Appendix to "Achieving Probabilistic Atomicity with Well-Bounded Staleness and Low Read Latency in Distributed Datastores"

Lingzhi Ouyang, Yu Huang, Hengfeng Wei, and Jian Lu

1 Monotonicity Properties of Shared Register Emulation Algorithms

In this section, we study the monotonicity properties of quorum-based algorithms theoretically. For simplicity, we assume that all operations in the history are applied to the same register. Some conventions in this appendix are specified as follows:

- σ: the execution history of the clients accessing the shared register.
- π: the linear extension / permutation of operations in σ.
- $\mathbb{W}$: the set of write operations.
- $\mathbb{R}$: the set of read operations.
- $\mathbb{O} = \mathbb{W} \cup \mathbb{R}$: the set of operations including writes and reads.
- $[o_{st}, o_{ft}]$: the time interval of the operation o. The time of invocation event and response event of o are denoted o_{st} and o_{ft} separately.
- $ver(o)$: the version attached to the returned value if operation $o \in \mathbb{R}$, or the version attached to the written value if $o \in \mathbb{W}$.
- $ver(s,t)$: the version stored in the server s at time t.
- $maxVer(A,t)/minVer(A,t)$: the maximum / minimum version stored in the server set A at a specific time t.
- $maxVer(A)/minVer(A)$: the maximum / minimum version collected in a query, where A is the responding server set of the query. Note that each server in A may respond at different time.

The pseudo-code for client-server interaction appears in Algorithm 1. It can be easily proved that:

Lemma 1.1 (Query-after-Update Property). *For any update u and any query q, if $u \prec_\sigma q$, and q receives responses from the server set Q, then $maxVer(Q) \geq ver(u)$.*

Proof. According to the server procedures in Algorithm 1, the version stored in servers keep non-decreasing as time goes by. Therefore, for any update u, when it is finished at time t_1, there exist a set of servers, denoted U, storing replicas with the version no lower than $ver(u)$. Namely, $minVer(U,t_1) \geq ver(u)$.

Now we consider any query q that starts after u. Namely, $u \prec_\sigma q$. Let Q denote the server set that q receives responses from.

According to the client procedures in Algorithm 1, any query or update will only be finished until a *majority* of servers reply, so $U \cap Q \neq \emptyset$. Assume the server $s \in U \cap Q$, and s returns the version at t_2 in the query q. Since q starts after u, then $t_2 > t_1$. Therefore, $maxVer(Q) \geq ver(s,t_2) \geq ver(s,t_1) \geq minVer(U,t_1) \geq ver(u)$. □

The pseudo-codes for read/write operations appear in Algorithm 2 & 3. Here, we mainly focus on the properties in terms of monotonicity of quorum-based algorithms, as shown in Table 1. These properties depict the relations between the temporal real-time order and the semantic read-from order of operations. The detailed proofs are mainly based on the *Query-after-Update Property* and displayed in Section 1.1 - 1.4 respectively.

TABLE 1
Algorithm properties in terms of monotonicity

Properties	W2R2	W2R1	W1R2	W1R1
$w \prec_\sigma r \Rightarrow ver(w) \leq ver(r)$	√	√	√	√
$w \prec_\sigma w' \Rightarrow ver(w) < ver(w')$	√	√	×	×
$r \prec_\sigma r' \Rightarrow ver(r) \leq ver(r')$	√	×	√	×
$r \prec_\sigma w \Rightarrow ver(r) < ver(w)$	√	×	×	×

1.1 Common Property: Write-Read Monotonicity

The quorum-based write / read algorithms with one or two communication round-trips guarantee the *write-read monotonicity* for multi-writer registers. Specifically,

Theorem 1.1 (Write-Read Monotonicity). *In the execution history σ, for any operations w, r using the quorum-based algorithms(see Algorithm 1, 2, 3), where $w \in \mathbb{W}, r \in \mathbb{R}$, if $w \prec_\sigma r$, then $ver(w) \leq ver(r)$.*

- *Corresponding author: Yu Huang, State Key Laboratory for Novel Software Technology, and Department of Computer Science and Technology, Nanjing University, China, 210023.*
  E-mail: yuhuang@nju.edu.cn.
- *Lingzhi Ouyang, Hengfeng Wei and Jian Lu are with the State Key Laboratory for Novel Software Technology, and Department of Computer Science and Technology, Nanjing University, China, 210023.*
  E-mail: lingzhi.ouyang@outlook.com, hfwei@nju.edu.cn, lj@nju.edu.cn.

Algorithm 1: Client-server interaction

```
▷ Code for client process $p_i (0 \le i \le n-1)$:
function query($key$)
    $vals \leftarrow \varnothing$
    pfor each server $s_j$                ▷ pfor: parallel for
        send $['query', key]$ to $s_j$
        $v \leftarrow [key, val, ver]$ from $s_j$
        $vals \leftarrow vals \cup v$
    until a majority of them respond
    return $vals$
function update($key, value, version$)
    $vals \leftarrow \varnothing$
    pfor each server $s_j$
        send $['update', key, value, version]$ to $s_j$
    wait for $['ACK']$s from a majority of them
▷ Code for server process $s_i (0 \le i \le N-1)$:
upon receive $['query', key]$ from $p_j$
    send $['query-back', key, val, ver]$ to $p_j$
upon receive $['update', key, value, version]$ from
  $p_j$
    pick $[k, val, ver]$ with $k == key$
    if $ver < version$ then
        $val \leftarrow value$
        $ver \leftarrow version$
    send $['ACK']$ to $p_j$
```

Algorithm 2: Write algorithms for client p_i

```
procedure TwoRoundWRITE($key, value$)
    $replicas \leftarrow$ query($key$)
    $version \leftarrow$(maxSeq($replicas$)+1, $i$)
    update($key, value, version$)
procedure OneRoundWRITE($key, value$)
    $localSeq[key] \leftarrow localSeq[key]$+1
    $version \leftarrow (localSeq[key], i)$
    update($key, value, version$)
```

Algorithm 3: Read algorithms for client p_i

```
procedure TwoRoundREAD($key$)
    $replicas \leftarrow$ query($key$)
    $version \leftarrow$maxVer($replicas$)
    $value \leftarrow$valWithMaxVer($replicas, version$)
    $localSeq[key] \leftarrow version.seq$        ▷ Optional.
    update($key, value, version$)
    return $value$
procedure OneRoundREAD($key$)
    $replicas \leftarrow$ query($key$)
    $version \leftarrow$maxVer($replicas$)
    $value \leftarrow$valWithMaxVer($replicas, version$)
    $localSeq[key] \leftarrow version.seq$        ▷ Optional.
    return $value$
```

Proof. $\forall w \in \mathbb{W}, r \in \mathbb{R}$, let u denote the update period of w, q denote the query period of r, and Q denote the server set that q receives responses from. Then $ver(w) = ver(u), ver(r) = maxVer(Q)$.

If $w \prec_\sigma r$, then, it is trivial to have $u \prec_\sigma q$. By Lemma 1.1, $maxVer(Q) \ge ver(u)$. Therefore, $ver(w) = ver(u) \le maxVer(Q) = ver(r)$. □

1.2 Two-Round-Trip Write Property: Write-Write Monotonicity

The two-round-trip write algorithm guarantees the *write-write monotonicity* for multi-writer registers. Specifically,

Theorem 1.2 (Write-Write Monotonicity). *In the execution history σ, for any write operations w_1, w_2 using the quorum-based algorithm that completes each write operation in 2 communication round-trips (see TwoRoundWRITE in Algorithm 2), if $w_1 \prec_\sigma w_2$, then $ver(w_1) < ver(w_2)$.*

Proof. For two-round-trip write operations w_1, w_2, let u_1 denote the update period of w_1, q_2 denote the query period of w_2, and Q_2 denote the server set that q_2 receives responses from. Then $ver(w_1) = ver(u_1), ver(w_2) > maxVer(Q_2)$ (Note: w_2 will construct a larger version according to the largest version in Q_2).

If $w_1 \prec_\sigma w_2$, then, it is trivial to have $u_1 \prec_\sigma q_2$. By Lemma 1.1, $maxVer(Q_2) \ge ver(u_1)$. Therefore, $ver(w_1) = ver(u_1) \le maxVer(Q) < ver(w_2)$. □

In comparison, the one-round-trip write algorithm (see *OneRoundWRITE* in Algorithm 2) can only guarantee the *write-write monotonicity* for *single-writer* registers, where the only writer can execute updates with *monotonically increasing local versions*. However, for *multi-writer* registers, one-round-trip write may result in *write inversion anomalies after a write*, which means that a *previous* write assigns a *larger* version for its written value than a *later* write (Namely, $(w_1 \prec_\sigma w_2) \cap (ver(w_1) > ver(w_2))$). What's worse, the procedure without the promise of assigning a version larger than previous writes may lead to unsuccessful updates on servers. More details will be discussed in Section 3 in this appendix.

1.3 Two-Round-Trip Read Property: Read-Read Monotonicity

The two-round-trip read algorithm guarantees the *read-read monotonicity* for multi-writer registers. Specifically,

Theorem 1.3 (Read-Read Monotonicity). *In the execution history σ, for any read operations r_1, r_2 using the quorum-based algorithm that completes each read operation in 2 communication round-trips (see TwoRoundREAD in Algorithm 3), if $r_1 \prec_\sigma r_2$, then $ver(r_1) \le ver(r_2)$.*

Proof. For two-round-trip read operations r_1, r_2, let u_1 denote the update period of r_1, q_2 denote the query period of r_2, and Q_2 denote the server set that q_2 receives responses from. Then $ver(r_1) = ver(u_1), ver(r_2) = maxVer(Q_2)$.

If $r_1 \prec_\sigma r_2$, then, it is trivial to have $u_1 \prec_\sigma q_2$. By Lemma 1.1, $maxVer(Q_2) \ge ver(u_1)$. Therefore, $ver(r_1) = ver(u_1) \le maxVer(Q) = ver(r_2)$. □

In comparison, the one-round-trip read algorithm (see *OneRoundREAD* in Algorithm 3) can guarantee the *read-read monotonicity* for *single-reader* registers only if clients store previous read records locally. However, for *multi-reader* registers, the one-round-trip read algorithm may lead to *read / write inversion anomalies after a read*, which means that the version of a later operation (a read or a write) is smaller than the version of a *previous* read. The anomalies involve several patterns and more details will be discussed in Section 2 in this appendix.

1.4 Exclusive Property of W2R2: Read-Write Monotonicity

Besides all above properties, the W2R2 algorithm also guarantees the *read-write monotonicity* for MWMR registers.

Theorem 1.4 (Read-Write Monotonicity). *In the execution history σ, for any operations w, r using the W2R2 algorithm, where $w \in \mathbb{W}, r \in \mathbb{R}$, if $r \prec_\sigma w$, then $ver(r) < ver(w)$.*

Proof. $\forall w \in \mathbb{W}, r \in \mathbb{R}$, let u_r denote the update period of r, q_w denote the query period of w, and Q_w denote the server set that q receives responses from. Then $ver(r) = ver(u_r), ver(w) > maxVer(Q_w)$ (Note: w will construct a larger version according to the largest version in Q_w).

If $w \prec_\sigma r$, then, it is trivial to have $u_w \prec_\sigma q_w$. By Lemma 1.1, $maxVer(Q_w) \geq ver(u_r)$. Therefore, $ver(r) = ver(u_r) \leq maxVer(Q_w) < ver(w)$. □

2 Atomicity Violation of W2R1: Staleness and Probability

In this section, we first prove the possible atomicity violation patterns in W2R1 (see Algorithm 4), then we prove the bound of data staleness and calculate the probability of atomicity violation in W2R1 based on the violation patterns.

Algorithm 4: W2R1

```
Code for client process $p_i (0 \leq i \leq n-1)$:
procedure TwoRoundWRITE($key, value$)
    $replicas \leftarrow$ query($key$)
    $version \leftarrow$(maxSeq($replicas$)+1,$i$)
    update($key, value, version$)
procedure OneRoundREAD($key$)
    $replicas \leftarrow$ query($key$)
    $version \leftarrow$maxVer($replicas$)
    $value \leftarrow$valWithMaxVer($replicas, version$)
    return $value$
```

2.1 Proof of Atomicity Violation Patterns

We aim to prove that atomicity violation incurred in W2R1 are composed of RI or WI. When clients read and write the shared register using the W2R1 algorithm and obtain a history σ, we have that:

Theorem 2.1. *If σ violates atomicity, then there exist some operations in σ that form either RI or WI.*

Proof. If σ violates atomicity, then for any permutation π of σ, we have a stale read r, the dictating write w of r and the interfering write w' satisfying: $w \prec_\pi w' \prec_\pi r$. Then, we exhaustively check all cases that make r a stale read.

According to the definition of atomicity, two types of relations between operations are of our concern: the temporal real-time relation and the semantic read-from relation. We first enumerate all possible cases according to the semantic relation. Then for each case, we further enumerate all possible sub-cases according to the temporal relation.

Since w is the dictating write of r, we have that $ver(r) = ver(w)$. According to the semantic relation between versions of w and w', we have two complementing cases: $ver(r) = ver(w) < ver(w')$ and $ver(r) = ver(w) > ver(w')$. In each case, we then consider the temporal relation between operations.

Case 1: $ver(r) = ver(w) < ver(w')$.

Note that r *must be concurrent with* w'. Namely, $r \parallel_\sigma w'$. This can be proved by contradiction. If $r \prec_\sigma w'$, w' can never be the interfering write of r. If $w' \prec_\sigma r$, according to the *write-read monotonicity* (Theorem 1.1), r must return the version of w' (or return an even larger version), contradicting the fact that $ver(r) = ver(w) < ver(w')$.

Next we exhaustively check all possible sub-cases according to the temporal relation to prove that: *there must exist a read operation r' that reads from w', and r' precedes r.* Namely, $\exists r' = R(w') : r' \prec_\sigma r$.

We first enumerate all possible cases according to the temporal relation between w and w'. Since in the permutation π, $w \prec_\pi w'$, we have that in σ, $w \prec_\sigma w'$ or $w || _\sigma w'$.

Case 1.1: $w \prec_\sigma w'$. The permutation between w and w' must be *determined* as $w \prec_\pi w'$. Then, we enumerate all cases as follows:

× $\nexists r' = R(w')$. Note that $w' \parallel_\sigma r$, so r can be ordered before w'. Thus, the permutation among w, w' and r must be $w \prec_\pi r \prec_\pi w'$, which contradicts the permutation $w \prec_\pi w' \prec_\pi r$.

× $\forall r' = R(w') : r' \nprec_\sigma r$. Similarly, the permutation among w, w' and r must be $w \prec_\pi r \prec_\pi w'$, contradicting the permutation $w \prec_\pi w' \prec_\pi r$.

✓ $\exists r' = R(w') : r' \prec_\sigma r$. Then the permutation among w, w' and r must be $w \prec_\pi w'(\prec_\pi r') \prec_\pi r$ (see Fig. 1 (a)).

Case 1.2: $w \parallel_\sigma w'$. In this case, the permutation between w and w' can not be *determined* only by the temporal relation of these two write operations.

Therefore, we then focus on other operations related to these two clusters [1]. We first consider the temporal relation between w' and any read operation r'' that reads from w. Notice that r is a special instance of r'', thus r'' must exist.

Similarly, by contradiction we have $w' \nprec_\sigma r''$; otherwise r'' must return the version of w' (or return an even larger version) according to the *write-read monotonicity* (Theorem

1. Here, we use the concept *cluster*, a terminology introduced by Gibbons and Korach [1], for our analysis. A *cluster* is a subset of operations in an execution history that consists of a write and all of its dictated reads. In [1] , Gibbons and Korach proved that atomicity violation indicates certain patterns related to different zones of clusters, where all the dictated reads of the related writes may impact the permutation. Therefore, for the soundness of proof, atomicity violation and the permutation concerned with w, w', r should be analyzed without missing other operations in the related clusters, i.e. all dictated reads of w besides r, as well as all dictated reads of w'.

1.1). Thus, the temporal relation between w' and r'' can only be $r'' \prec_\sigma w'$ or $r'' \parallel_\sigma w'$.

Case 1.2.1: $\exists r'' = R(w) : r'' \prec_\sigma w'$. By the temporal relation between r'' and w' , as well as read-from order between r'' and w, the permutation between w and w' must be *determined* as $w(\prec_\pi r'') \prec_\pi w'$.

- × $\nexists r' = R(w')$. Note that $w' \parallel_\sigma r$, so r can be ordered before w'. Thus, the permutation among w, w' and r must be $w \prec_\pi r \prec_\pi w'$, contradicting the permutation $w \prec_\pi w' \prec_\pi r$.
- × $\forall r' = R(w') : r' \nprec_\sigma r$. Similarly, the permutation among w, w' and r must be $w \prec_\pi r \prec_\pi w'$, contradicting the permutation $w \prec_\pi w' \prec_\pi r$.
- ✓ $\exists r' = R(w') : r' \prec_\sigma r$. Then the permutation must be $w \prec_\pi w'(\prec_\pi r') \prec_\pi r$ (see Fig. 1 (b)).

Case 1.2.2: $\forall r'' = R(w) : r'' \parallel_\sigma w'$. Then, We consider the temporal relation between w and any read operation r' that reads from w'. Note that r' may not exist.

Case 1.2.2.1: $\nexists r' = R(w')$. Then, there exists no determining factor to deciding the permutation between w and w'. Thus, the permutation among w, w', r is $w' \prec_\pi w \prec_\pi r$ or $w \prec_\pi r \prec_\pi w'$, both contradict the assumption that w, w' and r form an inconsistent read. [×]

Case 1.2.2.2: $\exists r' = R(w') : r' \prec_\sigma w$. By the temporal relation between r' and w , as well as read-from order between r' and w', the permutation between w and w' is *determined* as $w'(\prec_\pi r') \prec_\pi w$, which contradicts $w \prec_\pi w'$. [×]

Case 1.2.2.3: $\forall r' = R(w') : r' \nprec_\sigma w$. Then, there exists no determining factor to deciding the permutation between w and w'.

- × $\forall r' = R(w') : r' \nprec_\sigma r$. Thus, the permutation among w, w' and r is $w' \prec_\pi w \prec_\pi r$ or $w \prec_\pi r \prec_\pi w'$, both violating the given permutation $w \prec_\pi w' \prec_\pi r$.
- ✓ $\exists r' = R(w') : r' \prec_\sigma r$. Then the permutation among w, w', r and r' is either $w \prec_\pi w' \prec_\pi r' \prec_\pi r$ or $w' \prec_\pi w \prec_\pi r' \prec_\pi r$ (see Fig. 1 (c)), which involves atomicity violation inevitably. If the former permutation is selected, then r is a stale read [2].

Above all, for $ver(w) < ver(w')$, then w, w' and r can be linearly extended to $w \prec_\pi w' \prec_\pi r$ only when $\exists r' = R(w') : r' \prec_\sigma r$. Obviously, r and r' form RI.

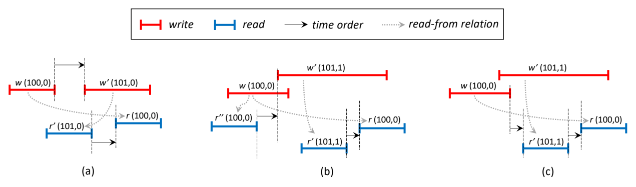

Fig. 1. Typical patterns of RI in Case 1.

Case 2: $ver(r) = ver(w) > ver(w')$.

Note that w must be concurrent with w'. This can be proved by contradiction. If $w \prec_\sigma w'$, according to *write-write monotonicity* that the two-round-trip write algorithm holds (Theorem 1.2), the version of w' must be larger. If $w' \prec_\sigma w$, we can never get $w \prec_\pi w' \prec_\pi r$ when linearly extending σ to π. Besides, according to the permutation $w' \prec_\pi r$, we have $w' \prec_\sigma r$ or $w' \parallel_\sigma r$.

Next we enumerate all possible sub-cases according to the temporal relation to prove that: there exists a read operation r'' (can be r) that reads from w, satisfying that r'' precedes either w' or another read operation r' that reads from w'. Namely, $\exists r'' = R(w) : (r'' \prec_\sigma w') \vee (\exists r' = R(w') : r'' \prec_\sigma r')$.

We first consider the temporal relation between w' and any read operation r'' that reads from w. Notice that r is a special instance of r'', thus r'' must exist.

Case 2.1: $\exists r'' = R(w) : r'' \prec_\sigma w'$. Then, the permutation between w and w' must be *determined* as $w(\prec_\pi r'') \prec_\pi w'$. Note that r cannot take the role of r'' here, because $r \nprec_\sigma w'$.

Case 2.1.1: $w' \prec_\sigma r$. Then the permutation among w, w' and r must be $w \prec_\pi w' \prec_\pi r$ (see Fig. 2 (a)). [✓]

Case 2.1.2: $w' \parallel_\sigma r$.

- × $\nexists r' = R(w')$. Note that $w' \parallel_\sigma r$, so r can be ordered before w'. Thus, the permutation among w, w' and r is $w \prec_\pi r \prec_\pi w'$, contradicting the assumption.
- × $\forall r' = R(w') : r' \nprec_\sigma r$. Similarly, the permutation among w, w' and r is $w \prec_\pi r \prec_\pi w'$, contradicting the assumption.
- ✓ $\exists r' = R(w') : r' \prec_\sigma r$. Then the permutation among w, w' and r must be $w(\prec_\pi r'') \prec_\pi w'(\prec_\pi r') \prec_\pi r$ (see Fig. 2 (b)).

Case 2.2: $\forall r'' = R(w) : r'' \nprec_\sigma w'$. Then, We consider the temporal relation between w and any read operation r' that reads from w'. Note that r' may not exist.

Case 2.2.1: $\nexists r' = R(w')$. Then, there exists no determining factor to deciding the permutation between w and w'.

- × $w' \prec_\sigma r$. Then the permutation among w, w' and r must be $w' \prec_\pi w \prec_\pi r$, contradicting the assumption.
- × $w' \parallel_\sigma r$. Thus, the permutation among w, w', r is $w' \prec_\pi w \prec_\pi r$ or $w \prec_\pi r \prec_\pi w'$, both contradicting the assumption.

Case 2.2.2: $\exists r' = R(w') : r' \prec_\sigma w$. Then the permutation between w and w' is *determined* as $w'(\prec_\pi r') \prec_\pi w$,which contradicts $w \prec_\pi w'$. [×]

Case 2.2.3: $\forall r' = R(w') : r' \nprec_\sigma w$. Then, We consider the temporal relation between any read operation r'' that reads from w and any read operation r' that reads from w'.

Case 2.2.3.1: $\forall r' = R(w'), r'' = R(w) : r'' \nprec_\sigma r'$.

- × $w' \prec_\sigma r$. Then the permutation among w, w' and r must be $w' \prec_\pi w \prec_\pi r$, contradicting the assumption.
- × $w' \parallel_\sigma r$. Thus, the permutation among w, w', r is either $w' \prec_\pi w \prec_\pi r$ or $w \prec_\pi r \prec_\pi w'$, both contradicting the assumption.

Case 2.2.3.2: $\exists r' = R(w'), r'' = R(w) : r'' \prec_\sigma r'$.

By contradiction we have $w \nprec_\sigma r'$; otherwise r' must return the version of w (or return an even larger version) according to the *write-read monotonicity* (Theorem 1.1). Besides, r'' cannot precede its dictating write w due to the regularity property. Thus, the temporal relation can only be $w \parallel_\sigma r''$ as well as $w \parallel_\sigma r'$. The permutation among w, r'', r' must be $w \prec_\pi r'' \prec_\pi r'$. Then the permutation can be $w \prec_\pi w' \prec_\pi r'' \prec_\pi r'$ or $w' \prec_\pi w \prec_\pi r'' \prec_\pi r'$, which involves atomicity violation inevitably. If $w' \prec_\sigma r$ (or r just

2. In this subcase (as well as **Case 2.2.3.2**), only one of the possible permutations will make r a stale read. However, we aim to prove the necessary conditions of atomicity violation in the proof, thus this possibility needs to be counted.

takes the role of r''), then r may become a stale read [3] (see Fig. 2 (c)). [✓]

Above all, for $ver(w) > ver(w')$, then w, w' and r can be linearly extended to $w \prec_\pi w' \prec_\pi r$ only when there exists a read operation r'' (can be r) that reads from w, satisfying that r'' precedes either w' or another read operation r' that reads from w'. Namely, $\exists r'' = R(w) : (r'' \prec_\sigma w') \vee (\exists r' = R(w') : r'' \prec_\sigma r')$, where r'' and w' form WI, or r'' and r' form RI.

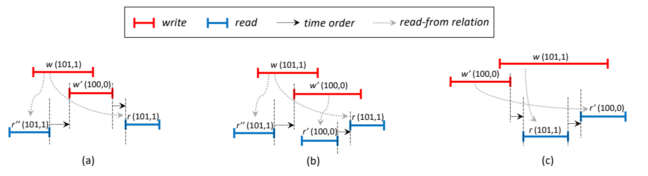

Fig. 2. Typical patterns of WI (Sub-fig (a) and (b))and RI (Sub-fig (c)) in Case 2.

In conclusion, by assuming r is a stale read, we exhaustively check all cases according to the temporal real-time relations and the semantic read-from relations of relative operations, At last, we have proved that the necessary conditions to make r a stale read involve RI or WI. Therefore, Theorem 2.1 is true. □

From the proof of Theorem 2.1 we can obtain some more detailed information, which is useful for our further analysis of staleness bound.

Proposition 2.1. *Suppose σ is the history obtained by clients using the W2R1 algorithm. If σ violates atomicity, then for any permutation π of σ, there exists a stale read r, the dictating write w of r and the interfering write w' that form $w \prec_\pi w' \prec_\pi r$, satisfying **Case 1** or **Case 2**, where*

- ***Case 1:*** *$ver(w) < ver(w')$. w' and r are concurrent, and there exists a read r' that reads from w', as well as precedes r. Namely, $(w' \parallel_\sigma r) \wedge (\exists r' = R(w') : r' \prec_\sigma r)$;*
- ***Case 2:*** *$ver(w) > ver(w')$. w and w' are concurrent, and there exists a read r'' (can be r) that reads from w, satisfying that r'' precedes either w' or another read r' that reads from w'. Namely, $(w \parallel_\sigma w') \wedge (\exists r'' = R(w) : (r'' \prec_\sigma w') \vee (\exists r' = R(w') : r'' \prec_\sigma r'))$.*

2.2 Proof of Bound of Data Staleness

Here we calculate the tight bound of data staleness when accessing the shared register using the W2R1 algorithm. Suppose the distributed storage system consists of N replicas($N \geq 3$). Denote the number of writer clients as n_w. We have that:

Theorem 2.2. *For any history σ, there exists a linear extension π of σ such that any read in π returns the value written by one of the latest B preceding writes. Here, $B = n_w + \frac{1}{2}n_w(n_w - 1) + 1$. Moreover, the bound B is tight, i.e., there exists a history σ and a linear extension π of σ in which some read returns the value written by the oldest write in the latest B preceding writes.*

3. **Case 2.2.3.2** and **Case 1.2.2.3** are nearly the same except the version relation between the clusters of w and w' is exchanged. However, in the proof we are not supposed to miss any possible subcase.

Proof. The proof of Theorem 2.2 is based on an adversary argument, to insert as many interfering writes as possible before the inconsistent read. The proof also needs to consider the same two cases as defined in Section 2.1. Specifically, suppose r is an inconsistent read, and the dictating write of r is w. There must exist another interfering write w' such that $w \prec_\pi w' \prec_\pi r$.

According to the versions $ver(r)$ and $ver(w')$, we consider two cases. In Case 1 where $ver(r) = ver(w) < ver(w')$, we prove that there are at most $B_1 = n_w$ interfering writes which can be inserted between r and w. In Case 2 where $ver(r) = ver(w) > ver(w')$, we prove that there are at most $B_2 = \frac{1}{2}n_w(n_w - 1)$ interfering writes.

Case 1: $ver(w) = ver(r) < ver(w')$.

From Case 1 of Proposition 2.1 we know that the interfering write w' must be concurrent with r, and there exists a read operation r' preceding r and dictated by w'. Note that the invocation of w' must be *earlier* than that of r, or r' is not able to read from w' when preceding r . Since each writer client can have at most one write operation that starts before r and is concurrent with r, the number of interfering writes in this case is no more than n_w. Thus we have the bound $B_1 = n_w$ in this case, and the bound B_1 is tight. One construction of the most stale read in Case 1 is displayed in Fig. 3, where the pattern of operations is shown in Fig. 1 (a).

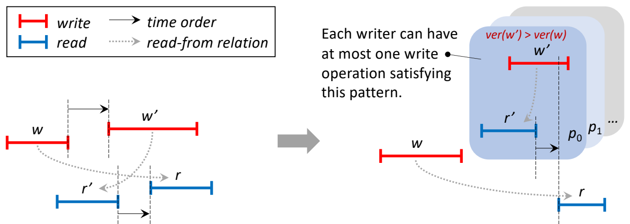
write
read
time order
read-from relation
w
w'
r'
r
Each writer can have at most one write operation satisfying this pattern.
ver(w') > ver(w)
w'
r'
p0
p1
...
w
r

Fig. 3. Maximum number of interfering writes in Case 1.

Case 2: $ver(w) = ver(r) > ver(w')$.

From Case 2 of Proposition 2.1 we know that the interfering write w' must be concurrent with w. Other than the writer client of operation w, we can have at most $n_w - 1$ writer clients to generate interfering writes. The challenge in the construction is that, for each interfering writer client, we cannot insert as many interfering write operations w' as we want, due to the constraint of $ver(w') < ver(w)$ and $w' \parallel_\sigma w$.

Denote the write operation versioned (seq, i) as $w(seq, i)$. In this case, both WI and RI are able to coexist to result in stale reads, we need to consider both patterns in our adversary argument. We first construct a simplified worst-case trace using WI alone, then we take both patterns into consideration.

The construction of WI without RI

In WI, there exists a read r'' that reads from $w(seq, i)$ and precedes w'. We focus on the maximum number of interfering write operations w' that can be inserted after r'' while still concurrent with $w(seq, i)$. Let w have the version (seq, i). The key point of our adversary argument is to keep the *maximum* version of some replica majority as *small* as possible before $w(seq, i)$ is done, so the interfering writes can have the chance to query that replica majority and write with the version value smaller than (seq, i).

Now let us describe the construction with adversary argument in detail. First of all, at the finish time of r'', if the maximum version of any replica majority become not smaller than (seq, i), then no interfering two-round-trip write operation after r'' can have the version value smaller than (seq, i). Therefore, there still exists some replica majority whose maximum version value is smaller than (seq, i) after the finish time of r'', and the interfering two-round-trip write operations may still have the chance to have the version value smaller than (seq, i).

Be aware that w will not have $ver(w) = (seq, i)$ unless it queries a majority of replicas whose maximum version is $(seq - 1, i_1)$ in its first round-trip. After the finish time of r'', there exist two possibilities. If the maximum version of *any* replica majority becomes not smaller than $(seq - 1, i_1)$, then any interfering two-round-trip write operation can only have the version not smaller than (seq, i'), where $i' < i$. If there still exists some replica majority whose maximum version is smaller than $(seq - 1, i_1)$ (the write operation $w(seq - 1, i_1)$ is not yet finished at this time for sure), take $(seq - 2, i_2)$ for instance, then the interfering two-round-trip write operations may still have the chance to have the version smaller than $(seq - 1, i_1')$. Since $(seq - 1, i_1') < (seq, i')$, for adversary argument, we choose the latter possibility, i.e., let $w(seq - 1, i_1)$ be unfinished at the finish time of r'', so more interfering write operations can be inserted after r''.

Recursively, be aware that a write operation will not have the version value $(seq - 1, i_1)$ unless it queries a majority of replicas whose maximum version is $(seq - 2, i_2)$ in its first round-trip. For adversary argument, let $w(seq - 2, i_2)$ be unfinished at the finish time of r'', so more interfering write operations can be inserted after r''. Similarly, let $w(seq - 3, i_3)$, $w(seq - 4, i_4)$, ..., $w(seq - n + 1, i_{n-1})$ be unfinished at the finish time of r'' until the operations of all writer clients (except the writer client of the operation $w(seq, i)$) are in.

Above all, we adversely make a construction that allows the most interfering write operations in the pattern of WI. Note that all of $w(seq - 1, i_1)$, $w(seq - 2, i_2)$, ..., $w(seq - n + 1, i_{n-1})$ can be ordered before r'' in π. Then, after $w(seq - n + 1, i_{n-1})$ is finished while $w(seq - 1, i_1)$, $w(seq - 2, i_2)$, ..., $w(seq - n + 2, i_{n-2})$ are processing, the writer client i_{n-1} may query a majority of replicas whose maximum version value is $w(seq - n + 1, i_{n-1})$ and can issue an interfering write operation $w(seq - n + 2, i_{n-1})$. Similarly, after $w(seq - n + 1, i_{n-1})$ and $w(seq - n + 2, i_{n-2})$ are finished while $w(seq - 1, i_1)$, $w(seq - 2, i_2)$, ..., $w(seq - n + 3, i_{n-3})$ are processing, the writer clients i_{n-1} and i_{n-2} may query a majority of replicas whose maximum version value is $w(seq - n + 2, i_{n-2})$, and then issue interfering write operations $w(seq - n + 3, i_{n-1})$ and $w(seq - n + 3, i_{n-2})$ respectively. In this way, after all of $w(seq - 1, i_1)$, $w(seq - 2, i_2)$, ..., $w(seq - n + 1, i_{n-1})$ are done, all writer clients except i may query a majority of replicas whose maximum version value is $w(seq - 1, i_{n-1})$, and then issue interfering write operations $w(seq, i'), i' < i$. Let i be the maximum client identifier. Then, the maximum number of interfering write operations using WI is:

$$B_2^{WI} = 1 + 2 + \cdots + (n_w - 2) + (n_w - 1) = \frac{1}{2} n_w (n_w - 1)$$

Fig. 4 shows a concrete example of the most stale read with WI when $n_w = 4$, where the pattern of operations is shown in Fig. 2 (a). The read operation r returns the value written by $w(100, 3)$ issued by the writer client with the maximum identifier.

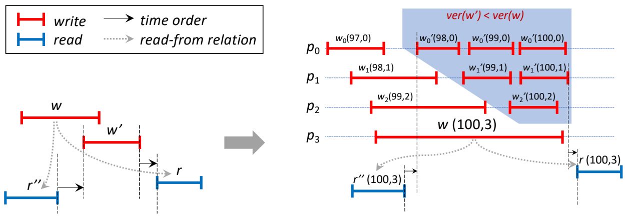

Fig. 4. Maximum number of interfering writes in the pattern of WI in Case 2.

The construction of WI and RI

Now we discuss the situation with the additional effects of RI. For RI, according to Case 2 of Proposition 2.1, there exists a read r'' (can be r) that is dictated by w preceding another read r' dictated by w'. For adversary argument, let r be the read operation with the latest starting time dictated by w, so it must be the most stale read among those dictated by w.

Consider any write operation w' that is concurrent with w and versioned smaller than $ver(w)$. As for the dictated read r'' of w, we mainly focus on the one that *ends earliest*, because the position of w in π is largely decided by it compared to other dictated reads of w. Note that if w' is an interfering write of r, then either w' forms WI with r'', or a dictated read of w' forms RI with r''. Without losing generality, w' can be preceding, concurrent with or starting after r''.

Now we describe an approach to linearly extending σ to a permutation π where the number of interfering writes in Case 2 is no more than B_2^{WI}. For any w' that is preceding or concurrent with r'', let w' precede w and r'' in π, preventing it from interfering the dictated reads of w. For any w' that starts after r'', w' must be ordered after w and r''. Thus, each of these w' will interfere the dictated read(s) of w that start after it. Repeat above procedures recursively for all write operations. In this way, all w' that start not after r'' will not interfere the dictated reads of w, and only those w' that starts after r'' can have the chance to interfere the dictated reads of w.

We now analyze the maximum staleness of each read dictated by w' or w in above permutation. For any read r' dictated by w', if it starts after some read operation r''' dictated by w , then r' and r''' form RI. Since $ver(r') = ver(w') < ver(w)$, according to the worst-case construction using RI in Case 1, the interfering write operations of r' is no more than $B_1 = n_w$. For any read r''' dictated by w, if it starts after some write w', then w' and r''' form the pattern of WI. According to the worst-case construction using WI in Case 2, the number of interfering write operations versioned smaller than $ver(w)$ and starting after r'' is no more than $B_2^{WI} = \frac{1}{2} n_w (n_w - 1)$.

Note that this approach only promises to obtain a legal permutation π of any given σ. Whether or not there exists some other legal permutations with smaller maximum staleness of all reads, the interfering writes of all dictated reads of w in Case 2 (versioned smaller than $ver(w)$) are definitely no more than B_2^{WI}.

Since we have constructed the worst-case trace with B_2^{WI} interfering writes using WI before, we have the bound $B_2 = B_2^{WI} = \frac{1}{2}n_w(n_w - 1)$, and the bound B_2 is tight.

Given a specific read operation r, the interfering write operations in Case 1 and Case 2 can appear at the same time in the history σ, so we can construct a trace with $B_1 + B_2$ interfering writes. Counting the dictating write itself, we have that the read always returns the value written by one of the latest

$$B = B_1 + B_2 + 1 = n_w + \frac{1}{2}n_w(n_w - 1) + 1$$

preceding writes, i.e. the history σ always satisfies B-atomicity. During the proof, we explicitly construct the worst-case trace, which contains the read having $B_1 + B_2$ interfering writes. This proves that the bound is tight [4]. □

An example of the worst-case construction is illustrated in Fig. 5. The process of version updates on replicas is shown in Table 2.

Till now, we prove the correctness of Theorem 2.2. The bound of data staleness using the W2R1 algorithm is related to the number of writer clients. Specially, for *single-writer* multiple-reader registers, W2R1 satisfies 2-atomicity, which conforms to the conclusion in [2].

2.3 Probability of Atomicity Violation

In this subsection we quantify the atomicity violation incurred in W2R1. The quantification follows from Theorem 2.1 that, atomicity violation incurred in W2R1 are composed of either RI or WI. We simplify the quantification problem into the following one: what is the probability that RI or WI incurred in W2R1? The simplification helps us to quantify atomicity violation by ignoring unnecessary details. Note that the occurrence of either RI or WI is merely the condition *necessary* but *not sufficient* for atomicity violation in W2R1, but this will not prevent us from obtaining the upper bound of the violation probability:

$$\begin{aligned} & \mathbb{P}\{\text{Violation}\} \leq \mathbb{P}\{\text{RI} \vee \text{WI}\} \\ = \;& \mathbb{P}\{\text{RI}\} + \mathbb{P}\{\text{WI}\} - \mathbb{P}\{\text{RI} \wedge \text{WI}\} \\ \leq \;& \mathbb{P}\{\text{RI}\} + \mathbb{P}\{\text{WI}\} \end{aligned} \tag{2.1}$$

According to the case-by-case proof of Theorem 2.1, RI and WI can only occur within certain conditions in W2R1. Intuitively, an execution with higher degree of concurrency would produce more atomicity violation, but it is still not sufficient without certain read-write patterns. Thus, we decompose RI and WI incurred in W2R1 into a *concurrency pattern* and a *read-write pattern* separately. The *concurrency pattern(CP)* describes the constraints of real-time orders among events of operations, and the *read-write pattern(RWP)* limits read/write semantics among operations. From the pattern analysis in Theorem 2.1, we have

4. Actually, the proof of some simple corner cases (e.g. $n_w < 3$) is a little different in details and not elaborated here, but the bound is not affected.

Definition 2.1 (The RI Pattern in W2R1). *The RI in W2R1 involves one write w' and two reads r, r', satisfying the requirements of*

- *the long-lived-write concurrency pattern(CP):*
  1) $r_{st} \in [w'_{st}, w'_{ft}]$,
  2) $r'_{ft} \in [w'_{st}, r_{st}]$;
- *the non-monotonic read-write pattern(RWP):*
  3) $ver(r') = ver(w')$;
  4) $ver(r) \neq ver(w')$.

Definition 2.2 (The WI Pattern in W2R1). *The WI in W2R1 involves two writes w, w' and one read r'', satisfying the requirements of*

- *the long-lived-write concurrency pattern(CP):*
  1) $w'_{st} \in [w_{st}, w_{ft}]$,
  2) $r''_{ft} \in [w_{st}, w'_{st}]$;
- *the non-monotonic read-write pattern(RWP):*
  3) $ver(r'') = ver(w)$;
  4) $ver(w') \neq ver(w)$.

To calculate the probabilities of RI and WI, we view RI and WI as the concurrent occurrences of multiple atomicity violation patterns in the single-writer case. According to our previous work [2], atomicity violation in the single-writer case is equivalent to the pattern named Old New Inversion (ONI). We further destruct the ONI into the temporal Concurrency Pattern (CP) and the semantic Read Write Pattern (RWP). Then we obtain the probability of ONI:

$$\begin{aligned} \mathbb{P}\{\text{ONI}\} \;&=\; \sum_{m \geq 1} \mathbb{P}\{\text{CP} \mid R' = m\} \cdot \mathbb{P}\{\text{RWP} \mid R' = m\} \\ &\approx\; \sum_{m=1}^{n_r} \Bigg(\bigg(\sum_{k=0}^{n_r - 1} \binom{n_r}{k} \binom{m-1}{n_r - k - 1} p_0^k r^{n_r - k} s^m \bigg) \\ &\quad \cdot e^{-q\lambda_w t} \frac{\alpha^q B(q, \alpha(N-q)+1)}{B(q, N-q+1)} \\ &\quad \cdot \bigg(1 - \Big(\frac{J_1}{B(q, N-q+1)} \Big)^m \bigg) \Bigg). \end{aligned} \tag{2.2}$$

where R' is a random variable denoting the number of r's in Definition 2.1. It is an approximation since the timed balls-into-bins model used for calculating the probability of read-write patterns assumes that there is at most one such r' in each reader queue, which can be justified by numerical results. For further details, please refer to our previous work [2] for detailed explanations of Equation 2.2.

Through observation we can see that ONI is a special case of RI when there exists one single writer. Based on the quantification of ONI, we can quantify RI for MWMR registers with extra consideration of the concurrency effects of multiple writers. As for WI, by replacing r, w', r' in Definition 2.1 with w', w, r'' accordingly, it can be analyzed with similar process. Above all, we simplify MWMR quantification by reusing results of ONI and taking the extra effects of multiple writers into consideration together.

We first model the occurrences of multiple writers with the following queuing model. Each client's workload is recognized as an independent queue characterized by the rate of operations and the service time of each operation (i.e.

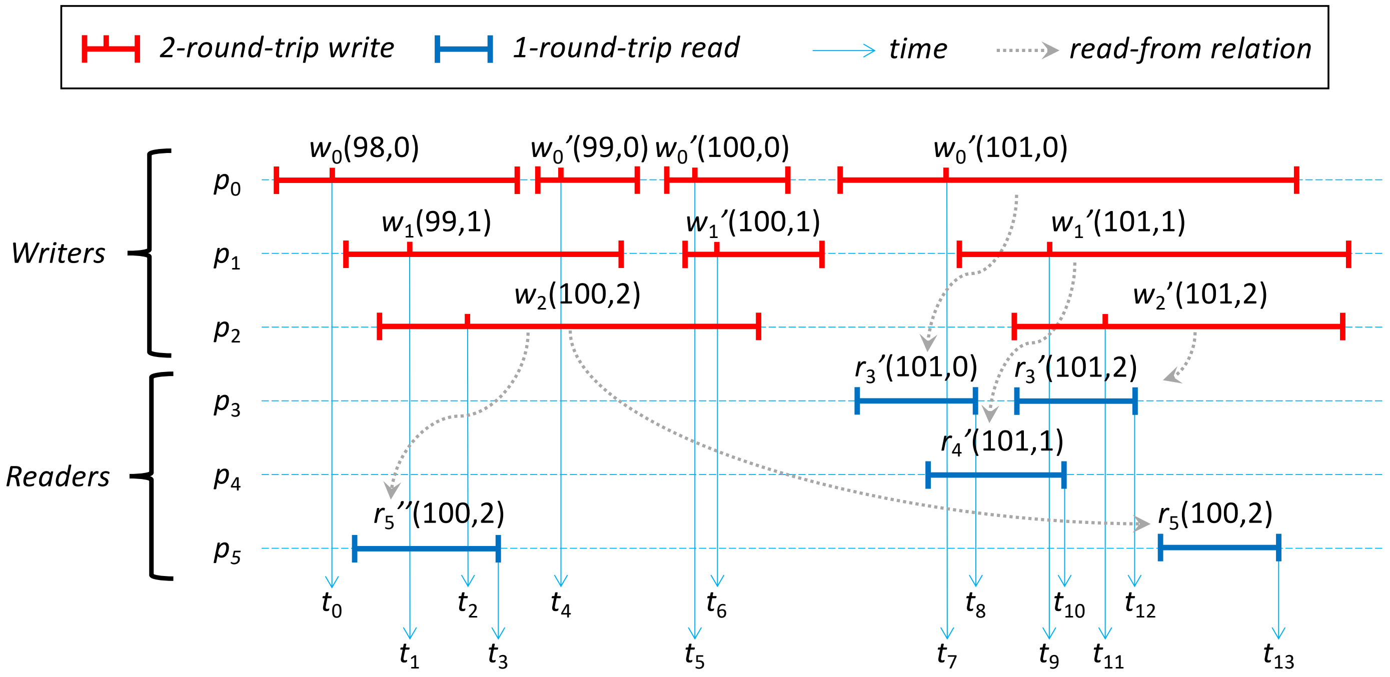

Fig. 5. An example of worst case with 3 writer clients and 3 reader clients using the W2R1 algorithm. The *version* that a *write* assigns or a *read* returns are displayed in brackets. One permutation of these operations can be: $w_0(98,0) \prec_\pi w_1(99,1) \prec_\pi \mathbf{w_2(100,2)} \prec_\pi r_5''(100,2) \prec_\pi w_0'(99,0) \prec_\pi w_0'(100,0) \prec_\pi w_1'(100,1) \prec_\pi w_0'(101,0) \prec_\pi r_3'(101,0) \prec_\pi w_1'(101,1) \prec_\pi r_4'(101,1) \prec_\pi w_1'(101,2) \prec_\pi r_3'(101,2) \prec_\pi \mathbf{r_5(100,2)}$. For $r_5(100,2)$, it returns the value of the latest 7 preceding write in π. Interfering write operations w' of r include $w_0'(99,0), w_0'(100,0), w_1'(100,1), w_0'(101,0), w_1'(101,1), w_2'(101,2)$. Among these operations, $r_5''(100,2)$ and each of $w_0'(99,0), w_0'(100,0), w_1'(100,1)$ form the pattern of WI respectively; $r_5(100,2)$ and each of $r_3'(101,0), r_4'(101,1), r_3'(101,2)$ form the pattern of RI respectively.

TABLE 2
The process of version updates on replicas in Fig. 5. Suppose the distributed storage system contains 3 replicas.

Time	Replica 1	Replica 2	Replica 3	Description
t_0	**(97,0)**	**(97,0)**	(97,0)	p_0 then assigns $(98,0)$ after the query returns maximum version $(97,0)$ in 1^{st} round-trip.
t_1	**(98,0)**	**(97,0)**	(97,0)	p_1 then assigns $(99,1)$ after the query returns maximum version $(98,0)$ in 1^{st} round-trip.
t_2	**(99,1)**	**(97,0)**	(97,0)	p_2 then assigns $(100,2)$ after the query returns maximum version $(99,1)$ in 1^{st} round-trip.
t_3	**(100,2)**	**(97,0)**	(97,0)	p_5 returns the value versioned $(100,2)$ in the read.
t_4	(100,2)	**(98,0)**	**(97,0)**	p_0 then assigns $(99,0)$ after the query returns maximum version $(98,0)$ in 1^{st} round-trip.
t_5	(100,2)	**(99,0)**	**(97,0)**	p_0 then assigns $(100,0)$ after the query returns maximum version $(99,0)$ in 1^{st} round-trip.
t_6	(100,2)	**(99,0)**	**(97,0)**	p_1 then assigns $(100,1)$ after the query returns maximum version $(99,0)$ in 1^{st} round-trip.
t_7	(100,2)	**(100,2)**	**(97,0)**	p_0 assigns $(101,0)$ after the query returns maximum version $(100,2)$ in 1^{st} round-trip.
t_8	(100,2)	**(101,0)**	**(97,0)**	p_3 returns the value versioned $(101,0)$ in the read.
t_9	**(100,2)**	(101,0)	**(97,0)**	p_1 assigns $(101,1)$ after the query returns maximum version $(100,2)$ in 1^{st} round-trip.
t_{10}	(100,2)	**(101,1)**	**(97,0)**	p_4 returns the value versioned $(101,1)$ in the read.
t_{11}	**(100,2)**	(101,1)	**(97,0)**	p_2 assigns $(101,2)$ after the query returns maximum version $(100,2)$ in 1^{st} round-trip.
t_{12}	(100,2)	**(101,2)**	**(97,0)**	p_3 returns the value versioned $(101,2)$ in the read.
t_{13}	**(100,2)**	(101,2)	**(97,0)**	p_5 returns the value versioned $(100,2)$ in the read.

$[o_{st}, o_{ft}]$). Assume a Poisson process with parameter λ for the scenario of each client issuing a sequence of read/write operations, and an exponential distribution with parameter μ for the service time of each operation. We then have n independent, parallel $M/M/1/1/\infty/FCFS$ queues (i.e., a single-server exponential queuing system, whose capacity is 1 with the "first come first served" discipline) [3]. All the queues have arrival rate λ and service rate μ. For simplicity, queues of writes and reads are separate; and if there is any operation in service, no more operations can enter it in that queue.

Let $X^i(t)$ be the number of operations in queue i at time t. Then $X^i(t)$ is a continuous-time Markov chain with two states: 0 when the queue is empty and 1 when some operation is being served. Its stationary distribution $P_s \triangleq P(X^i(\infty) = s), s \in \{0, 1\}$ is:

$$P_0 = \frac{\mu}{\mu+\lambda}, P_1 = \frac{\lambda}{\mu+\lambda}.$$

Now we consider the concurrency conditions of RI and WI respectively. For RI, given a read r in Q_i, let W'_{cr} be a random variable denoting the number of w's satisfying $r_{st} \in [w'_{st}, w'_{ft}]$ in RI. The probability of the event that r starts during the service period of some write w' in Q_j equals the probability that when r arrives Q_i, it finds Q_i empty, as well as finds Q_j full as a bystander (with the constraint that Q_j is a writer queue). Since the events in different queues are independent, by the PASTA property and through combinatorial analysis, we have

$$\mathbb{P}\{W'_{cr} = x\} = \binom{n_w}{x} \left(\frac{\lambda}{\mu+\lambda}\right)^x \left(\frac{\mu}{\mu+\lambda}\right)^{n_w-x+1} \quad (2.3)$$

Conditioning on $W'_{cr} = w$, the probability of RI is no more than the sum of each w' forming RI with r seperately. Therefore, a probabilistic and combinatorial analysis shows that

$$\begin{aligned} \mathbb{P}\{\text{RI}\} &= \sum_{x=1}^{n_w} \mathbb{P}\{W'_{cr} = x\} \cdot \mathbb{P}\{\text{RI} \mid W'_{cr} = x\} \\ &\leq \sum_{x=1}^{n_w} \mathbb{P}\{W'_{cr} = x\} \cdot x\mathbb{P}\{\text{ONI}\} \quad (2.4) \end{aligned}$$

As for WI, given a specific write w' in Definition 2.2, let $W_{cw'}$ be a random variable denoting the number of writes w satisfying $w'_{st} \in [w_{st}, w_{ft}]$ in RI. Similarly, through probabilistic and combinatorial analysis we have

$$\mathbb{P}\{W_{cw'} = x\} = \binom{n_w-1}{x} \left(\frac{\lambda}{\mu+\lambda}\right)^x \left(\frac{\mu}{\mu+\lambda}\right)^{n_w-x} \quad (2.5)$$

$$\begin{aligned} \mathbb{P}\{\text{WI}\} &= \sum_{x=1}^{n_w-1} \mathbb{P}\{W_{cw'} = x\} \cdot \mathbb{P}\{\text{WI} \mid W_{cw'} = x\} \\ &\leq \sum_{x=1}^{n_w-1} \mathbb{P}\{W_{cw'} = x\} \cdot x\mathbb{P}\{\text{ONI}\} \quad (2.6) \end{aligned}$$

Substituting Formula (2.2)-(2.6) into (2.1), we obtain an upper bound of the rate of violating atomicity incurred in W2R1.

$$\begin{aligned} &\mathbb{P}\{\text{Violation}\} \\ \leq &\sum_{x=1}^{n_w} \mathbb{P}\{W'_{cr} = x\} \cdot x\mathbb{P}\{\text{ONI}\} + \sum_{x=1}^{n_w-1} \mathbb{P}\{W_{cw'} = x\} \cdot x\mathbb{P}\{\text{ONI}\} \\ \approx &\frac{(2n_w-1)\lambda\mu}{(\lambda+\mu)^2} \sum_{m=1}^{n_r} \left(\left(\sum_{k=0}^{n_r-1} \binom{n_r}{k} \binom{m-1}{n_r-k-1} p_0^k r^{n_r-k} s^m \right) \right. \\ &\left. e^{-q\lambda_w t} \frac{\alpha^q B(q, \alpha(N-q)+1)}{B(q, N-q+1)} \left(1 - \left(\frac{J_1}{B(q, N-q+1)}\right)^m\right) \right) \end{aligned}$$

The results in Table 3 and Fig. 6 are obtained when setting $\lambda = \mu = 10s^{-1}$ and $\lambda_r = \lambda_w = 20s^{-1}$. The source code for calculation of the numerical results can be found in our open source project on line [5]. The numerical results show that the probability of atomicity violation is quite low (mostly below 0.1% and can be below 10^{-9}) and will decrease when the number of replicas increases. Moreover, the probability of the concurrency pattern is close to 1. The probability of the read-write pattern is significantly less and decreases as the number of replicas increases. The probability of the read-write pattern ensure that the probability of the atomicity violation is almost zero. Besides, the probability of atomicity violation using the W2R1 algorithm has positive correlation with the number of concurrent writers. The comprehensive experimental results in Section 4 in the appendix further confirm our theoretical analysis.

5. https://github.com/Lingzhi-Ouyang/Almost-Strong-Consistency-Cassandra/tree/master/numerical_results

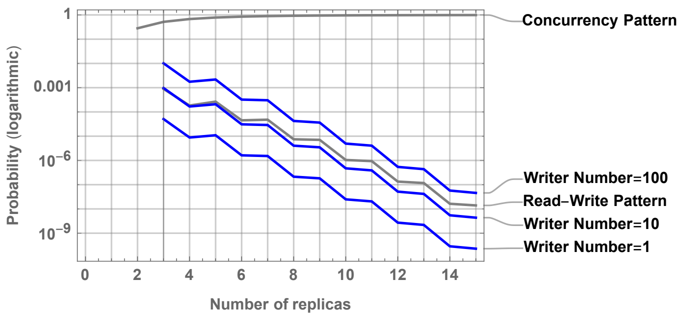

Fig. 6. The probabilities of atomicity violation.

3 W1R2 and W1R1: Theoretical Analysis and Experimental Evaluation

From the diamond schema we have one-round-trip write algorithms W1R2 & W1R1, as shown in Algorithm 5 & 6, which complete each write in only one communication round-trip. When a writer client initiates a write request, it would constructs a version for the written key-value pair based on its local sequence number record. The version assigned for its written value is only guaranteed to be larger than any version returned by its previous reads or assigned by its previous writes, but not others'. Therefore, one-round-trip write works well only for *single-writer* registers since the only writer client can promise to preserve the version for a register monotonically increasing without conflicting with other writer clients. For *multiple-writer* registers, one-round-trip write may result in *write inversion anomalies after a write*: a later write updates a value with a smaller version than a previous write from a different writer client. Here, we define a write is *invisible* if before the start time of the write, the maximum version of any server replica majority is larger than its assigned version for the update. Obviously, the data from an invisible write cannot be returned by any read.

In this section, we tend to demonstrate that one-round-trip write algorithms are useless from two aspects. First we prove that neither W1R2 nor W1R1 can guarantee bounded staleness of data for reads in Section 3.1. Then we demonstrate that both algorithms lead to frequent unsuccessful updates by quantifying the rate of invisible writes in Section 3.2.

3.1 Proof of Atomicity Violation in One-Round-Trip Write

We prove that, in the W1R2 or W1R1 algorithm, the value returned by each read can be any stale. That is,

Theorem 3.1. *For MWMR registers, given $k \in \mathbb{N}^*$, the W1R2 or W1R1 algorithm doesn't satisfy k-atomicity.*

Proof. Assume W1R2 or W1R1 satisfies k-atomicity, where $k \in \mathbb{N}^*$. Then, we can raise a counter-example to show the absurdity of above assumption. The counter-example

TABLE 3
Numerical results on the probabilities of concurrency patterns, read-write patterns, and old-new inversions.

# Replicas	$\mathbb{P}\{\text{CP}\}$	$\mathbb{P}\{\text{RWP}\}$	$\mathbb{P}\{n_w = 1\}$	$\mathbb{P}\{n_w = 10\}$	$\mathbb{P}\{n_w = 100\}$
2	0.28125	0.	0.	0.	0.
3	0.518555	0.00088802	0.0000509207	0.000967493	0.0101332
4	0.677307	0.000183791	8.82396×10^{-6}	0.000167655	0.00175597
5	0.781222	0.000266569	1.09295×10^{-5}	0.000207661	0.00217498
6	0.849318	0.0000450835	1.62306×10^{-6}	0.0000308382	0.00032299
7	0.89429	0.0000478926	1.5218×10^{-6}	0.0000289142	0.000302839
8	0.924335	0.43561×10^{-6}	2.13453×10^{-7}	4.0556×10^{-6}	0.0000424771
9	0.9447	7.06025×10^{-6}	1.82686×10^{-7}	3.47103×10^{-6}	0.0000363545
10	0.95874	1.04312×10^{-6}	2.48339×10^{-8}	4.71844×10^{-7}	4.94195×10^{-6}
11	0.968604	9.37995×10^{-7}	2.04234×10^{-8}	3.88044×10^{-7}	4.06425×10^{-6}
12	0.975675	1.34085×10^{-7}	2.72056×10^{-9}	5.16906×10^{-8}	5.41391×10^{-7}
13	0.98085	1.16911×10^{-7}	2.19289×10^{-9}	4.1665×10^{-8}	4.36386×10^{-7}
14	0.984717	1.63195×10^{-8}	2.87944×10^{-10}	5.47094×10^{-9}	5.73009×10^{-8}
15	0.987662	1.39573×10^{-8}	2.29571×10^{-10}	4.36184×10^{-9}	4.56846×10^{-8}

Algorithm 5: W1R2

```
Code for client process $p_i (0 \le i \le n-1)$:
procedure OneRoundWRITE($key, value$)
    $localSeq[key] \leftarrow localSeq[key]$+1
    $version \leftarrow (localSeq[key], i)$
    update($key, value, version$)
procedure TwoRoundREAD($key$)
    $replicas \leftarrow$ query($key$)
    $version \leftarrow$maxVer($replicas$)
    $value \leftarrow$valWithMaxVer($replicas, version$)
    $localSeq[key] \leftarrow version.seq$   ▷ Recommended.
    update($key, value, version$)
    return $value$
```

Algorithm 6: W1R1

```
procedure OneRoundWRITE($key, value$)
    $localSeq[key] \leftarrow localSeq[key]$+1
    $version \leftarrow (localSeq[key], i)$
    update($key, value, version$)
procedure OneRoundREAD($key$)
    $replicas \leftarrow$ query($key$)
    $version \leftarrow$maxVer($replicas$)
    $value \leftarrow$valWithMaxVer($replicas, version$)
    $localSeq[key] \leftarrow version.seq$   ▷ Recommended.
    return $value$
```

is constructed as follows. Suppose there exist two writer clients whose identifiers are p_0 and p_1, and we only focus on the same register. At time t_0, p_0 stores v_0 as its local sequence number for the register, and p_1 stores v_1, satisfying $v_1 > v_0$. During the period from t_0 to t_1, p_1 executes k writes successively while p_0 initiates no writes. Thus, at time t_1, p_1's local version is increased to be $(v_1 + k, 1)$ while p_0 still keeps v_0. After that, during the period from t_1 to t_2, only p_0 executes k writes successively and its local version is increased to be $v_0 + k$ at t_2. Since $(v_0 + k, p_0) < (v_1 + k, p_1)$, the written value versioned $(v_0 + k, p_0)$ is invisible. Then at t_3, a read occurs but still reads the value versioned $(v_1 + k, p_1)$. In above case, the permutation of operations on p_0 and p_1 goes like this: (The *version* that a *write* assigns or a *read* returns are shown in brackets.) $w(v_1 + 1, p_1) \prec_\pi w(v_1 + 2, p_1) \prec_\pi ... \prec_\pi \underline{w(v_1 + k, p_1)} \prec_\pi w(v_0 + 1, p_0) \prec_\pi w(v_0 + 2, p_0) \prec_\pi ... \prec_\pi \underline{w(v_0 + k, p_0)} \prec_\pi \underline{r(v_1 + k, p_1)}$. Since $r(v_1 + k, p_1)$ returns a value not written by one of the latest k preceding writes in π, so the history violates k-atomicity. By contradiction, we prove the correctness of Theorem 3.1. $\square$

3.2 Quantification of Visible Writes in One-Round-Trip Write

The one-round-trip write algorithms including W1R2 and W1R1 not only cannot provide bounded k-atomicity, but also lead to frequent invisible writes that cannot be read. Here, we quantify the rate of visible writes incurred in W1R2 & W1R1 using the following assumption. Assume a Poisson process with parameter λ for the scenario of each client issuing a sequence of write/read operations, but *ignore the duration of each operation*. That is, each write/read is regarded as an instant event taking effect on the register at its starting moment. Suppose there are n_w writer clients($n_w > 1$), identified 0, 1, ..., $n_w - 1$ separately. Then we have n_w independent, parallel write sequences, each satisfying a Poisson process with parameter λ. Assume that at the initial moment, all n_w writer clients are informed of the version (v_0, id) for some specific register that we focus on here. Let $N^i(t)$ be the number of write operations issued by writer client i from the initial moment till time t, where $0 \le i \le n_w - 1$. According to the formula of Poisson probability, we have

$$\mathbb{P}\{N^i(t) = k\} = \frac{(\lambda t)^k e^{-\lambda t}}{k!}$$

Assume that writer client i stores the local sequence number $k - 1$ at time t. Let $V^i(t, k)$ be a random variable denoting the visibility of the un-occurred write versioned $(v_0 + k, i)$ at time t. Then, $V^i(t, k)$ has two status: invisible(denoted 0) and visible(denoted 1). $V^i(t, k) = 1$ occurs if and only if all other writer clients' local sequence number stayed smaller than $(v_0 + k, i)$ at time t. Since write sequences are independent, we have

$$\begin{aligned}&\mathbb{P}\{V^i(t,k)=1\}\\&=\prod_{j=0}^{i-1}\mathbb{P}\{N^j(t)<k+1\}\prod_{j=i+1}^{n_w-1}\mathbb{P}\{N^j(t)<k\}\\&=\left(\sum_{j=0}^{k}\frac{(\lambda t)^j e^{-\lambda t}}{j!}\right)^i\left(\sum_{j=0}^{k-1}\frac{(\lambda t)^j e^{-\lambda t}}{j!}\right)^{n_w-1-i}\end{aligned}$$

Here, we quantify the rate when $k=1$ at the expected arrival time $t=\frac{1}{\lambda}$, when the probability of $N^j(t)=1$ is the highest.

By setting $\lambda=10s^{-1}$, we present the numerical results in Fig. 7 and Table 4. The numerical results show that the probability of visible writes will fall as there exist multiple writer clients. When the number of writer clients is up to 10, the probability of visible writes for the writer client with the maximum id is about 0.063, while for the writer client with the minimum id, the probability is about 10^{-4}.

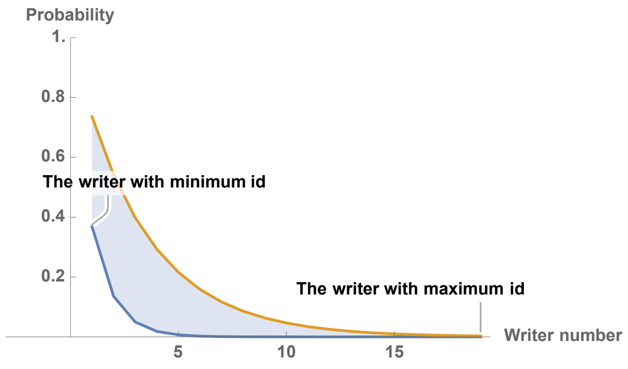

Fig. 7. The probabilities of visible writes from writer clients with different identifiers ($\lambda=10s^{-1}, t=100ms$).

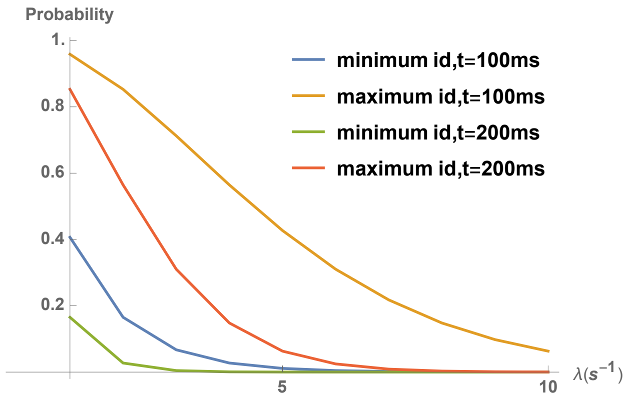

Fig. 8. The probabilities of visible writes from the writer client with maximum / minimum identifier by varying λ ($n_w=10$).

3.3 Experimental Results of One-Round-Trip Write Algorithms

The rough theoretical analysis in Section 3.2 doesn't take the duration of each operation into consideration, so we conduct more comprehensive experimental evaluations of W1R2 and W1R1 in quorum-replicated datastores. The experimental design and the environment settings are described in the experiments of W2R1, and the experimental results of W1R2 and W1R1 are presented in Table 5.

For read operations, the overall probability of obtaining an up-to-date value is varied from 46.6% to 91.1%. However, under most environment settings, the probability is around 50%. That is, nearly half of the read operations will return stale data. Moreover, the staleness of the returned data can be quite high. We only calculate the rate of k up to 10 in terms of k-atomicity. Specifically, the probability of returning an value with the staleness of 5 is varied from 0.7% to 8.1%. When the staleness is up to 10, the probability is around 1%. Note that if the returned data is with staleness of 5 or even more, it is nearly of no use for users. Thus, reader clients may frequently obtain stale or even invalid data. From the aspect of writes, the considerable possibility of issuing invisible updates is the main reason for reads to return stale data. Therefore, the algorithms of both W1R2 and W1R1 without any improvement are far away from use [6].

3.4 Improvement of One-Round-Trip Write Algorithms

Imagine slightly modifying the procedure of $update$ function. When a replica server receives an "update" message and finds the intended update version smaller than it stores, the server additionally fills the ACK with extra information (like the sequence number of its stored version) instead of nothing and reply to the client. A client would have to wait for ACKs from a majority of replicas and update its local sequence number according to the information in the ACKs. Then, an invisible one-round-trip write may work like the $query$ function and help update the local sequence number on the client side. In this way, W1R2 can be proved to satisfy n_w-atomicity given n_w writer clients.

Here we quantify the rate of visible writes in the improved W1R2 & W1R1 algorithms. Suppose the writer client i issues a write w after time t from its last write operation w'. Let $U^i(t)$ be a random variable denoting the visibility of the write w. Similarly, $U^i(t)$ has two status: invisible(denoted 0) and visible(denoted 1). Here, we only focus on a specific but frequent pattern that would inevitably make w invisible: during the period from the finish time of w' to the invocation of w, there exists a writer client identified smaller than i completing at least three writes, or a writer client identified larger than i completing at least two writes [7]. Since write sequences are independent, through probabilistic and

6. For MWMR registers, the W1R2 and W1R1 algorithms that use discrete version (seq, id) are far away from use. However, the one-round-trip write algorithms in Cassandra and some other datastores use timestamps following the UTC standard, which is different from our algorithms. Theoretically speaking, these datastores cannot always promise to produce unique timestamps for different data, and that is out of scope in this work.

7. This can be proved through a case-by-case analysis.

TABLE 4
Numerical results on the probabilities of **invisible** writes in one-round-trip write algorithms without extra information in ACK ($\lambda = 10s^{-1}, t = 100ms$).

Writer numbers	$id = 0$	$id = 1$	$id = 2$	$id = 3$	$id = 4$	$id = 5$	$id = 6$	$id = 7$	$id = 8$	$id = 9$
$n_w = 2$	0.632121	0.264241								
$n_w = 3$	0.864665	0.729329	0.458659							
$n_w = 4$	0.950213	0.900426	0.800852	0.601703						
$n_w = 5$	0.981684	0.963369	0.926737	0.853475	0.70695					
$n_w = 6$	0.993262	0.986524	0.973048	0.946096	0.892193	0.784386				
$n_w = 7$	0.997521	0.995042	0.990085	0.98017	0.96034	0.92068	0.84136			
$n_w = 8$	0.999088	0.998176	0.996352	0.992705	0.98541	0.97082	0.94164	0.883279		
$n_w = 9$	0.999665	0.999329	0.998658	0.997316	0.994633	0.989265	0.97853	0.957061	0.914122	
$n_w = 10$	0.999877	0.999753	0.999506	0.999013	0.998025	0.996051	0.992102	0.984204	0.968407	0.936814

TABLE 5
Experimental results of one-round-trip write algorithms without extra information in ACK ($\lambda = 10s^{-1}, t = 100ms$).

Algorithm	Quorum level	Replica	Inter-DC delay/ms	Client num.	Read ratio	$\mathbb{P}(k = 1)$	$\mathbb{P}(k = 5)$	$\mathbb{P}(k = 10)$
W1R2	QUORUM	1_1_1	$\mathbb{N}(50, 25^2)$	30	0.50	0.911	0.007	0.010
W1R2	QUORUM	1_1_1	$\mathbb{N}(50, 25^2)$	30	0.60	0.878	0.014	0.009
W1R2	QUORUM	1_1_1	$\mathbb{N}(50, 25^2)$	30	0.90	0.541	0.058	0.011
W1R2	QUORUM	3	$\mathbb{N}(50, 25^2)$	30	0.90	0.494	0.057	0.006
W1R2	QUORUM	3_1_1	$\mathbb{N}(50, 25^2)$	30	0.90	0.518	0.069	0.004
W1R2	QUORUM	3_3_3	$\mathbb{N}(50, 25^2)$	30	0.90	0.529	0.064	0.016
W1R2	EACH_QUORUM	1_1_1	$\mathbb{N}(50, 25^2)$	30	0.90	0.540	0.071	0.016
W1R1	QUORUM	1_1_1	$\mathbb{N}(50, 25^2)$	30	0.50	0.864	0.017	0.012
W1R1	QUORUM	1_1_1	$\mathbb{N}(50, 25^2)$	30	0.60	0.816	0.024	0.021
W1R1	QUORUM	1_1_1	$\mathbb{N}(50, 25^2)$	30	0.90	0.507	0.052	0.023
W1R1	QUORUM	3	$\mathbb{N}(50, 25^2)$	30	0.90	0.496	0.091	0.009
W1R1	QUORUM	3_1_1	$\mathbb{N}(50, 25^2)$	30	0.90	0.466	0.061	0.011
W1R1	QUORUM	3_3_3	$\mathbb{N}(50, 25^2)$	30	0.90	0.563	0.059	0.014
W1R1	EACH_QUORUM	1_1_1	$\mathbb{N}(50, 25^2)$	30	0.90	0.496	0.053	0.019

TABLE 6
Numerical results on the probabilities(lower bound) of **invisible** writes in one-round-trip write algorithms if sequence number is included in ACK ($\lambda = 10s^{-1}, t = 100ms$).

Writer numbers	$id = 0$	$id = 1$	$id = 2$	$id = 3$	$id = 4$	$id = 5$	$id = 6$	$id = 7$	$id = 8$	$id = 9$
$n_w = 2$	0.264241	0.0803014								
$n_w = 3$	0.458659	0.323324	0.154154							
$n_w = 4$	0.601703	0.502129	0.377662	0.222077						
$n_w = 5$	0.70695	0.633687	0.542109,	0.427636	0.284545					
$n_w = 6$	0.784386	0.730482	0.663103	0.578878	0.473598	0.341997				
$n_w = 7$	0.84136	0.8017	0.752125	0.690156	0.612695	0.515869	0.394836			
$n_w = 8$	0.883279	0.854099	0.817624	0.77203	0.715037	0.643796	0.554745	0.443431		
$n_w = 9$	0.914122	0.892652	0.865815	0.832269	0.790336	0.73792	0.6724	0.5905	0.488125	
$n_w = 10$	0.936814	0.921018	0.901272	0.87659	0.845738	0.807172	0.758965	0.698707	0.623383	0.529229

combinatorial analysis we have

$$
\begin{aligned}
&\mathbb{P}\{U^i(t) = 0\} \\
\geq& 1 - \prod_{j=0}^{i-1} \mathbb{P}\{N(t) < 3\} \prod_{j=i+1}^{n_w - 1} \mathbb{P}\{N(t) < 2\} \\
=& 1 - e^{-\lambda t(n-1)} \cdot (1 + \lambda t + \frac{1}{2}(\lambda t)^2)^i \cdot (1 + \lambda t)^{n_w - 1 - i}
\end{aligned}
$$

Then, we have

$$
\begin{aligned}
&\mathbb{P}\{U^i(t) = 1\} = 1 - \mathbb{P}\{U^i(t) = 0\} \\
\leq& e^{-\lambda t(n-1)} \cdot (1 + \lambda t + \frac{1}{2}(\lambda t)^2)^i \cdot (1 + \lambda t)^{n_w - 1 - i}
\end{aligned}
$$

Here, we quantify the rate at the expected time $t = \frac{1}{\lambda}$. By setting $\lambda = 10s^{-1}$, we present the numerical results in Fig. 9 and Table 6. The numerical results of the improved W1R2 and W1R1 are similar to those without improvement in Section 3.2, although the probability of visible writes raises a little. When the number of writer clients is up to 10, the probability of visible writes for the writer client with the maximum *id* is about 0.047, while for the writer client with the minimum *id*, the probability is about 0.063. However, the improvement is not significant. Actually, the modification in *update* doesn't prevent the occurrence of invisible writes, so the problem of the one-round-trip write algorithm still exists.

4 W2R1: Experimental Evaluations

In this section, we analyze the experimental results of the W2R1 algorithm in detail. First we explore the impacts of different environment factors, then we conclude the tradeoff of data consistency and read latency in different algorithms.

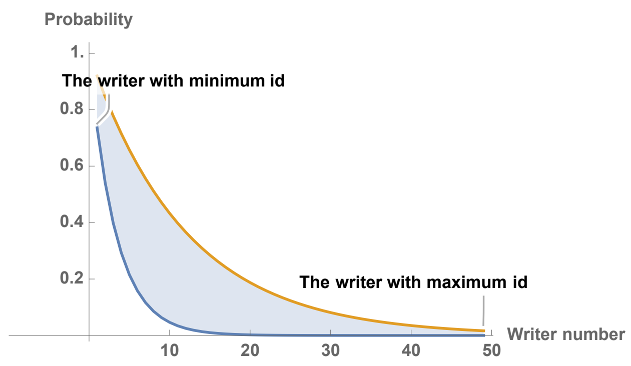

Fig. 9. The probabilities of visible writes from writer clients with different identifiers using the improved update ACK ($\lambda = 10s^{-1}, t = 100ms$).

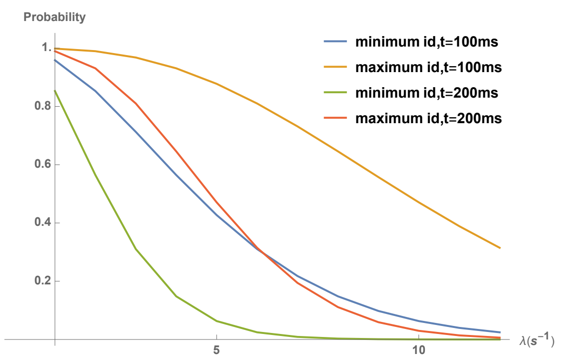

Fig. 10. The probabilities of visible writes from the writer client with maximum / minimum identifier by varying λ using the improved update ACK ($n_w = 10$).

4.1 Impacts of Environment Factors

We investigate the impacts of different environment factors on the consistency and latency performance of the W2R1 algorithm (with / without optimization) running on Cassandra. Besides, the W2R2 algorithm is explored for comparison. The environmental settings are varied from workload pattern (client number and read ratio), replica configuration, communication delay to read quorum level.

4.1.1 Impact of Client Number

The client number reflects the concurrency degree of operations to some extent. Intuitively, more clients are able to result in higher concurrency. In the experiments, each client is acted by one YCSB instance. All clients are allowed to initiate read/write requests concurrently, while each of them can only execute at most one request at any time. By varying the client number from 10 to 40, we derive the results shown in Fig. 11.

First we have a look at the data consistency performance of the algorithms. W2R2 guarantees atomicity all the time as is expected, while other algorithms can lead to atomicity violation more or less. From the perspective of worst case we obtained, W2R1 without any optimization produced some non-atomic traces with k up to 4 in terms of k-atomicity, while others resulted in better performance. From the perspective of average consistency, W2R1 also has the worst performance compared to those with one of the Cassandra's optimizations(snitch, digest or read repair). However, although all but W2R2 cannot satisfy atomicity all the time, they guarantee atomicity most of the time, with a confidence that more than 99.97% read requests can obtain the most up-to-date data. In overall, the average consistency performance becomes weaker as the concurrent client number increases, and we produced the worst case of $k = 4$ when client number is up to 40. We speculate that the rate of returning fresh data for reads would decrease with more concurrent clients. However, most of the time these non-atomic algorithms can still promise atomicity. Therefore, we believe that W2R1 is applicable for many applications and scenarios, especially when concurrent clients operating on single shared register are no more than 40.

As for latency, we observe that the average read latency is positively relative to the client number in a subtle degree. The main reason is obvious: the processing capacity of each server is limited. With the concurrency control, the waiting time in the buffer will become longer as the user throughput increases. From another aspect, W2R2 has higher average read latency compared to the W2R1 algorithms, while it gains strong consistency all the time.

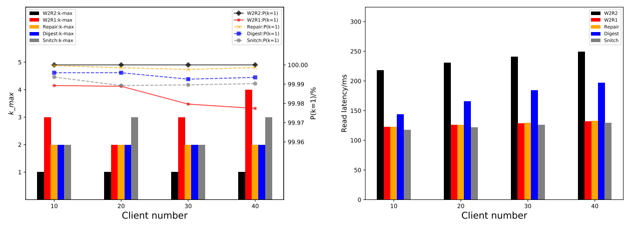

Fig. 11. Consistency and latency results by tuning the number of clients.

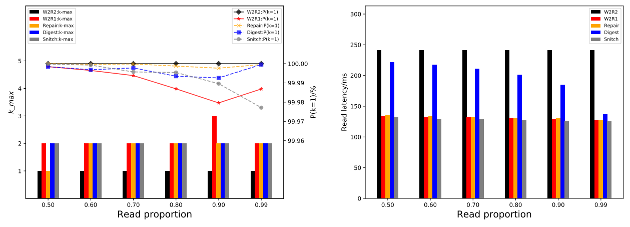

Fig. 12. Consistency and latency results by tuning read ratio.

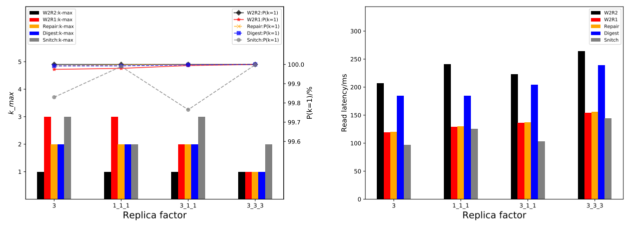

Fig. 13. Consistency and latency results by tuning replica configuration.

4.1.2 Impact of Read Ratio

In the experiments, each client can initiate read and write requests. The ratio between read and write can also be a critical factor to impact the chance to produce violation patterns. Here, we use *read ratio*(the ratio of read numbers to the total operation numbers) to describe the relations between read and write numbers. By varying the read ratio from 0.50 to 0.99, we derive the results shown in Fig. 12. As we can see, W2R2 still guarantees atomicity while others do not. W2R1 with snitch leads to a monotonically decreasing atomicity degree with higher read ratio. For the W2R1 algorithm without optimization or with digest / repair, higher read ratio within certain range (0.5 - 0.9) can result in weaker consistency guarantee in average. However, when the read ratio is up to 0.99, the consistency performance reversely becomes better. One potential reason is that, k-atomicity violation is described by the staleness of data returned by reads, so within certain range, more reads will have more chances to form more complicated read-write patterns that make reads stale. However, as the ratio of reads grows too high, writes will become too rare and sparsely distributed for reads to obtain stale written values.

As for latency, we observe that the average read latency is approximately irrelative to read ratio for all algorithms except W2R1 with digest. The reason why request latency for W2R1 with digest grows higher when read ratio gets lower is mainly that, when write proportion becomes relatively higher, more frequent updates will make digest mismatch occur more frequently, triggering extra round-trip for collecting full data.

4.1.3 Impact of Replica configuration

The replica factor in Cassandra specifies the number of replicas in each data center. The number of replicas impacts the consistency and latency performance in some degree, while the data center location of replicas impacts the request latency. Given a number of replicas, besides spreading them into different data centers, we can also place them all into a single data center for the convenience of local access. In the experiments, the replica factor is varied in 3(inside single data-center), 1_1_1, 3_1_1, and 3_3_3. As is shown in Fig. 13, more replicas would result in stronger consistency but higher latency in overall. The reason is obvious: the latency of an operation depends on the slowest responding replica which costs the longest time to communicate. In the network where delays are randomly distributed, it is more probable to have longer delays when communicating with more replicas. However, the replica configuration of 3_1_1 is an exception. 3_1_1 may sometimes select a quorum of replicas all in the local data center, which can reduce the overall read latency at the cost of consistency lost compared to 1_1_1. Besides, W2R1 with snitch behaves in a similar way. Applying smart routing through the snitch strategy makes the algorithm always process each read/write request by accessing local replicas in priority, which tends to lower the overall latency, but replica updates in remote data centers are usually not able to spread in time. The results under the setting of the 3_1_1 replica configuration with snitch is a powerful evidence for above explanations.

4.1.4 Impact of Network Delay

In our experiments, we mainly focus on inter-data center delay because it impacts the performance in a significant way compared to intra-data center delay. The default injected one-way inter-data center delays in network are normally distributed with the average of $50ms$ and the standard deviation of $25ms$ (denoted $\mathbb{N}(50, 25^2)$). By varying μ from $10ms$ to $50ms$, and σ from $5ms$ to $25ms$ respectively, we obtain the results shown in Fig. 14 and Fig. 15 . We observe that, the average latency of read operations is basically in proportion to the average network delays across data centers. Besides, the consistency performance become weaker when the average network delays or jitters grow higher and higher. This is due to, not only higher network delay would make each operation duration stay longer, but also larger jitters or variances of delays would aggravate the occurrence of out-of-sync replicas. Then, the joint effects of above both lead to frequent atomicity violation patterns.

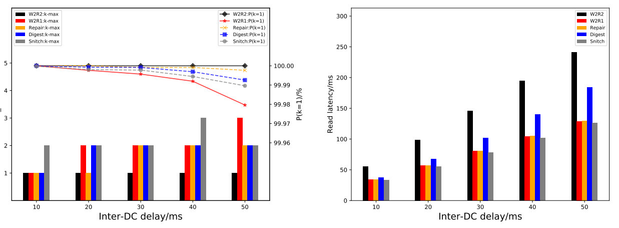

Fig. 14. Consistency and latency results by tuning the average value of inter-DC delay ($\mu = 10/20/30/40/50ms$, $\sigma = \frac{\mu}{2}$, accordingly).

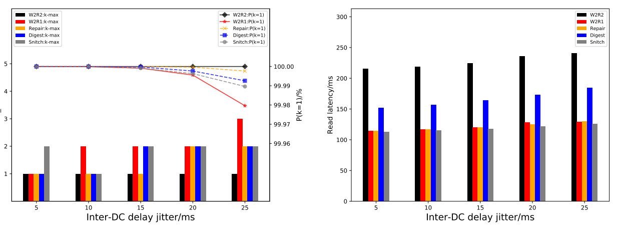

Fig. 15. Consistency and latency results by tuning inter-DC delay jitter ($\mu = 50ms$, $\sigma = 5/10/15/20/25ms$).

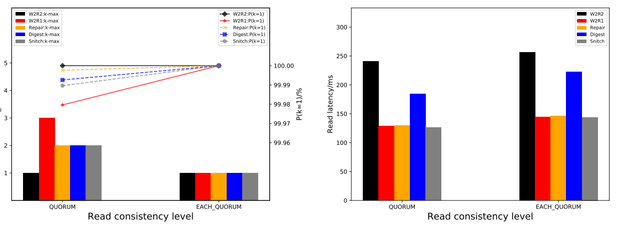

Fig. 16. Consistency and latency results by tuning read quorum level.

4.1.5 Impact of Read Quorum Level

We explore two types of read quorum level in Cassandra: QUORUM and EACH_QUORUM [8]. The level of QUORUM

8. Cassandra also provides a special quorum level: LOCAL_QUORUM. The level of LOCAL_QUORUM only requires to access a majority of replicas in the local data-center, which speeds up a lot while does actually not meet the requirement of majority-replication. Therefore, it is out of scope for this work.

TABLE 7
ASC tradeoff under the default setting

	k_{max}	$\mathbb{P}(k=1)$	$\mathbb{P}(k=2)$	$\mathbb{P}(k=3)$	$\mathbb{P}(staleness)$	Read latency
W2R2	1	100%	0	0	0	100%
W2R1	3	99.9796%	0.0203%	0.0001%	0.0204%	53%
Snitch	2	99.9896%	0.0104%	0	0.0104%	52.3%
Digest	2	99.9926%	0.0074%	0	0.0074%	76.5%
Repair	2	99.9977%	0.0023%	0	0.0023%	53.9%

requires to access a majority of replicas from all data-centers, which is actually the same as our theoretical model. The level of EACH_QUORUM requires to access a majority of replicas in each data-centers, which requires more replicas for responses compared to our theoretical model. As is shown in Fig. 16, EACH_QUORUM promises better consistency performance in average at the cost of extra latency; QUORUM lowers the average latency at the sacrifice of a little bit lower atomicity rate. The tradeoff between consistency and latency is well presented in this experiment.

4.2 Consistency-Latency Tradeoff in Different Algorithms

From all above experimental results, we observe that distinct algorithms vary in the consistency-latency tradeoff. Here, we summarize some valuable insights.

4.2.1 Consistency

We first compare the data consistency provided by W2R1 and W2R2. No matter under what circumstance, W2R2 provides atomicity all the time, while W2R1 (with / without optimization) does not. The W2R1 algorithm produced non-atomic traces with k up to 4 in terms of k-atomicity, although the probability of atomicity violation is quite low. Specifically, the W2R1 algorithm produced stale reads with the probability lower than 0.3% in our experiments under all circumstances, and lower than 0.02% under most circumstances. For example, from the experimental results under the default setting in Table 7, the rate of 2-atomic reads is lower than 0.03%, and 3-atomic reads even lower, not to mention higher values of k in terms of k-atomic reads.

Next, we make a comparison among different W2R1 optimizations. W2R1 with repair performs best among all W2R1 algorithms, and its probability of consistent reads in average is merely a little inferior than W2R2. W2R1 with digest or snitch doesn't perform so well as W2R1 with repair, but better than W2R1 without any optimization in overall. Above all, we believe that W2R1 (with / without optimization) rarely violates atomicity, and still promises atomicity most of the time.

4.2.2 Read Latency

As expected, W2R1 costs lower read latency than W2R2 in average. Among different W2R1 optimizations, W2R1 with digest has the highest read latency in average. We speculate about two reasons. Firstly, the mechanism of using digest requests requires two round-trips of communication sometimes. Specifically, in the first communication round-trip of read, only one replica would be requested with full data, and all others with digest. Once there exists some inconsistent replica responded, an extra round-trip for data retransmission is required. Secondly, compared to raw data collection, the digest request reduces data size in communication but requires extra work for data hashing in replica servers, which takes extra time. For W2R1 with repair, it leads to approximately the same or just slightly higher read latency compared to W2R1 without repair. This is due to that it applies asynchronous mechanism to repair stale replicas in the background after the response of the read, which has nearly no effect on the read response time. As for W2R1 with snitch, it gains the highest efficiency for reads mainly because it always selects the replicas with high proximity and route requests to them, which significantly decreases latency.

4.2.3 Consistency-Latency Tradeoff

W2R2 provides atomicity all the time at the sacrifice of extra communication round-trip for write-back in reads, which inevitably leads to high read latency. By omitting the second round-trip of read, W2R1 without any optimization gains lower latency but worse data consistency, although it still guarantees atomicity most of the time.

Based on the theoretical W2R1 algorithm, the optimization in Cassandra can help speedup request process by selecting replicas smartly(*snitch*), provide stronger consistency through replica synchronization(*repair*), or reduce communication data size(*digest*). W2R1 with snitch has the lowest read latency due to its smart routing, but provides the weakest consistency among three. W2R1 with repair performs better in providing atomicity without increasing much read latency since it puts repair work in the background. W2R1 with digest performs worse than W2R1 with repair but better than W2R1 with snitch in data consistency provided, while its read latency is a lot higher than the other two.

Anyway, the W2R1 algorithm with or without optimization achieves probabilistic atomicity with well-bounded staleness as well as the almost strong consistency in the common case. Specially, W2R1 with *repair* performs significantly well. The major wisdom of this optimization lies in the *divergence-oriented communication*, i.e., only when inconsistency has been found will the algorithm take extra action for repair. In this way, one communication round-trip is enough for those consistent responding replicas. From this perspective, W2R1 with *repair* can be regarded as a combination of W2R2 and W2R1. Moreover, repairing replicas in the background asynchronously is an efficient design to provide both strong consistency and low latency.

References

[1] P. B. Gibbons and E. Korach, "Testing shared memories," *SIAM Journal on Computing*, vol. 26, no. 4, pp. 1208–1244, 1997.

[2] H. Wei, Y. Huang, and J. Lu, “Probabilistically-atomic 2-atomicity: Enabling almost strong consistency in distributed storage systems,” *IEEE Trans. Comput.*, vol. 66, no. 3, pp. 502–514, Mar. 2017. [Online]. Available: https://doi.org/10.1109/TC.2016.2601322

[3] S. M. Ross, *Introduction to probability models*. Academic press, 2014.