An Optimized Key-Value Raft Algorithm for Satisfying Linearizable Consistency

Xiao Liu, Zhao Huang*, Quan Wang, Nan Luo
School of Computer Science and Technology
Xidian University, Xi'an, China
Email: cantona@stu.xidian.edu.cn; {z_huang, qwang, nluo}@xidian.edu.cn

2022 International Conference on Networking and Network Applications (NaNA) | 978-1-6654-6131-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/NANA56854.2022.00096

***Abstract*—Nowadays, the distributed systems have been widely applied in a variety of fields. However, this raises more concerns on reliability. Consensus algorithm is an important measure to ensure reliability of distributed systems, but its strong consistency may reduce the performance, resulting in cluster failure or even downtime. To this end, we propose an accelerated log backtracking optimization Raft algorithm, called ALB-Raft. It can improve the performance of traditional raft algorithm by enabling the backward tracker to update quickly. In particular, to achieve strong consistency, we construct a fault-tolerant distributed key-value (KV) service which conforms to the linearizable semantics. The experimental results illustrate that, when compared to the traditional raft algorithm, the proposed ALB-Raft consensus algorithm can resolve 20% of hundreds of log entry conflicts. Moreover, the ALB-Raft algorithm can also avoid the linear increase in the number of messages with the aggravation of log conflicts to ensure strong consistency.**

***Index Terms*—Raft; Linearizability; Key-Value; Consensus**

I. Introduction

Distributed systems have the natural concurrency feature. However, it also faces various complex failure scenarios such as server failure, message delay, loss, out-of-order, repetition, network partitioning [1], [4]. Moreover, the services provided by distributed system need to hide errors from applications [1]. They also need to ensure service availability even in the case of a partial server failure [1], [2]. However, this requires distributed services to automatically handle server crash and recovery, and its corresponding fault-tolerant processing is a major challenge faced by distributed systems [1], [3], [5].

Byzantine failures [1] and fail-stop failures [2] are the two most commonly used failure models in distributed systems [2], [3]. The most general method to deal with these two failures is to implement a reliable consensus algorithm [6], [10]. However, existing mainstream consensus algorithms such as Paxos [7], ViewStamped Replication [8], and Zab [9] have some defects. First, these algorithms are difficult to understand in the algorithms themselves which are not easy to build the actual system. Second, building a reliable distributed system needs the additional expansion or performance optimization of the above consensus algorithms.

Due to the above problems among the existing consensus algorithms, Raft algorithm has now been selected as the main solution for fault tolerance of distributed systems by both academic and industrial communities [4], [10], [16]. Generally speaking, Raft algorithm is a consensus algorithm based on the idea of Paxos algorithm which is well understandable and easy to implement [10]. But in a real distributed system, users usually have the requirements of high performance and strict data consistency. However, the origin Raft algorithm may fail to achieve this, while conducting multiple leader elections will significantly reduce the performance of distributed system. Especially in the case of network partition or node downtime, it is also difficult to ensure effective log replications by using origin Raft. Therefore, it is necessary to design a follower's log update mechanism with strong consistency requirement for the origin Raft algorithm to improve the performance.

In order to improve the performance of the Raft algorithm, this paper proposes an accelerated log backtracking optimization Raft algorithm, called ALB-Raft. It can effectively reduce the number of message communications required to resolve conflicts with hundreds of log entries, thereby improving the performance of traditional Raft. Moreover, we have also built a Key-Value service, which can effectively improve the fault tolerance of distributed systems while ensuring strong consistency. In particular, the main contributions of this paper are as follows.

- We propose a high-performance optimized Raft Algorithm.
- We achieve Linearizability by implementing Key-Value service based on ALB-Raft algorithm. The strong consistency of ALB-Raft algorithm can be effectively verified by the Porcupine Linearizability detector [18].
- The experiment results show that the proposed ALB-Raft algorithm can effectively improve the performance of distributed systems while ensuring the strong consistency requirements.

The rest of the paper is organized as follows: Section II describes related work. Section III presents our design of ALB-Raft algorithm. Section IV constructs a fault-tolerant Key-Value service that conforms to strong consistency semantics. The analysis of system performance is provided in Section V. Conclusion and future works appear in Section VI.

II. Related Work

The "gold standard" algorithm in distributed consensus is the Paxos protocol [6]. However, Paxos has two significant drawbacks. First, the definition of Paxos is a bit abstract, thus it may take considerable efforts for user to fully understand [7]. The second problem with Paxos is that it may fail to provide

978-1-6654-6131-3/22/$31.00 ©2022 IEEE
DOI 10.1109/NaNA56854.2022.00096

a good foundation for building practical implementations [7], [10]. ViewStamped Replication [8] and Zab [7] are two other well-known consensus algorithms. These two algorithms are equivalent to the variants of the Paxos algorithm. However, they have the same problems as the Paxos [10].

Raft is a new distributed consensus algorithm proposed by Diego in 2014 [10]. Origin Raft is designed to be very understandable and the performance is also as good as Paxos. But it still has some drawbacks. First, it ignores the higher performance requirements for fault tolerance in distributed systems. Second, the traditional Raft consensus algorithm cannot ensure the strong consistency of the distributed system, which will result in the problems of log inconsistency, message loss and data disorder of the clusters. Therefore, some improvement works attempt to solve the problem with the origin Raft algorithm which have lower performance and weaker consistency.

Since multiple invalid leader elections and unnecessary node communications will cause long delay and low throughput of the distributed systems, some efforts have been done to solve these problems to increase origin Raft algorithm performance. For example, Du et al. [11] proposes a novel leader election optimized Raft method to maintain low request latency and high throughput per second. Xu et al. [12] proposes a weighted Raft algorithm with a new node communications model. This approach reduces the latency in achieving final consistency in IoT applications. Choumas et al. [13] designs a leader election time optimized Raft protocol which significantly reduce the latency between the leader and all the other nodes.

On the other hand, it is difficult for distributed systems using the Raft algorithm to achieve high performance while maintaining strong consistency. Thus, academic community has made a lot of efforts to ensure the strong consistency by optimizing the origin Raft algorithm. For example, Wang et al. [14] proposes a Key-Value Raft (KV-Raft) algorithm to keep the strong consistency and reduce operations latency at the same time. Liu et al. [15] designs an optimized Raft algorithm with a new leader selection scheme to keep the blockchain network maintaining strong consistency. Huang et al. [16] implements a Raft-based Hybrid Transaction and Analytical Process (HTAP) database with strong consistency and high performance.

However, most of the currently optimized Raft consensus algorithms only take the performance or strong consistency as a factor to improve the reliability and fault tolerance of distributed systems. Only a few studies consider improving both performance and strong consistency at the same time. However, these efforts also have ignored strong consistency correctness verification for distributed systems. Therefore, this paper proposes an optimized Key-Value Raft algorithm with Linearizability verification to improve the reliability and fault tolerance with high performance and strong consistency.

III. Raft Optimization Implementation

The traditional Raft consensus algorithm mainly includes log replication, leader election, persistence and safety [10]. However, during the implementation of the origin Raft protocol, frequent leader crashes or multiple network partitions will result in serious inconsistency between the follower logs and the leader log. Hence, we propose an accelerated log backtracking algorithm called ALB-Raft to solve these problems. In particular, Fig.1 exhibits the generic process diagram of the proposed ALB-Raft method.

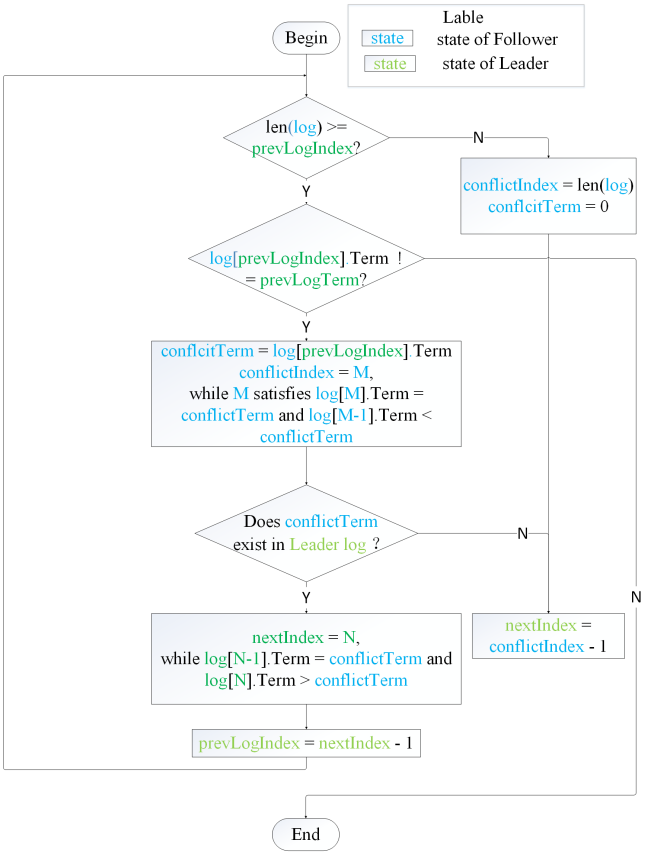

Fig. 1. Flow-process diagram of the ALB-Raft algorithm.

The goal of our optimization is to reduce the number of rejected *AppendEntries Remote Procedure Calls (RPCs)* of leader, while ensuring that the number of entries carried in each follower's *AppendEntries RPCs* are as small as possible. The *AppendEntries RPCs* define as the Remote Procedure Calls of heartbeat and communications of log replication which are sent by the leader to the followers of origin Raft [10]. That is to say, the ALB-Raft allows the old followers to update quickly than origin Raft protocol. Based on this goal, the optimized ALB-Raft scheme is mainly to reduce log conflicts between follower and leader nodes. Note that the inconsistency of the log entries between the followers and the leader is called log conflict [10]. Processing the inconsistency of the log entries in the leader will reduce the number of entries carried in each follower's *AppendEntries RPCs*. Log conflict handled by follower will reduce *AppendEntries RPCs* rejected by the leader. Therefore, the ALB-Raft is divided into two

parts: handling the conflicted logs in follower and processing the leader's conflicted logs.

Firstly, ALB-Raft handles conflicted log which happened in follower. We compare the index of log between the follower and leader. When the index *prevLogIndex* exists and the term *log[prevLogIndex].Term* doesn't equal to the one of leader, it means that the conflicted log exists in follower. We find the conflicted log in the follower. The follower sends the conflicted log to the leader. The leader handles the conflicted log and keeps the next log to be updated consistent with the follower. Therefore, we can significantly reduce the inconsistency logs in follower which are rejected by the leader through the *AppendEntries RPCs*.

Secondly, the conflicted log may occur in the leader. When a log conflicted term *conflictTerm* exists in the leader, it means that the leader has the conflicted log. Thus, we reduce the leader conflict log by updating the log entries in leader and follower. Therefore, the log of leader is consistent to the follower. Thus, decreasing the numbers of leader conflicted logs can reduce the number of entries carried in each follower's *AppendEntries RPCs*.

IV. Linearizability Key-Value Service Implementation

After finishing the design of ALB-Raft consensus protocol, we have only completed the client-server model based on two layers of *Clerk-Raft*. However, this model only guarantees weak consistency in distributed systems and will not keep consistent at every moment of the logical clock. This will make it difficult for all server nodes to ensure the consistency of logs across the entire system in the case of network partitions and node failures. Therefore, we add the Key-Value service layer on the basis of the original two-layer structure, thereby improving the log consistency when a fault occurs. To sure stronger consistency, we build a Key-Value service with linearizable semantics to improve the fault tolerance of distributed systems.

A. Architecture Implementation of Key-Value Service

When a node in the Raft layer appears the downtime failure, the Raft layer mainly transmits the latest logs of all nodes to the Key-Value layer through the message passing mechanism *applyCh*. After the node logs of the Key-Value layer are updated by *applyCh*, the logs of the server node will be inconsistent. The master node (Leader) needs to tell the client that the logs of all nodes in the key-value layer need to be consistent. Thus, we deliver the consistent messages to clients through a notification mechanism *notifyCh*.

The architecture of the Key-Value service is shown in Figure 2. It is typically a client-server architecture. When the Raft layer has a failure of the master node downtime, we send all the logs from the Raft layer to the Key-Value layer through the message passing mechanism *applyCh*. Then, we update the logs of all servers in Key-Value layer. However, the logs of these servers may be inconsistent. The client then handles the consistency request from the leader server by the notification mechanism *notifyCh*. Finally, the client replicates the leader's log to all other servers via the Raft module again. Therefore, each *kvserver* performs the operations in the Raft log sequentially and applies these commands to its own Key-Value database. As a result, all servers maintain the same copy of the Key-Value state machine. The logs of all server node are consistent.

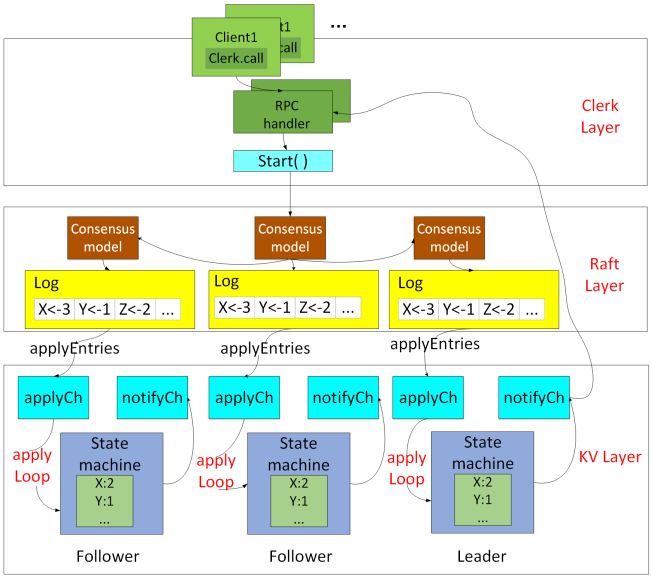

Fig. 2. Key-Value service architecture.

B. Implementation of Linearizability Semantics based on Key-Value Service

When completing the basic Key-Value service, we need to constrain the log orders of all nodes to achieve the requirements of Linearizability. Since the client can only perform one operation at a time [6], all operations keep a straight line at logical timestamps [17]. However, the current Raft algorithm can execute one command multiple times. Therefore, we need to implement a Linearizability consistency constraint scheme for all nodes based on Key-Value service.

1) Client Linearizability Constraint:

From perspective of client, the Linearizability system satisfies the following constraints:

(1) Single client operations are sequential: each operation of each client is executed after the previous operation returns. It represents the operations in the same client are sequential. There are no concurrent operations.

(2) If the operations from different clients are concurrent, they may return old or new values. But after any read returns a new value, all subsequent reads' operations must return a new value from server.

2) Client Deduplication Scheme:

For the client, how to handle repeated operation requests from servers via the client is a linearizable consistency requirement. To this end, a client operation request deduplication scheme is designed to achieve this goal as described follows.

First, the server requests the client to perform an operation. Here we assume that a client can only make one request call at a time. When the request sequence number of an operation *seq* from the server is greater than the most recently updated operation sequence number *latestSeq*, the client performs this operation. Second, client transmits the operation to the server via RPC. The server implements the operation and writes it to its own log. Finally, the client verifies that the operation is not duplicated. Otherwise, when the *seq* is small than the *latestSeq*, this operation is duplicated. The client does not perform the repeat operation.

V. Experiments and Results

The performance test mainly includes three parts: origin Raft and ALB-Raft performance comparison, Key-Value service simulation verification, and linearizable semantics verification. We utilize a consensus algorithm simulation framework based on Go language to simulate. Then, we give the performance results of the Raft implementation and the Key-Value service in the test framework. Finally, we test the Linearizability consistency correctness with the open source Linearizability verification module Porcupine [18].

A. ALB-Raft Optimization Performance Comparison

To effectively verify the performance of ALB-Raft, we evaluate its performance under two failure scenarios. The two scenarios are the network partition failure and unreliable network failure.

First, we compare the performance of Raft algorithm and ALB-Raft under the network partition failure. The results are shown in Fig.3. In this scenario, there are five nodes in the distributed system. The simulation framework divides the entire distributed system into two networks. The servers of each partition execute the Raft algorithm successively and submit 100 logs. Finally, when the partition is repaired, we need to commit the log again. Simulation framework checks whether the entire process is conformed successfully to the Raft protocol specification. There may be nearly a hundred conflicted logs between the leader and the follower during this process of checking the Raft algorithm. As can be seen from Fig.3, we execute the consensus algorithm multiple times under network partition failure, Origin Raft needs to send more RPCs to reach consensus. However, the ALB-Raft protocol can significantly reduce the number of rejected RPCs. Therefore, compared to the original Raft algorithm, ALB-Raft protocol can reduce RPCs by 20% when resolve 100 conflicted entries.

Second, we need to test ALB-Raft performance under unreliable network failure. In this scenario, the follower's logs may update far behind the leader. The inconsistencies between node logs become more serious. Unreliable network failure leads to inconsistencies in the logs of the distributed system. The more operations submitted, the more RPCs in distributed need to be sent in this failure. Fig.4 shows the comparison of operation 20 times in the original Raft algorithm (blue columns) and the ALB-Raft (Accelerated Log Backtracking) algorithm (red columns).

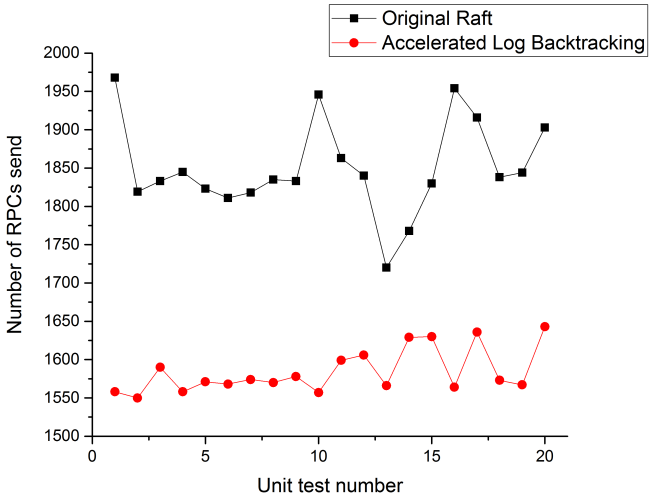

Fig. 3. Performance comparison of ALB-Raft and Raft algorithm under the network partition..

In this test scenario, as the number of operations submitted increases, the number of conflicted logs between followers and leaders increase. Therefore, client node needs to send more messages to resolve the conflicted logs. When the distributed system needs to perform more operations, it increases the number of communications between nodes. Thus, this will bring more increase in the number of RPCs. We can see from Fig.4 is as follows. When the same number of operations are performed, we can see that ALB-Raft algorithm sends fewer numbers of RPCs than Raft. It means that ALB-Raft optimization can ensure fast backtracking when the consistency checking fails. ALB-Raft algorithm can effectively reduce the number of rejected RPCs with the increasing number of operations. It means the ALB-Raft algorithm is basically remains stable.

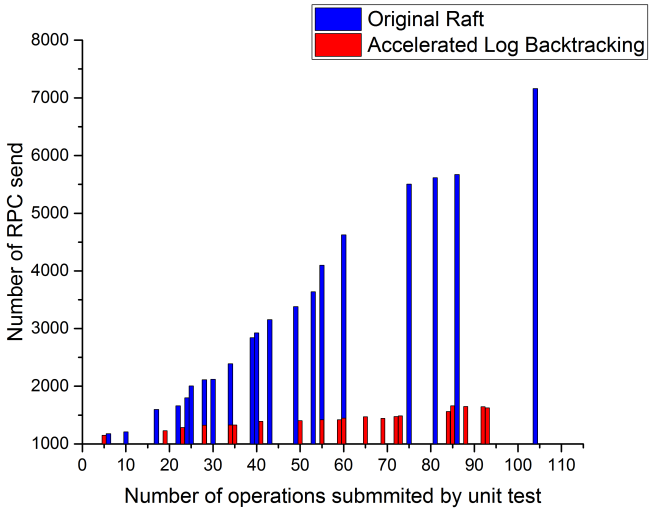

Fig. 4. Performance comparison of ALB-Raft and Raft algorithm under the unreliable network.

B. Key-Value Service Test Results and Performance

The Key-Value test runs under the simulation test framework as shown in Fig.5. The Key-Value service layer we implemented can run correctly under fault conditions such as client downtime, node downtime, network partition, etc. All the servers can send RPCs to client normally. It means that the client can communicate with the servers in Key-Value layer normally. Distributed systems can effectively maintain consistency in the presence of failure.

The experimental results also show that the Key-Value service can provide reliable service for the client under various fault conditions. It guarantees normal operation even in an extreme multiple mixed fault scenario. As we can see from the Fig.5 that the results of all fault points are passed. It proves that Key-Value service is highly available.

```
Performance     time  kvservers  RPCs   Ops
Test: [a] one client (KV) ...
  ... Passed --  15.5    5     8849    976
Test: [b] many clients (KV) ...
  ... Passed --  15.5    5    14680   1872
Test: [c] unreliable net, many clients (KV) ...
  ... Passed --  22.5    5     3722    323
Test: [d] concurrent append to same key, unreliable (KV) ...
  ... Passed --   2.9    3      308     52
Test: [e] progress in majority (KV) ...
  ... Passed --   1.2    5       90      2
Test: [f] no progress in minority (KV) ...
  ... Passed --   1.0    5      107      3
Test: [g] completion after heal (KV) ...
  ... Passed --   1.3    5       87      3
Test: [h] partitions, one client (KV) ...
  ... Passed --  22.3    5     9075    949
Test: [i] partitions, many clients (KV) ...
  ... Passed --  23.5    5    14402   1686
Test: [j] restarts, one client (KV) ...
  ... Passed --  21.8    5    10451    965
Test: [k] restarts, many clients (KV) ...
  ... Passed --  22.0    5    16998   1991
Test: [l] unreliable net, restarts, many clients (KV) ...
  ... Passed --  25.1    5     6289    386
Test: [m] restarts, partitions, many clients (KV) ...
  ... Passed --  28.9    5    17769   1924
Test: [n] unreliable net, restarts, partitions, many clients (KV) ...
  ... Passed --  29.9    5     5266    226
Test: [o] unreliable net, restarts, partitions, many clients, linearizability checks (KV) ...
  ... Passed --  30.3    7    13112    534
PASS
ok      kvraft  264.572s
```

Fig. 5. Key-Value service test result.

C. Linearizability Verification

The general steps for Linearizability verification are running a group of clients concurrently to perform a series of operations in the distributed system. The validation module randomly injects errors and failures into the system. We should record the history of all operations. Finally, the operation history is verified by the validation module to determine whether it meets the requirement of Linearizability. The client generates a set of history records based on Key-Value service. The history is shown as Fig.6.


```
{"Input":{"Op":2,"Key":"6","Value":"x 3 0 y"},"Call":1422660,"Output":{"Value":""},"Return":870767444}
{"Input":{"Op":2,"Key":"10","Value":"x 3 1 y"},"Call":870771703,"Output":{"Value":""},"Return":895885775}
{"Input":{"Op":0,"Key":"1","Value":""},"Call":895893359,"Output":{"Value":""},"Return":920649394}
{"Input":{"Op":2,"Key":"1","Value":"x 3 2 y"},"Call":928658124,"Output":{"Value":""},"Return":958048000}
{"Input":{"Op":2,"Key":"5","Value":"x 3 3 y"},"Call":958054304,"Output":{"Value":""},"Return":983369155}
```


Fig. 6. The history of the Key-Value Service.

Each row represents a history of a client operation under this test. If two columns of the ”Call” and ”Return” return data overlap, the two operations are concurrent. If there is no intersection, they are sequential. After getting the historical records, we use Porcupine Linearizability detector [18] to verify them. We build a Linearizability consistency verification model through the historical records. The model verification process is shown as Fig 7.

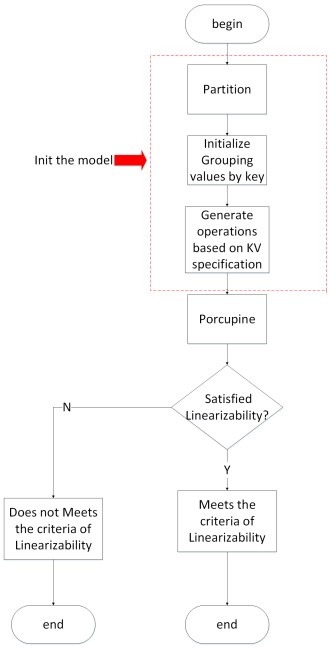

Fig. 7. The sequential operation model of the Key-Value service.

We can see from the above Fig.7 are as follows. The values under the same Key are grouped into the same group and corresponding operations are added in the order of Key in this model. We can see that this model converts the original input log specification into a Key-Value model based on sequential operations. At this point, we have completed the initialization of the operation history model which meets the requirement of Key-Value service. When the model does not meet the Linearizability, the established Key-Value service does not meet the requirements of strong consistency. When we execute the verification model successfully, the execution results of running the Porcupine detector is shown in Fig. 8.

```
qiuyu@qiuyu:~/code/DS/go/mycode/linearizability$ go test
parse history log: operations.history to operations...
parse done!
Check linearizability...
Linearizability check done in 9.154917ms; result: true
history is linearizable!
PASS
ok      _/home/qiuyu/code/DS/go/mycode/linearizability  0.062s
```

Fig. 8. Results of running the Porcupine detector.

As can be seen from Fig.8, Porcupine first parses the list of history records that met its format requirements from the history file. Then, it tests the history records according to the object and history list. The results show that the history records passed the Porcupine test. The Key-Value service in this paper satisfied the linearizable semantics.

VI. Conclusion

This paper implements a high-performance optimized Raft algorithm, named as ALB-Raft. The ALB-Raft algorithm effectively reduces the number of conflicted logs and improves the performance of the distributed system. We also implement the Key-Value service based on ALB-Raft. The Key-Value service satisfies the requirement of strong consistency. The experimental results show that the ALB-Raft algorithm has good performance. The results also show that we can also ensure strong consistency in distributed systems by building efficient Key-Value services. But the current work about the ALB-Raft is not perfect. The method in this paper has shortcomings in the aspects of low consensus performance, low efficiency of log replication, and lack of effective consistency verification. In the future, we plan to accomplish works mainly includes:

(1) Raft protocol implementation with higher optimized performance.

(2) We use batch log and pipeline log replication approach to optimize the ALB-Raft algorithm. It will improve the efficiency of log replication.

(3) Using Jepson distributed verification framework to verify the Linearizability of the system with higher standard.

VII. Acknowledgements

This work was supported in part by the National Natural Science Foundation of China under Grant 61972302, in part by the Fundamental Research Funds for the Central Universities under Grant XJS220306, in part by the Natural Science Basic Research Program of Shaanxi under Grant 2022JQ-680, in part by the Key Research and Development Program of Shannxi Province under grant numbers 2021GY-086 and 2021GY-014, and in part by the Key Laboratory of Smart Human Computer Interaction and Wearable Technology of Shaanxi Province. *Zhao Huang is the corresponding author*.

References

[1] Lamport L, Shostak R, Pease M. The Byzantine generals problem[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1982, 4(3): 382-401.

[2] Schlichting R D, Schneider F B. Fail-stop processors: an approach to designing fault-tolerant computing systems[J]. ACM Transactions on Computer Systems (TOCS), 1983, 1(3): 222-238.

[3] Schneider F B. Implementing fault-tolerant services using the state machine approach: A tutorial[J]. ACM Computing Surveys (CSUR), 1990, 22(4): 299-319.

[4] Gu X S, Wei H F, Qiao L, Huang Y. Raft with Out-of-Order Executions[J]. International Journal of Software and Informatics,2021,11(4):473-503.

[5] Burrows M. The Chubby lock service for loosely-coupled distributed systems[C]//Proceedings of the 7th symposium on Operating systems design and implementation. 2006: 335-350.

[6] Lamport L. The part-time parliament[J]. ACM Transactions on Computer systems, 1998, 16(2): 133-169.

[7] Lampson B. The ABCD's of Paxos[C]//PODC. 2001, 1: 13.

[8] Liskov B, Cowling J. Viewstamped replication revisited[J]. 2012.

[9] Junqueira F P, Reed B C, Serafini M. Zab: High-performance broadcast for primary-backup systems[C]//2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN). IEEE, 2011: 245-256.

[10] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm[C]//2014 USENIX Annual Technical Conference (Usenix ATC 14). 2014: 305-319.

[11] Du H, , Zhu D, Sun Y, et al. Leader Confirmation Replication for Millisecond Consensus in Private Chains[J]. IEEE Internet of Things Journal, 2021.

[12] Xu X, Hou L, Li Y, et al. Weighted RAFT: An Improved Blockchain Consensus Mechanism for Internet of Things Application[C]//2021 7th International Conference on Computer and Communications (ICCC). IEEE, 2021: 1520-1525.

[13] Choumas K, Korakis T. On using Raft over Networks: Improving Leader Election[J]. IEEE Transactions on Network and Service Management, 2022.

[14] Wang Y, Wang Z, Chai Y, et al. Rethink the Linearizability Constraints of Raft for Distributed Key-Value Stores[C]//2021 IEEE 37th International Conference on Data Engineering (ICDE). IEEE, 2021: 1877-1882.

[15] Liu Z M, Hou L,Zheng K,Zhou Q H, Mao S W. A DQN-based Consensus Mechanism for Blockchain in IoT Networks[J]. IEEE Internet of Things Journal, 2021.

[16] Huang D, Liu Q, Cui Q, et al. TiDB: a Raft-based HTAP database[J]. Proceedings of the VLDB Endowment, 2020, 13(12): 3072-3084.

[17] Herlihy M P, Wing J M. Linearizability: A correctness condition for concurrent objects[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1990, 12(3): 463-492.

[18] A fast linearizability checker written in Go. [Online]. Available:https://github.com/anishathalye/porcupine