\cdot)$ guarantees that $S_{ad}(R_1) \subseteq S_{ad}(R_2)$ [12]. Here $S_{ad}(R_i)$ $(i = 1, 2)$ denotes the set of admissible values on R_i. Denote the return values of R_1 and R_2 as $val_1 = \max(S_{ad}(R_1))$ and $val_2 = \max(S_{ad}(R_1))$ respectively. The potential threat to our multi-writer implementation is that R_2 chooses a new return value only due to the difference in writer ID while the ts values are the same. Specifically, assume that $val_2 \neq val_1$, but $val_2.ts = val_1.ts$ and $val_2.writer\text{-}id > val_1.writer\text{-}id$. Since $val_2 \notin S_{ad}(r_1)$ (or R_1 and R_2 will choose the same return value), we have that val_2 is not admissible in R_1's view, but val_2 is admissible in R_2's view. This ensures that R_1 must be concurrent with W_2 (let W_i denote the write operation of val_i for $i = 1, 2$). Thus we have W_1, R_1, W_2, R_2 is a correct permutation of these operations as required by the definition of atomicity. This ensures that the return value of R_2 is correct.

For other cases, the correctness proof of our W2R1 implementation is principally the same with that of the W1R1 implementation in [12]. We present our W2R1 implementation and its detailed proof of correctness in the Appendix of [18]. Note that impossibility results in the crash failure model directly imply impossibility in the Byzantine failure model. However, for our W2R1 implementation, we can further study whether it can be extended to further tolerate Byzantine failures. The extension is principally the same with that in the single-writer case, as detailed in [12]. We thus omit detailed discussions here.

6 RELATED WORK

The importance of low latency data access in distributed storage systems motivates the study on fast implementations of distributed atomic registers. Fast implementation in the single-writer case is studied in [12], where the sufficient and necessary condition for fast implementation is derived. As for the multi-writer case, only impossibility for fast read-write implementations is presented. When examined at a finer granularity, it is still open whether fast implementations are possible when only read or write are required to be fast. The notion of semifast implementation is presented in [14]. It is proved that semifast implementation is not possible for multi-writer atomic registers. In this work, we consider implementations where the read can always be slow (using two or more round-trips). The implementation we consider is strictly stronger than semifast implementations. Thus our our impossibility proof is more general and directly implies the impossibility of semifast implementations.

This work concludes this series of studies on fast implementations of distributed atomic registers. Impossibility proof for fast write implementations is presented, and necessary and sufficient condition for fast read implementations is derived.

Our impossibility proof for fast write implementations are inspired by the classical result in a shared-memory setting that "atomic reads must write" [7, 20, 21]. The CAP theorem [8, 15] and the PARCELC tradeoff [3] in distributed systems also inspire us to prove the impossibility of fast (low latency, strongly consistent and fault-tolerant) implementations. Our use of the crucial-info model is inspired by the CHT proof of the weakest failure detector for consensus [10, 13]. In the CHT proof, a directed acyclic graph is used to store the failure detector outputs on all processes as well as the temporal relations between them.

The study on atomic register implementations on the Oh-RAM model is closely related to our work [16]. Both works use chain arguments [6] to prove the impossibility. The main difference lies in the system model. In the Oh-RAM model, servers are allowed to exchange messages , while in our client-server model, we only model communications between the client and the server. We derive our system model from the existing work [4, 5, 12, 23], as well as from our study of popular distributed storage systems [1, 2, 19].

7 CONCLUSION AND FUTURE WORK

In this work, we study fast write and fast read implementations of multi-writer atomic registers. For fast write implementations, we come up with the impossibility proof, which is based on a three-phase chain argument. For fast read implementations, we provide the necessary and sufficient condition for fast implementations, by extending the result of the single-writer case.

In our future work, we will study fast implementations for multi-writer atomic registers from a different perspective. Specifically, we will fix fast implementations in the first place, and then quantify how much data inconsistency will be introduced when strictly guaranteeing atomicity is impossible. We also plan to introduce knowledge calculus to reason about quorum-based distributed algorithms at a higher level of abstraction.

ACKNOWLEDGMENTS

This work is supported by the National Key R&D Program of China (No. 2017YFB1001801), the National Natural Science Foundation of China (No. 61772258), and the Fundamental Research Funds for the Central Universities (No. 14380063).

REFERENCES

[1] 2020. Redis distributed data structure store. http://redis.io/.
[2] 2020. Riak distributed database. https://riak.com/.
[3] Daniel J. Abadi. 2012. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *Computer* 45, 2 (Feb 2012), 37–42. https://doi.org/10.1109/MC.2012.33
[4] James Aspnes. 2019. *Notes on Theory of Distributed Systems.* Yale University, CPSC 465/565.
[5] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing Memory Robustly in Message-passing Systems. *J. ACM* 42, 1 (Jan. 1995), 124–142. http://doi.acm.org/10.1145/200836.200869
[6] Hagit Attiya and Faith Ellen. 2014. *Impossibility Results for Distributed Computing.* Morgan & Claypool.
[7] Hagit Attiya and Jennifer Welch. 2004. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* John Wiley & Sons.
[8] Eric A. Brewer. 2000. Towards Robust Distributed Systems (Abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing* (Portland, Oregon, USA) *(PODC'00)*. ACM, New York, NY, USA, 7–. https://doi.org/10.1145/343477.343502
[9] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. ACM, New York, NY, USA, 271–284. https://doi.org/10.1145/2535838.2535848
[10] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. 1996. The Weakest Failure Detector for Solving Consensus. *J. ACM* 43, 4 (July 1996), 685–722. https://doi.org/10.1145/234533.234549
[11] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Ctein. 2009. *Introduction to Algorithms (third edition).* the MIT Press.
[12] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Marko Vukolić. 2010. Fast Access to Distributed Atomic Memory. *SIAM J. Comput.* 39, 8 (Dec. 2010), 3752–3783. https://doi.org/10.1137/090757010
[13] Felix C. Freiling, Rachid Guerraoui, and Petr Kuznetsov. 2011. The Failure Detector Abstraction. *ACM Comput. Surv.* 43, 2, Article 9 (Feb. 2011), 40 pages. https://doi.org/10.1145/1883612.1883616
[14] Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. 2009. Fault-tolerant semifast implementations of atomic read/write registers. *J. Parallel and Distrib. Comput.* 69, 1 (2009), 62 – 79. https://doi.org/10.1016/j.jpdc.2008.05.004
[15] Seth Gilbert and Nancy A. Lynch. 2012. Perspectives on the CAP Theorem. *Computer* 45, 2 (2012), 30–36. https://doi.org/10.1109/MC.2011.389
[16] Theophanis Hadjistasi, Nicolas Nicolaou, and Alexander A. Schwarzmann. 2017. Oh-RAM! One and a Half Round Atomic Memory. In *Networked Systems*, Amr El Abbadi and Benoît Garbinato (Eds.). Springer International Publishing, Cham, 117–132.
[17] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12 (July 1990), 463–492. Issue 3. http://doi.acm.org/10.1145/78969.78972
[18] Kaile Huang, Yu Huang, and Hengfeng Wei. 2020. *Fine-grained Analysis on Fast Implementations of Multi-writer Atomic Registers.* Technical Report. Institute of Computer Software, Nanjing University. https://arxiv.org/abs/2001.07855
[19] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922
[20] Leslie Lamport. 1986. On interprocess communication. Part I: Basic formalism. *Distributed Computing* 1, 2 (1986), 77–85. http://dx.doi.org/10.1007/BF01786227
[21] Leslie Lamport. 1986. On interprocess communication. Part II: Algorithms. *Distributed Computing* 1, 2 (1986), 86–101. http://dx.doi.org/10.1007/BF01786228
[22] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL) *(NSDI'13)*. USENIX Association, Berkeley, CA, USA, 313–328. http://dl.acm.org/citation.cfm?id=2482626.2482657
[23] N. A. Lynch and A. A. Shvartsman. 1997. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*. 272–281. https://doi.org/10.1109/FTCS.1997.614100
[24] Henrique Moniz, João Leitão, Ricardo J. Dias, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2017. Blotter: Low Latency Transactions for Geo-Replicated Storage. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia) *(WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 263–272. https://doi.org/10.1145/3038912.3052603
[25] Lingzhi Ouyang, Yu Huang, Hengfeng Wei, and Jian Lu. 2019. *Enabling Almost Strong Consistency for Quorum-replicated Datastores*. Technical Report. Institute of Computer Software, Nanjing University. https://github.com/Lingzhi-Ouyang/Almost-Strong-Consistency-Cassandra/blob/master/document/almost-strong-consistency.pdf
[26] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 413–424. https://doi.org/10.1145/2737924.2737981
[27] Paolo Viotti and Marko Vukoliundefined. 2016. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.* 49, 1, Article 19 (June 2016), 34 pages. https://doi.org/10.1145/2926965
[28] Hengfeng Wei, Yu Huang, and Jian Lu. 2017. Probabilistically-Atomic 2-Atomicity: Enabling Almost Strong Consistency in Distributed Storage Systems. *IEEE Trans. Comput.* 66, 3 (March 2017), 502–514. https://doi.org/10.1109/TC.2016.2601322