

Research on Fault Tolerant Mechanism of Payment Order State Machine Based on Raft Consensus Algorithm

Jiayue Hu

Electrical and Computer Engineering, New York University, Brooklyn, NY, 11201, USA

Keywords: Raft consensus algorithm; payment order; state machine replication; bounded replay; version fence; idempotency

Abstract: To address the uncertainty and delayed event interference caused by master node failures when the state application is completed but the response has not yet returned in the Raft payment order service, this paper proposes a bounded consistent replay mechanism, Raft-BRR. This mechanism, based on order-level request identity, introduces external event version context, dual representation of historical results and current state, version fencing, and a two-level result lifecycle to maintain request semantics under limited storage conditions. Experiments were conducted on a three-node prototype, including post-application response loss, cross-master node retries, old version events, and window sensitivity. Results show that Raft-BRR achieves a median TRSR of 96.4% and 93.8% in F1 and F2 scenarios, respectively, a dual-state response integrity rate of 95.7%, and zero old version event error acceptances. Compared to a strong idempotent caching baseline, its throughput decreases by 8.2%, and P95 latency increases by 19.5%. These results demonstrate that Raft-BRR can improve the interpretability and version security of the payment order state machine layer at a controllable resource cost.

Introduction

The digital payment order service connects transaction acceptance, channel processing, status callbacks, and exception compensation. After a payment request completes status application, the master node may exit before sending the response, making it impossible for the client to determine whether the original request has taken effect. When the client retries with the new master node, if the system only regenerates the response based on the current state, the initial processing result may be replaced; if a late callback is still generated based on the old state, it may also push the order migration again. These types of problems usually do not cause Raft replica divergence, but they reduce the certainty of business results and increase the costs of compensation, reconciliation, and dispute resolution.

State machine replication provides a fault-tolerant basis for distributed services by executing deterministic commands in the same order through multiple replicas. Existing research points out that the internal state recovery consistency does not mean that the external response semantics are also consistent, and the replay of requests and the return of results after a failure still need to be processed by the application layer [1]. Related studies have also analyzed the fault boundary and

liveness issues of replication services [2][3], and the research on composite state machines further illustrates that unified application semantics are needed between different state objects [4]. P-Raft and LRD-Raft improve the master node selection and log replication path respectively [5][6], but these methods mainly focus on consensus efficiency and cannot directly handle the historical results, subsequent states and late events of payment orders.

Payment orders may simultaneously receive user requests, channel callbacks, compensation tasks, and timed events. Stable request identifiers and result caches can identify retries of the same request, but it is difficult to determine whether an event arriving with a new identifier is based on an expired state. Research on transactional state handling also shows that the system needs to coordinate duplicate messages, out-of-order events, and state recovery within a unified transaction boundary [7]. Long-term storage of complete results will also increase persistence and query overhead [8], while directly deleting records will lose summary conflict information. Research on payment protocols and payment networks further points out that confirmation results and failure retries will affect the judgment of transaction completion [9][10].

To address the aforementioned issues, this paper studies bounded consistent replay of "applied but not responded" requests for payment orders under Raft master node switching conditions. This paper does not modify the Raft election and log replication mechanisms, nor does it discuss end-to-end exactly-once processing for third-party channels. The study constructs order-level request identities and external event version contexts, incorporating dual-state responses, version fencing, and two-level result lifecycles into a single process. Experiments examine post-application response loss, cross-master node retries, isolation of old version events, and changes in the replay window to evaluate request semantic guarantees and their resource costs.

1. Definition of the Bounded Replay Problem for Payment Orders

1.1 State Sets and Transition Constraints

This paper abstracts the transfer processing status in the payment order service as an internal execution state of a state machine, without discussing account balance, settlement finality, and external channel side effects. RECEIVED, RESERVED, COMMITTED, and ABORTED correspond to Mojaloop FSPIOP TransferState; SETTLED is an order service state added in this paper for the settlement stage, used to indicate that the transfer has been submitted and subsequent settlement has been completed. This state is not part of the FSPIOP TransferState enumeration. The state set is defined as follows:

$$S = \{\text{RECEIVED}, \text{RESERVED}, \text{COMMITTED}, \text{ABORTED}, \text{SETTLED}\}$$

RECEIVED indicates the request has been accepted, RESERVED indicates payment resources have been reserved, COMMITTED indicates the transfer has been submitted, ABORTED indicates the request has been rejected or terminated, and SETTLED indicates the application layer's subsequent settlement has been completed. The state transition function is represented as follows:

$$\delta(s, e, v, r, h) \rightarrow (s', v', \text{result})$$

In the formula, s and s' represent the states before and after the migration, e represents the input event, v represents the order version observed when the event is generated, r represents the stable request identifier, and h represents the normalized payload summary. Research on fund transfer objects shows that the order of operations and legality constraints jointly affect the consistency of the distributed state, and the newly arrived event cannot be interpreted without the state conditions at the time of its generation [12]. Accordingly, this paper requires that the state change belongs to the set of legal migrations, ABORTED and SETTLED cannot be rolled back; when the same

request identifier corresponds to different summaries, the system must reject the request; when the event carries an expired version, the system must not advance the order. The relevant content is shown in Figure 1 and Table 1.

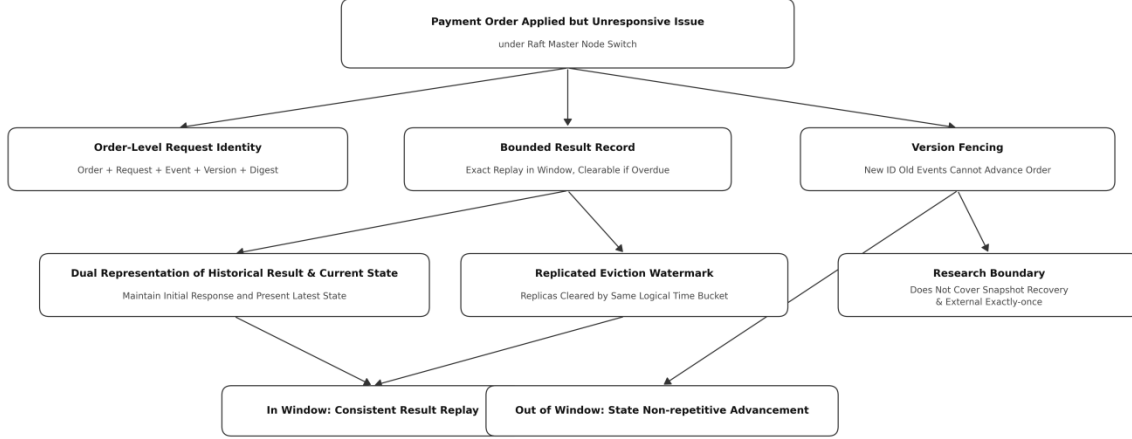


Figure 1. Research boundary and constraint relationships of bounded consistent replay mechanism

Table 1. Payment Order Status-Event-Version Decision Matrix

Current status	Input Events	Request conditions	Target state	Output
RECEIVED	RESERVE	Version consistent, new request	RESERVED	APPLY
RECEIVED or RESERVED	ABORT	Version consistent, new request	ABORTED	APPLY
RESERVED	COMMIT	Version consistent, new request	COMMITTED	APPLY
COMMITTED	SETTLE	Version consistent, new request	SETTLED	APPLY
any state	Same request within window	Abstract Consistency	State unchanged	REPLAY
any state	Full record or tombstone hit	Same identifier, different summary	State unchanged	REQUEST_CONFLICT
any state	Same request for tombstone date	Abstracts consistent, complete results have been cleaned	State unchanged	RESULT_EXPIRED
any state	New logo event	Version expired	State unchanged	VERSION_CONFLICT
any state	external callback	Version context missing	State unchanged	VERSION_UNKNOWN
Final state	Illegal follow-up incident	any	State unchanged	ILLEGAL_TRANSITION

1.2 Multi-subject identity requests and fault windows

Payment orders may simultaneously receive user submissions, channel callbacks, compensation tasks, and timed events. Different entities cannot reliably share the same client session and locally incrementing sequence numbers; therefore, this paper uses order identifier, stable request_id, event type, expected_version, and payload_hash to jointly describe the request identity. The business

initiator generates the `request_id` and retains this identifier during retries; `payload_hash` is used to identify different content carried by the same identifier; `expected_version` records the order version observed at the time the event is generated.

$$I = \langle \text{order_id}, \text{request_id}, \text{event}, \text{expected_version}, \text{payload_hash} \rangle$$

User requests and internal tasks directly read the order version upon creation. External channel callbacks typically do not carry the internal version; therefore, the adaptation layer saves the `ObservedVersionContext` with `transferId` and `source_event_id` as keys when issuing channel requests or registering pending events. Upon callback arrival, the adaptation layer restores `expected_version` based on the associated identifier; if the mapping is missing or expired, the system returns `VERSION_UNKNOWN` and does not use the current version for backfilling. This rule separates the replay of identical business requests from the handling of new identifiers for old events.

The study sets up three core scenarios: F1, F2, and F3. F1 terminates the master node after the state machine completes the atomic application but before the response enters the sending queue, verifying whether the initial result has already been replicated. F2 retries with the same global `request_id` and a new transmission session after a new master node is generated, verifying the continuity of results across master nodes. F3 first records the version at the time of event generation, then advances the order version with another event, and finally delivers the original event, verifying the isolation effect of the new identifier for the old event. Submissions that have not yet been applied, full node restarts, and snapshot installation interruptions are not included in this study's scenarios.

1.3 Bounded Guarantees and Evaluation Attributes

Let the logical time for the application to complete the request be t_r , the client retry time be t_a , and the result replay window be W_r . The system guarantees accurate return of historical results only if the following formula is true:

$$t_r - t_a \leq W_r$$

Within the replay window, requests with the same identity and identical digests return the initial normalized result. After the complete result expires, compact tombstones continue to store the request identifier, digest, and application version; requests with the same identifier but different digests return `REQUEST_CONFLICT`, and requests with the same identifier and identical digests return `RESULT_EXPIRED`. After the tombstone expires, the system no longer provides request-level result guarantees; order versions and valid migration rules are still responsible for preventing old commands from advancing to the next state. This paper validates three security properties: historical result determinism, single-event effective impact, and version isolation, and evaluates availability by the proportion of accurate replays completed within a given timeout.

2. Bounded Consistent Replay Mechanism for Payment Orders

2.1 Deterministic Commands and Atomic Applications

This paper names the proposed mechanism Raft-BRR (Raft-based Bounded Result Replay). Client requests are normalized before being written to the Raft log as follows:

$$C = \langle o, r, e, v, h \rangle$$

In the formula, o represents the order identifier, r represents the stable request identifier, e represents the payment event, v represents the expected version, and h represents the SHA-256

digest calculated after encoding the key fields in a fixed order. The digest does not contain local time, thread number, or node address, and the state machine does not read replica-private information. When the command application is committed, the order status, order version, complete result or tombstone metadata, and AppliedIndex are committed in the same bbolt write transaction. After the transaction is committed, the result takes effect at the application layer.

Algorithm 1. Bounded Result Replay and Version Fencing for Payment Orders

Require: command $C = \langle o, r, e, v, h \rangle$; committed log index i ; replicated epoch epoch; full-result window W_r ; tombstone window W_t .

Ensure: a deterministic response q and at most one effective order-state transition.

```

1:   $F \leftarrow \text{FullResult.lookup}(r)$ 
2:  if  $F \neq \emptyset$  then
3:    if  $F.\text{order\_id} = o$  and  $F.\text{payload\_hash} = h$  then
4:       $q \leftarrow \text{ComposeReplay}(F.\text{original\_result}, \text{Order}[o].\text{state})$ 
5:    else
6:       $q \leftarrow \text{REQUEST\_CONFLICT}$ 
7:    end if
8:     $\text{AtomicCommit}(\text{AppliedIndex} \leftarrow i)$ 
9:    return  $q$ 
10:  end if
11:  $T \leftarrow \text{Tombstone.lookup}(r)$ 
12: if  $T \neq \emptyset$  then
13:   if  $T.\text{order\_id} = o$  and  $T.\text{payload\_hash} = h$  then
14:      $q \leftarrow \langle \text{RESULT\_EXPIRED}, \text{Order}[o].\text{state} \rangle$ 
15:   else
16:      $q \leftarrow \text{REQUEST\_CONFLICT}$ 
17:   end if
18:    $\text{AtomicCommit}(\text{AppliedIndex} \leftarrow i)$ 
19:   return  $q$ 
20: end if
21:  $(\text{state}, \text{version}) \leftarrow \text{Order.read}(o)$ 
22: if  $v = \perp$  then
23:    $\text{reason} \leftarrow \text{VERSION\_UNKNOWN}$ 
24: else if  $v \neq \text{version}$  then
25:    $\text{reason} \leftarrow \text{VERSION\_CONFLICT}$ 
26: else if  $(\text{state}, e) \notin T\_state$  then
27:    $\text{reason} \leftarrow \text{ILLEGAL\_TRANSITION}$ 
28: else
29:    $\text{reason} \leftarrow \text{NONE}$ 

```

```

30: end if
31: if reason  $\neq$  NONE then
32:    $q \leftarrow \text{NormalizeReject}(\text{reason}, \text{state}, \text{version})$ 
33:    $R \leftarrow \text{BuildFullResult}(o, h, q, \text{state}, \text{version}, i, \text{epoch} + W\_r, \text{epoch} + W\_r + W\_t)$ 
34:    $\text{AtomicCommit}(\text{FullResult}[r] \leftarrow R, \text{AppliedIndex} \leftarrow i)$ 
35:   return  $\text{ComposeResponse}(q, \text{state})$ 
36: end if
37:  $(\text{next\_state}, q) \leftarrow \text{ApplyTransition}(\text{state}, e)$ 
38:  $\text{next\_version} \leftarrow \text{version} + 1$ 
39:  $R \leftarrow \text{BuildFullResult}(o, h, q, \text{next\_state}, \text{next\_version}, i, \text{epoch} + W\_r, \text{epoch} + W\_r + W\_t)$ 
40:  $\text{AtomicCommit}(\text{Order}[o] \leftarrow (\text{next\_state}, \text{next\_version}), \text{FullResult}[r] \leftarrow R, \text{AppliedIndex} \leftarrow i)$ 
41: return  $\text{ComposeResponse}(q, \text{next\_state})$ 

```

Both complete results and tombstones are indexed by request_id. Conflicting requests must not overwrite existing digests, and rejected results are only recorded under the request's own identifier. Garbage collection commands and business commands are written to the Raft log together, so the log order determines the window boundaries: if the garbage collection command is applied first, the system returns RESULT_EXPIRED; if the retry command is applied first, the system still returns the complete historical results.

2.2 Dual Representation of Historical Results and Current State

The strongly generalized idempotent caching baseline stores the global request_id, payload_hash, and complete processing result, and adopts the same bounded lifecycle as Raft-BRR. Payment orders may continue to enter the SETTLED state after the initial COMMIT. If the system only returns the cached result, the client cannot obtain the latest order status; if the system only returns the current status, the initial semantics of the original request will be changed. Therefore, Raft-BRR defines the result record within the window as:

$$R(r) = \langle o, h, \text{code}, \text{result_state}, \text{result_version}, \text{index}, \text{expire_epoch} \rangle$$

When the original COMMIT request is retried after the order has already entered the SETTLED state, the system returns:

$$\text{Response} = \langle \text{original_result} = \text{COMMITTED}, \text{current_state} = \text{SETTLED} \rangle$$

`original\_result` retains the initial business result, while `current\_state` provides the latest view of the order. This response neither modifies historical results nor misleads the client into believing that the order is still in the old state. The `index` is only used to associate the result with the state machine application location and does not serve as a long-term audit proof function.

2.3 Replicable Lifecycle Management

Raft-BRR uses a full result window (W_r) and a compact tombstone window (W_t) to manage the result lifecycle. The W_r stores the normalized response, result status, application version, and log index. Upon W_r expiration, the system compresses the record into a tombstone (request_id, order_id, payload_hash, applied_version, outcome_class, expire_epoch). The default W_t time is 24 hours. The tombstone continues to identify digest conflicts and expired replays, but does not save the full response. Upon W_t expiration, the system deletes the tombstone, retaining only the order

status and version.

GC(cutoff_epoch)

The master node periodically commits the `GC_FULL(cutoff_epoch)` and `GC_TOMBSTONE(cutoff_epoch)` commands. All replicas are compressed and deleted in the same log order, without automatically eliminating replicas based on the local clock. For identical requests within W_r , the first result is returned; for identical requests between W_r and W_t , `RESULT_EXPIRED` is returned; outside of W_t , request-level conflict detection is no longer guaranteed. Commands carrying the old `expected_version` will still be rejected by version fences; therefore, result cleanup only changes the range of information that can be returned, not the conditions for legal state transitions.

Let the number of complete results be $N_r(W_r)$, the average record size be $\bar{s}_r$, the number of tombstones be $N_t(W_t)$, and the average tombstone size be $\bar{s}_t$. Then the online replay metadata overhead is expressed as:

$$M(W_r) = N_r(W_r) \cdot \bar{s}_r$$

This paper does not set up a permanent archive layer, nor does it study historical result queries beyond W_t . B2 and Raft-BRR use the same W_r and W_t to avoid lifecycle differences affecting the comparison. The main difference between the two methods is that Raft-BRR also saves order version, version context, and dual-state response fields.

2.4 Version Fence

Each change in valid status will cause the order version to monotonically increase:

$$v_{t+1} = v_t + 1$$

After an order transitions from version 2 to version 3, late `ABORTs` generated based on version 2 must return `VERSION_CONFLICT` even with a new `request_id`. If an external callback lacks an `ObservedVersionContext`, the system returns `VERSION_UNKNOWN`. Version fencing does not depend on whether the complete result or tombstones are still retained; therefore, two-level result cleanup does not weaken order state constraints.

3 Research Methods

3.1 Research Design and Workload

This study employed controlled fault injection and parameter sensitivity experiments, using running sessions as paired blocks. Each method executed 2000 warm-up requests and 20,000 metering requests per block. Five fixed random seeds (20260301, 20260317, 20260402, 20260419, and 20260507) were used, each running in three independent sessions, resulting in 15 paired blocks. Each method processed a total of 300,000 metering requests. Normal migration accounted for 60%, while F1, F2, and F3 accounted for 15%, 15%, and 10%, respectively. Event duration covered 24 hours, with 70% of retry delays occurring within 1 minute, 20% between 1 and 10 minutes, 8% between 10 and 60 minutes, and 2% between 60 and 180 minutes.

3.2 Comparison Methods and Experimental Environment

B1 uses Raft and a valid transition matrix, handling retries only based on the current order state. B2 employs a strongly generic idempotent cache, storing a stable global `request_id`, `payload_hash`,

and complete result, and uses the same Wr/Wt lifecycle as Raft-BRR, but does not use order version, external version context, or dual-state responses. Raft-BRR adds order version fencing, ObservedVersionContext, and a dual representation of historical results and current state to B2. The mechanism components of the compared methods are shown in Table 2.

Table 2. Mechanisms of the comparison methods

mechanism	B1	B2	Raft-BRR
Legal state transition matrix	√	√	√
Stable global request_id and digest	—	√	√
Full result cache	—	√	√
Compact tombstones and bounded elimination	—	√	√
Order version fence	—	—	√
External event version context	—	—	√
Historical Results - Current State (Dual Representation)	—	—	√

The experimental prototype version is BRR-Proto 1.0.0, using Go 1.26.5, go.etcd.io/raft/v3 v3.7.0, and go.etcd.io/bbolt v1.5.0. Three independent processes communicate via TCP loopback, with each node configured with an independent Raft WAL and bbolt file. The runtime environment is Ubuntu 24.04.4 LTS, Linux 6.8, 4 vCPUs, 16 GB of memory, and a general-purpose cloud SSD (3000 IOPS, 125 MB/s). The bbolt contains four buckets: order_state, full_result, request_tombstone, and meta. State, version, and result metadata are atomically committed with the AppliedIndex via a single Update transaction, with NoSync kept off. The median command payload is 384 bytes, reaching 512 bytes with P95. Each persistent batch contains a maximum of 32 log entries, or executes fdatasync after a 2 ms wait. The experiment settings were: ElectionTick=10, HeartbeatTick=1, MaxInflightMsgs=256, client concurrency was 64, and request timeout was 800 ms; Wr was 30 min by default, Wt was fixed at 24 h, and the sensitivity experiment set Wr to 5, 30 and 90 min.

3.3 Indicators and Statistical Scope

Error history replay rate (ERR) is defined as:

$$ERR = \frac{N_{wrong\ historical\ result}}{N_{returned\ replay}}$$

The Time-Limited Precision Replay Success Rate (TRSR) represents the percentage of requests that meet the window criteria to return their first result within 800 ms.

$$TRSR = \frac{N_{exact\ replay\ within\ 800ms}}{N_{eligible\ retry}}$$

The Old Version Event Error Acceptance Rate (SAER) represents the proportion of events carrying outdated versions that are incorrectly executed.

$$SAER = \frac{N_{accepted\ stale\ event}}{N_{stale\ event}}$$

Auxiliary metrics include dual-state response integrity rate, VERSION_UNKNOWN ratio, throughput, P95 latency, and storage scale per 10,000 online replay metadata records. F1 and F2 are calculated separately, without sharing a denominator. Only requests that actually hit the fault hook and meet the preconditions are included in the main analysis; replica non-convergence, state

abnormalities, or index errors caused by the tested method are all counted as failures. Ratios and performance metrics are calculated using 15 running sessions as the statistical unit, reporting the median, quartile range, and run-level pairings to the Bootstrap 95% range, with 10,000 resampling attempts. The main text does not treat requests within the same session as independent samples.

3.4 Fault Coverage Boundary

The F1 hook is located between the completion of the bbolt transaction commit and the writing of the response to the sending queue. After the hit, it sends SIGKILL to the main process. F2 uses the same commit point. After the new master node is generated, the system retries with the same global request_id, the same payload_hash and a new transmission session. F3 writes to the ObservedVersionContext when the event is generated. After another event completes the version advancement, the original event is delivered. The failure time is driven by a fixed seed. The experiment records the term, index, request_id, application time and response time. The experiment does not cover persistent network partitions, random packet loss, slow disk writes and long GC pauses. Chaos engineering is only used as a supplement to external validity[11].

4. Research Results

4.1 F1: Response lost after application

The 15 runs generated a total of 45,000 F1 candidate requests, of which 43,614 completed atomic applications and hit the pre-response fault window. B1, B2, and Raft-BRR returned 30,651, 42,043, and 41,987 first results precisely within 800 ms, respectively, with median TRSRs of 70.5%, 96.5%, and 96.4%, and quartile ranges of 69.1%–71.6%, 95.9%–97.0%, and 95.7%–96.9%, respectively.

The difference in operational level between Raft-BRR and B2 was -0.12 percentage points, with a 95% range of -0.27 to 0.04 percentage points. Among requests that had already returned responses, neither B2 nor Raft-BRR observed replay of erroneous historical results; B1 had 12,284 responses inconsistent with the initial results, and no secondary state transitions caused by the same requests occurred[13-14].

4.2 F2: Retry across master nodes

The 15 runs generated a total of 45,000 F2 candidate requests, of which 43,187 were included in the statistics after the new master node was generated. B1, B2, and Raft-BRR returned 29,280, 40,542, and 40,542 exact results, respectively, with median TRSRs of 67.6%, 94.1%, and 93.8%, and quartile ranges of 66.5%–69.2%, 93.0%–94.8%, and 92.8%–95.2%, respectively.

The average difference in operational level between Raft-BRR and B2 is 0.00 percentage points, with a 95% range of -0.15 to 0.15 percentage points. Of the 16,942 requests that continued to evolve after the initial result, Raft-BRR returned both the historical result and the current status within 800 ms in 16,210 requests, with a median dual-state response completeness rate of 95.7% and an quartile range of 94.8% to 96.4%. B2 only returns the historical result, while B1 only returns the current status[15-16].

4.3 Event Isolation in Previous Versions

Each method received 30,000 events carrying expired versions and using the new request_id. B1 and B2 incorrectly received 900 and 802 events respectively, with median SAERs of 3.0% and

2.7%, and interquartile ranges of 2.7%–3.4% and 2.3%–3.1%, respectively[17].

Raft-BRR did not observe any older version events being accepted. Of these, 29,748 events returned `VERSION_CONFLICT`, and 252 returned `VERSION_UNKNOWN` due to missing or expired `ObservedVersionContext`. The median proportion of `VERSION_UNKNOWN` events was 0.8%, with an interquartile range of 0.6%–1.1%. No replica state divergence was observed in any of the three methods.

4.4 Replay Window Sensitivity

When W_r was set to 5, 30 and 90 min respectively, the median F1/F2 combined TRSR of Raft-BRR was 87.6%, 95.1% and 95.9% respectively, and the interquartile ranges were 85.9%–89.8%, 94.1%–95.9% and 95.0%–96.6% respectively.

Under the three window types, the storage size per 10,000 online replay metadata entries was 0.93, 3.18, and 8.54 MB, respectively; the median throughput was 1249, 1215, and 1186 req/s, respectively; and the P95 latency was 18.9, 19.8, and 20.9 ms, respectively. When W_r increased from 5 min to 30 min, TRSR improved by 7.5 percentage points; when W_r increased from 30 min to 90 min, TRSR improved by 0.8 percentage points.

No differences in the replica result index were observed under any of the three window settings. The number of failures in compressing the complete result to the tombstone was 2, 3, and 4, respectively, and the relevant batches were retried in the next GC command. The result coverage and online replay metadata scale under different replay windows are shown in Figure 2.

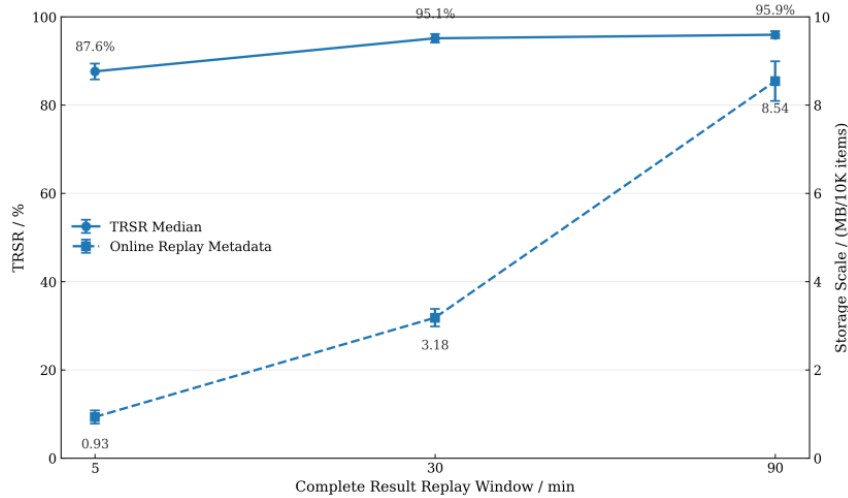


Figure 2. Result coverage and online replay metadata scale under different replay windows.

4.5 Performance Cost

With default $W_r=30$ min and $W_t=24$ h, the median throughput of B1, B2, and Raft-BRR is 1425, 1333, and 1215 req/s, respectively, with quartile ranges of 1382–1458, 1301–1365, and 1197–1245 req/s. The average run-level reduction of Raft-BRR relative to B2 is 8.2%, with a 95% range of 7.7%–8.6%.

The median P95 latency for the three methods were 15.1, 16.7, and 19.8 ms, respectively. The average performance increase of Raft-BRR over B2 was 19.5%, with a 95% accuracy range of 17.9%–21.0%. The throughput and latency differences in the 15 operating groups showed some fluctuation, and Raft-BRR did not demonstrate a performance advantage. The main experimental results are shown in Table 3.

Table 3 Main Experimental Results

index	B1	B2	Raft-BRR
F1-TRSR median/%	70.5	96.5	96.4
F2-TRSR median / %	67.6	94.1	93.8
ERR/%	Approximately 29.0	0	0
Dual-state response integrity rate / %	—	—	95.7
SAER median/%	3.0	2.7	0
VERSION_UNKNOWN/%	—	—	0.8
Median throughput / (req/s)	1425	1333	1215
P95 latency/ms	15.1	16.7	19.8
Online replay metadata (MB/10,000 records)	—	2.41	3.18

5. Discussion

Experimental results show that a strongly generalized idempotent cache can handle duplicate submissions of the same request quite well, and Raft-BRR has not established a stable advantage in accurate result replay rate. Therefore, the contribution of the proposed method is not to improve Raft consensus performance, nor to redesign the duplicate request cache. Raft-BRR focuses on how the system can simultaneously retain the initial processing result and the current state as payment orders continue to evolve, and how late events arriving with new identifiers are constrained by the original state conditions. These results demonstrate that consensus logs can unify command order, but the interpretation of payment order results still requires application-layer semantic support.

The main difference between B2 and Raft-BRR lies in the scope of event recognition. B2 can identify recorded duplicate requests, but it cannot determine whether a new request originates from an expired order status. Raft-BRR supplements this judgment with order version and external event context, ensuring that late events are checked before state transition. Some events return VERSION_UNKNOWN due to missing version context, indicating that the mechanism does not assume all callback information is always complete. Rejecting unacknowledged events reduces the risk of erroneous transitions, but also incurs some availability loss. This phenomenon demonstrates that version security depends not only on state machine rules but also on the quality of event association information stored by the channel adaptation layer.

Replay window experiments show a clear trade-off between result coverage and online state size. Extending the retention time beyond the main retry period only yields limited replay benefits, while metadata size and tail latency still increase. Therefore, result records are not suitable for long-term storage in Raft hot states; the window length should be determined based on the actual retry distribution, order processing cycle, and storage budget. For services where requests are inherently idempotent, do not require simultaneous return of historical results and current state, and where late events are controlled by other rules, B2 can meet the requirements at a lower cost.

This study is still limited by the experimental environment and the scope of failures. A three-process loopback network and a single cloud SSD cannot reflect cross-server communication, persistent network partitioning, long disk tails, and GC pauses. Deterministic failure hooks can accurately hit the "applied but not responded" window, but cannot replace random failures and long-term load testing. The current version context still relies on transferId and source_event_id to maintain event association; missing information in cross-channel links may affect request availability. Future research should incorporate random process termination, network latency, packet loss, and disk disturbances in a physical multi-node environment, and use real payment retry data to verify the configuration relationships between replay windows, tombstone cycles, and external archiving.

6. Conclusion

This paper addresses the issue of payment requests failing to receive a response after state application has been completed following a Raft master node switchover, proposing a Raft-BRR bounded consistent replay mechanism. Research results show that a strongly generalized idempotent cache can maintain a high accurate replay rate. The main value of Raft-BRR lies in utilizing order versions and dual-state responses to handle subsequent order evolution and new identifiers for old events. The two-level result lifecycle further defines the timeframe for result guarantees, clearly defining the boundaries between digest conflict detection, state safety, and storage scale. This mechanism trades certain throughput and tail latency for interpretable semantics of payment order requests, making it suitable for order services with complex callback sources, continuously evolving states, and the need to distinguish between historical results and the current state.

References

- [1] Fragkoulis M, Carbone P, Kalavri V, Katsifodimos A. A survey on the evolution of stream processing systems[J]. *The VLDB Journal*, 2024, 33(2): 507-541. DOI: 10.1007/s00778-023-00819-8
- [2] Freitas T, Soares J, Correia M E, et al. Deterministic or probabilistic? A survey on Byzantine fault tolerant state machine replication[J]. *Computers & Security*, 2023, 129: 103200. DOI: 10.1016/j.cose.2023.103200.
- [3] Bravo M, Chockler G, Gotsman A. Liveness and latency of Byzantine state-machine replication[J]. *Distributed Computing*, 2024, 37(2): 177-205. DOI: 10.1007/s00446-024-00466-4.
- [4] Alves C M, De Souza M A, Idalino T B, et al. Composing state machine replication[J]. *Journal of Internet Services and Applications*, 2025, 16(1): 629-644. DOI: 10.5753/jisa.2025.5914.
- [5] LU S F, ZHANG X Y, ZHAO R K, et al. P-Raft: An efficient and robust consensus mechanism for consortium blockchains[J]. *Electronics*, 2023, 12(10): 2271. DOI: 10.3390/electronics12102271.
- [6] Wang, B. (2025). *Methods of Load Optimization for Computer Systems Based on Physical Principles*.
- [7] Guo, X. (2025, March). *Research on Blockchain-Based Financial AI Algorithm Integration Methods and Systems*. In *2025 IEEE International Conference on Electronics, Energy Systems and Power Engineering (EESPE)* pp. 816 – 821. IEEE.
- [8] Guo, X. (2025, April). *Research on Financial Trading Algorithms Based on Deep Reinforcement Learning*. In *2025 IEEE 3rd International Conference on Control, Electronics and Computer Technology (ICCECT)* pp. 1711 – 1715. IEEE.
- [9] Yu, X. (2026). *Exploration of Multi-Channel Conversion Path Optimization Methods Based on A/B Testing*.
- [10] Jing, X. (2025, November). *In-Depth Analysis of the Optimization Decision Model of Financial Investment Portfolio Based on Machine Learning Algorithms in Python Scikit-Learn Library*. In *International Conference on Computational Technologies for Research in Data Analytics* pp. 130 – 140. Cham: Springer Nature Switzerland.
- [11] Liu, X. (2026). *Research on the Application of Artificial Intelligence in Consumer Privacy Data Protection*. *Procedia Computer Science*, 279, 956-965.
- [12] Liu, X. (2026). *Construction of Consumer Data Privacy Protection System Based On Blockchain Technology*. *Procedia Computer Science*, 282, 2082-2091.

- [13] Huijie Pan. *Discussion on Low-Latency Computing Strategies in Real-Time Hardware Generation*. *International Journal of Neural Network* (2025), Vol. 4, Issue 1: 57-64.
- [14] Yang, Y. (2026). *Causal Inference-based Identification of Incremental Effects in Digital Advertising and Optimization Pathways for Resource Allocation*. *Advances in Computer and Communication*, 7(2).
- [15] Yuan, Y. (2026). *Research on Memory Management and Dynamic Reasoning Methods for Intelligent Agents Based on Large Language Models*.
- [16] Han, W. (2025, November). *An Empirical Study on Building a High-precision Forecasting System for Financial Market Trends Based on the PyTorch Deep Learning Framework*. In *International Conference on Computational Technologies for Research in Data Analytics* (pp. 518-528). Cham: Springer Nature Switzerland.
- [17] Sun, L. (2026). *Consistency Optimization Strategies for Distributed Advertising Systems Under Cross-regional Deployment*. *Advances in Computer and Communication*, 7(2).