vRaft: Accelerating the Distributed Consensus Under Virtualized Environments

Yangyang Wang[1,2] and Yunpeng Chai[1,2(✉)]

[1] Key Laboratory of Data Engineering and Knowledge Engineering, MOE, Beijing, China
ypchai@ruc.edu.cn
[2] School of Information, Renmin University of China, Beijing, China

Abstract. In recent years, Raft has been gradually widely used in many distributed systems (e.g., Etcd, TiKV, PolarFS, etc.) to ensure the distributed consensus because it is effective and easy to implement. However, because the performance of the virtual nodes in cloud environments is usually heterogeneous and fluctuant due to the "noisy neighbor" problem and the cost efficiency, the strong leader mechanism makes the Raft protocol encounter a serious performance challenge. Specifically, when the performance of the leader node is low, the whole system performance will descend accordingly since both the write and the read requests serving will be blocked by the slow leader processing. Aiming to solve this problem, we proposed a modified version of Raft specially optimized for virtualized environments, i.e., vRaft. It breaks Raft's strong leader restriction and can fully utilize the temporarily fast followers to accelerate both the write and the read requests processing in a virtualized cloud environment, without affecting the linearizability guarantee of Raft. The experiments based on the virtual nodes in Tencent Cloud indicate that vRaft improves the throughput by up to 64.2%, reduces average latency by 38.1%, and shortens the tail latency by 88.5% in a typical read/write-balanced workload compared with Raft.

1 Introduction

For distributed systems, the consensus algorithm is a key component to guarantee data consistency and system reliability, especially in the presence of system faulty processes. Traditionally, the Paxos [1] protocol is employed by many distributed systems to achieve the distributed consensus. However, Paxos is particularly difficult to understand and implement in practical distributed systems. In this case, the Raft protocol [2,3], which was proposed in 2014, is easy to be comprehended and realized, and thus soon has been widely adopted by many practical distributed systems like Etcd [4], TiKV [5], and PolarFS [6]. Although the sequential execution limitation weakens the performance of Raft compared

C. S. Jensen et al. (Eds.): DASFAA 2021, LNCS 12681, pp. 53–70, 2021.
https://doi.org/10.1007/978-3-030-73194-6_4

with Paxos, the multiple Raft groups (see Sect. 2.1) or the ParallelRaft [6] mechanism can improve the parallelism of operation processing and promote the performance. According to statistics from Raft's official website, as of November 2020, Raft has been used in 117 projects [7].

Motivation. In recent years, more and more distributed systems are deployed in cloud environments, i.e., in virtual machines (e.g., KVM [8], Xen [9], etc.) or containers (e.g., Docker [10]). And the CPU, memory, I/O, and network resources are isolated by tools like *cgroup* [11]. However, the nowadays technique cannot guarantee accurate performance isolation, the performance of a virtual node is highly affected by the other virtual nodes located on the same physical machine; this is called the noisy neighbor problem [12]. In addition, the emerging storage devices (e.g., SSDs or non-volatile memory (NVM)) have obvious performance advantages over the traditional ones. However, these new devices are usually much more expensive, so we may deploy them in only a subset of the clusters for cost efficiency. Therefore, the virtual nodes, even with the same configurations, often have different performance, and the performance of any node may fluctuate frequently. For example, when the same program runs 300 times in a virtualized environment, the performance difference is up to 60× or more [13]. Moreover, we rent two virtual nodes with exactly the same configuration from Tencent Cloud [14], but the I/O performance of these two nodes has 3 to 10 times difference, as Fig. 1 shows.

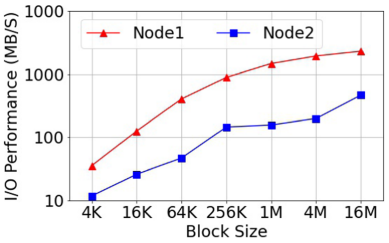


Fig. 1. The I/O performance gap between two virtual nodes with completely the same configuration. The tests were performed by using *fio* [15] with the block size setting ranging from 4 KB to 16 MB.

Considering the heterogeneous and unstable performance of virtual nodes, the distributed systems based on Raft have an important performance challenge: Raft adopts a strong leader mechanism to ensure the data consistency, i.e., the leader undertakes much more jobs compared with the followers and is the most critical part for performance. Once the leader locates on a slow node in a time period, the performance of the whole system will be slowed down (see Sect. 2.1 for more about Raft). For example, we have made some comparative experiments by forcing the leader to locate on the fastest node or the slowest node. The system throughput gap between the two configurations reaches 62.8% (see Sect. 2.2 for details).

Basic Idea. For the above problem appeared in virtualized environments, there are no existing solutions; for example, a common-sense method of migrating the leader to fast nodes introduces other additional problems (see Sect. 2.2 for more). Therefore, in this paper, we propose an improved version of the Raft protocol under the virtualized environment, i.e., vRaft, to solve the above problem and improve the performance. vRaft breaks the strong leader limitation on the basis of maintaining the same level data consistency, allowing fast followers to boost both the write and the read request processing. Two new mechanisms called Fastest Return (Sect. 3.1) and Optimal Read (Sect. 3.1) are proposed in vRaft, in order to fully take the advantage that some followers located on temporarily fast virtual nodes has fast progress and strong processing ability.

The comparison experiments between vRaft and Raft in the Tencent Cloud environment indicate that vRaft improves the throughput by 64.2% in a read/write-balanced workload, reduces the average latency by 38.1%, and shortens the tail latency by 88.5% at the same time. Furthermore, more experiments under different configurations (e.g., the numbers of replicas, system loads, and system scales) exhibit that vRaft is effective in various environments.

Our contributions in this paper are summarized as follows:

(1) *We identify the important new performance problem of Raft introduced by the virtualized environment.* Due to the heterogeneous and unstable performance of virtual nodes, if the performance of the node where the Raft leader locates is temporarily poor, the system performance will be deteriorated.
(2) *We solve the above problem of Raft by proposing a modified version of Raft, called vRaft.* vRaft breaks Raft's strong leader mechanism and thus can fully utilize the fast follower(s) to accelerate the request processing in a virtualized environment. And we prove that vRaft does not break the linear consistency guaranteed by Raft.
(3) *We improve the performance of an industrial-grade distributed key-value storage systems (i.e., TiKV) by incorporating vRaft to demonstrate its effectiveness.* Compared with Raft, vRaft promotes the throughput by 64.2%, shrinks the average latency by 38.1%, and reduces 88.5% tail latency for typical workloads.

The rest of this paper is organized as follows. Section 2 introduces the background of our research and the motivation of this paper. In Sect. 3, we present the design of our proposed vRaft. The implementation details and the evaluations of vRaft are described in Sect. 4, followed by the related work presented in Sect. 5. Finally, we conclude this paper in Sect. 6.

2 Background and Motivation

2.1 The Raft Protocol for Distributed Consensus

Traditionally, the Paxos [1] protocol is classical to ensure data consistency in distributed systems. However, Paxos is particularly difficult to understand and

implement. In this case, the Raft protocol [2], which is easy to be comprehended and realized, has been widely adopted by many practical distributed systems like Etcd [4], TiKV [5], and PolarFS [6] since it was proposed in 2014.

Write Process of Raft. Among the N copies of any data segment, one of them is elected as the leader replica according to Raft, while the other $N-1$ ones become followers. Raft's processing of write requests includes three key operations, i.e., *append*, *commit*, and *apply*, as shown in Fig. 2.

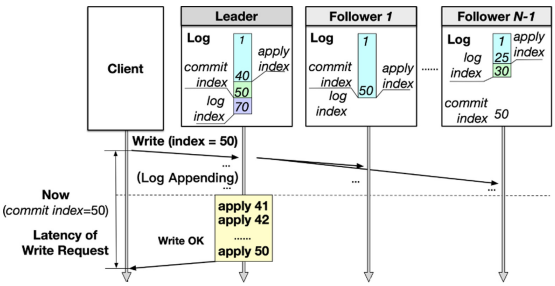


Fig. 2. Raft's procedure of processing write requests.

First, when a client sends a write request to the leader, the leader *appends* the content of the write request into the persistent log, and meanwhile the leader sends the content of the write request to all the other followers in parallel. When a follower receives the write request from the leader, it will also *append* the content into the log and then notify the leader after its appending procedure succeeds.

Subsequently, when the leader finds out that more than half of the replicas (maybe not including the leader itself) have successfully finished the append operations, the log is in the *committed* status and the *commit index* (i.e., the version id) of the replica is increased (e.g., from C to $C+1$). Note that the version of the latest log (a.k.a., *log index*) on a node is often newer than the *commit index*; this means that the latest content is not safe enough because only minority nodes have this content. For instance, as the example in Fig. 2 illustrates, the *log index* of the leader has already reached 70, while the *commit index* is only 50.

Then, after a written data is committed, the leader and all the followers start *applying* the log into the state machine. And after the leader's *apply* operation succeeds, the leader can return success to the client, and the *apply index* is increased. According to Raft, the log appending should be performed sequentially according to requests' arrival order. So do the log committing and the data applying. Therefore, the leader has to apply the previous updates (e.g., index 41–49 in Fig. 2) first before processing the target data (e.g., index 50).

Read Process of Raft.[1] In Raft, all the read requests are processed by the leader to ensure that the client would not get the out-of-date data. When the leader receives a read request from a client, it records the current *commit index* as the *read index* of the read request [3].

When the leader's *apply index* is no less than the *read index* of the read request, the leader can execute the read request immediately and return the result to the client. However, when its *apply index* is lower than the *read index*, some additional time-consuming operations should be performed first before processing the read request. As Fig. 3 plots, the *read index* is 50 while the *apply index* of the leader is only 40. So the leader has to apply the contents of 41–50 first and then process the read request, in order to ensure linear consistency.

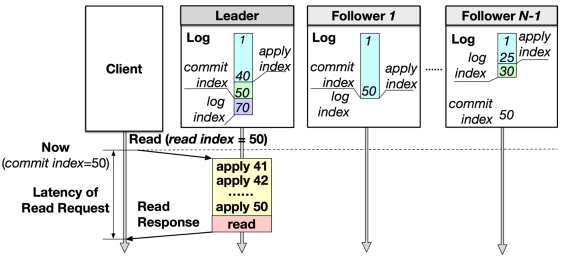


Fig. 3. Raft's procedure of processing read requests.

The four kinds of indexes used in Raft are summarized in Table 1.

Table 1. The four kinds of indexes used in Raft.

Index name	Descriptions
log index	Index of log that has been appended on a node
commit index	Index of log that has been appended by majority nodes
apply index	Index of log that has been written into status machine
read index	The commit index at the time a read request arrives

Multiple Raft Groups. If all the data are put into one Raft group, the system scalability is poor, because only N nodes can be used for N copies. In addition, all the Raft's operations will be executed sequentially, without parallelism. Therefore, practical systems usually adopt the solution of *multiple Raft groups*. i.e., the data are divided into many segments and the replicas of each data segment compose one independent Raft group. Figure 4 is an example of

[1] Raft's read process is not detailed described in the original paper [2], but in the doctoral thesis [3] of the author.

multiple Raft groups with the 3-copies setting, 6 nodes, and 4 Raft groups, where the operations of different Raft groups can be processed in parallel for higher performance.

Coupled with the multiple Raft groups, we can solve the performance problem of Raft caused by sequential processing of requests, making Raft comparable with Paxos in performance. Thus most Raft-based practical distributed systems adopt the multiple Raft groups such as TiKV and PolarFS. Note that all experiments in this paper are based on multiple Raft groups, and the size of each Raft group is usually about 100 MB.

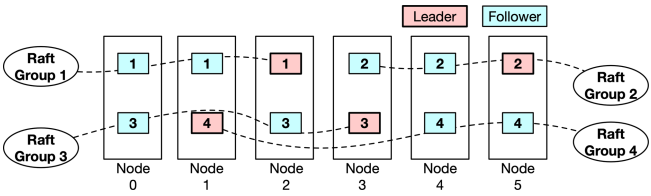


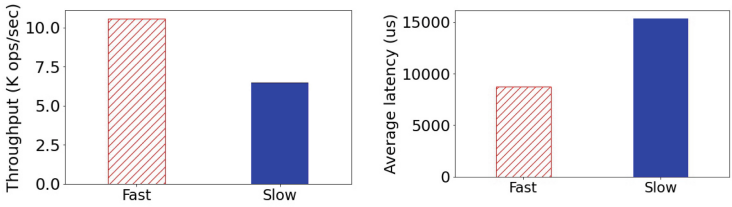
Fig. 4. An example of multiple Raft group.

2.2 Motivation

Raft Is Not Performing Well in a Virtualized Environment. The reason lies in that the leader processing affects the performance the most in Raft and the node performance is often heterogeneous and unstable in a virtualized environment. When a leader is locating on a temporarily weak node, it will slow down the whole Raft group significantly.

Specifically, for write operations, when the progress of the leader is slow, even if most nodes have already finished writing, we have to wait for the leader's accomplishment before replying to the client. For read requests, if the leader's *apply index* falls behind the *commit index* due to the poor performance of the leader node, the read request will not be executed until the *apply index* of the leader reaches the *read index* of the read request, even though other follower nodes can serve the read request already.

To illustrate the impacts of the leader node's performance. We wrote 10GB of data into a three-node cluster, forcing the leader to locate on the fastest node or the slowest node, respectively. The results exhibit the former leads to a 62.8% higher throughput and a 42.9% lower latency compared with the latter, indicating that the slow progress of the leader can result in significant performance declining (Fig. 5).



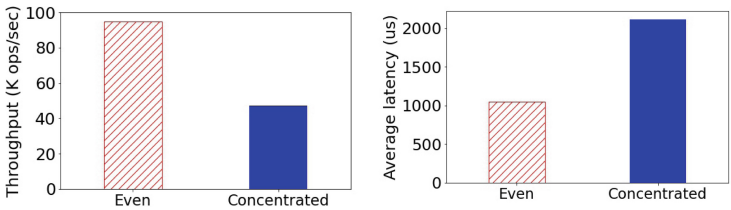
(a) Write Throughput (b) Write Latency

Fig. 5. The performance when the leader locates on the fastest or the slowest node.

The Leader Migration Solution Does Not Work. A common-sense idea of solving the slow leader problem is to migrate the leader replica to a fast node. However, there are some problems for this solution: First, it is different to measure the node performance accurately in real-time, because the software (e.g., a key-value engine) performance on a node is usually affected by multiple factors.

Second, the leader migration causes significant additional overhead, such as the latency brought by the leader election or the new leader fetching its missing logs from others. Especially in virtualized environments, the virtual node performance often fluctuates, leading to frequent leader switches and much overhead.

Finally, another problem of leader migration lies in the possible excessive leader concentration, which will weaken the parallelism of the read request processing and make the fast node become overloaded, resulting in performance degradation. For example, as Fig. 6 shows, we read 10GB of data in a real physical cluster, comparing the performance of the evenly distributed leaders and the concentrated leaders on the fastest node. The results illustrate that the evenly distributed leader solution has a 101% higher throughput and a 50.4% lower latency than the other one.



(a) Throughput (b) Latency

Fig. 6. The performance gap between the even and concentrated leader distributions.

Summary. According to the above discussions, we cannot solve the slow leader problem through the leader migration. Therefore, we should change the direction, i.e., fully utilizing the fast follower to accelerate the request processing in a

virtualized environment. We should make the follower replace part of the leader's work, breaking Raft's strong leader mechanism, thereby improving performance, but at the same time not destroying the linear consistency of Raft.

3 Design of vRaft

In this section, we will present the design of our improved Raft protocol, i.e., vRaft, which aims to boost both the write and the read request processing under the virtualized environments. We first present the basic idea of vRaft in Sect. 3.1. Then the algorithm design of vRaft will be given in Sect. 3.2 and the linearizability of vRaft will be discussed in Sect. 3.3.

3.1 Overview

In order to solve the slow leader problem under virtualized environments, vRaft boosts both the write and read processing by breaking the roadblock of the leader and creating new paths for request processing, without influence the linear consistency. Specifically, vRaft introduces two key components, i.e., Fastest Return and Optimal Read, to boost writing and reading, respectively.

Fastest Return. For the write request processing of the original Raft, if the progress of the leader is slow, even if most other nodes have finished writing, the client has to wait until the slow leader completes writing. Recall the example shown in Fig. 2, i.e., for a write operation with the index 50, since the *apply index* of the leader is only 40, older than the *write index*, we have to apply logs 41–50 to the state machine in order to finish this write operation. The massive applying operations slow down the write operation processing significantly.

However, a Raft group contains multiple nodes. Some of the follower nodes may be faster than the leader at the current time period in virtualized cloud environments. In this case, the follower should send its accomplished apply index to the leader when its apply index changes. Therefore, when the leader knows one of the followers has finished the applying phase, it can return success to the client ahead of time compared with the original Raft, even if the leader itself has not finished applying.

Example 3.1. *An write operation processing example of vRaft.* As Fig. 7 illustrates, the follower 1 notifies the leader that its apply index is 50, so the leader can return the response of the write request with index 50 to the client, even if the leader's own apply index is less than 50.

Optimal Read. Because we need to read the target data from the state machine, only the nodes with newer apply indexes compared with the read index can serve the read request. For reading, the performance problem of the original Raft lies in that when the leader node is temporarily slow in the virtualized environment and has a low apply index, we have to increase the leader's apply

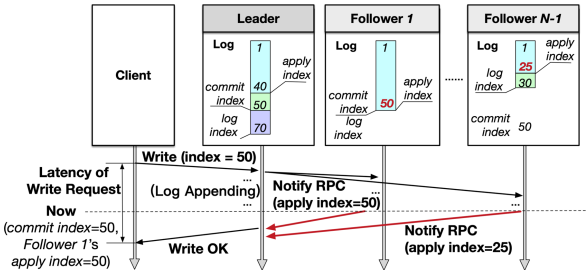


Fig. 7. vRaft's procedure of processing write requests.

index first before the request serving. Recall the example shown in Fig. 3, the read index is 50, and the temporarily slow leader which has an apply index of 40. So the read request will not be executed until the leader has applied the logs 41–50 to the state machine.

However, if there is a follower whose apply index is greater than or equal to the read index, it may be faster to redirect the read request to the follower for processing. Therefore, when the leader cannot serve the request immediately due to the low apply index, the leader checks whether there is a follower with a high enough apply index who can immediately process the read request. And the leader also needs to judge whether the time it waits for the apply index increment is greater than the time to redirect the read request to such a follower. If there are multiple followers that meet the condition, the leader will redirect the read request to the follower with the lowest pressure for processing.

Furthermore, the redirected read request must include the read index, and the follower who receives the read request must make sure the apply index not lower than the read index before executing the read request. In this way, even if an error occurs in the redirection, linear consistency can be guaranteed (See Sect. 3.3 for details).

Example 3.2 *An read operation processing example of vRaft.* As Fig. 8 plots, the leader finds that the time waiting for the apply index increment of itself is larger than the time to redirect the read request to the follower 1, so the follower 1 will process this request.

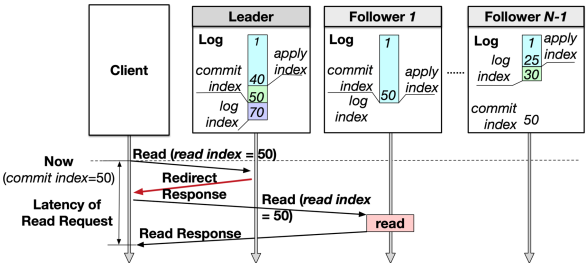


Fig. 8. vRaft's procedure of processing read requests.

3.2 Algorithm Design

Fastest Return. Algorithm 1 exhibits the key functions of FR, i.e., the newly added *sendNotify* for the followers and *handleNotify* for the leader. When the apply index of any follower has changed, this follower will send a *Notify RPC* (including its apply index) to the leader, shown as Lines 1–3 in Algorithm 1.

When the leader receives a *Notify RPC*, it updates the record of the corresponding follower' apply index, and calculates the maximum apply index of all the nodes in the Raft group, as shown in Lines 4–6. Through the variable *status* in Line 5, the leader records the status of all the nodes including their apply indexes, log indexes, etc.

Finally, the leader will check all the write requests that have not finished (i.e., the list *unresponsiveWrites* in Line 7). If the updated *maxApplyIndex* is newer than the index of a waiting write request, the write request is considered as an accomplished one and the leader can directly return success to the client, as shown in Lines 7–12. Note that the list *unresponsiveWrites* is a FIFO queue, so when the first request's write index is newer than the current max apply index, the other request will also have to wait.

Algorithm 1. Fastest Return

```
1: function Follower :: sendNotify() :
2:     message.set(leaderID, this.applyIndex);
3:     send(message);
4: function Leader :: handleNotify(message) :
5:     status.update(message);
6:     maxApplyIndex ← status.getMaxApplyIndex();
7:     while unresponsiveWrites.len() > 0 do
8:         writeIndex ← unresonsiveWrites[0].index;
9:         if maxApplyIndex >= writeIndex then
10:            write ← unresonsiveWrites.remove();
11:            respond(write);
12:        else break;
```

Optimal Read. The key functions of Optimal Read are described in Algorithm 2. All the incoming read requests are put into a list of the leader called *pendingReads*. The leader checks all the pending read requests when a new request enters the pending request queue, or when the leader's apply index changes, or when it receives the follower's *Notify RPC*. If the apply index of the leader is no less than the read index of the request, the leader can immediately execute the read operation, as shown in Lines 1–5.

Otherwise, if the apply indexes of some followers are newer or equal to the read index, we need to compare the cost of redirecting the request to one follower and the overhead of waiting for the leader finishing the apply

operations. Assume that the average time for a leader to apply a log is a, and the additional time of network transmission caused by request redirecting is b. If $(read\ index - leader's\ apply\ index) * a > b$, the redirecting plan is faster. Let $c = b/a$; when $the\ leader's\ apply\ index + c < read\ index$ and $all\ followers'\ maxApplyIndex >= read\ index$ are satisfied, we should redirect the read request to a follower to process, as shown in Lines 6–7.

Then, if there are multiple followers that satisfy the apply index condition, the leader will choose the follower with the minimal read load, as shown in Lines 8–9, where F is the set of the followers that satisfy the apply index condition. In this case, the leader will send a redirect response to the client (including the read index and the follower id) and update the follower's read load record, as shown in Lines 10–11. Note that each node records its own read load and the followers report their records to the leader periodically.

After that, the follower receives the redirected read request. If its apply index is greater than or equal to the read index of the request, the follower can directly execute the read request, as shown in Lines 12–15.

For the client, it first sends the read request to the leader and gets a response. If the response is a redirect message, the client sends the read request to the corresponding follower to get the target data, as shown in Lines 16–22.

Algorithm 2. Optimal Read

```
1: function Leader :: checkReads() :
2:     for each readReq ∈ pengdingReads do
3:         readIndex ← readReq.readIndex;
4:         if applyIndex >= readIndex then
5:             execute(readReq);
6:         else if applyIndex + c < readIndex then
7:             if maxApplyIndex >= readIndex then
8:                 F ← getFollowers(readIndex);
9:                 followerId ← F.minReadLoadNode();
10:                redirect(readReq, readIndex, followerId);
11:                status.update(followerId);
12: function Follower :: handleRedirectRead(readReq) :
13:     readIndex ← readReq.readIndex;
14:     if applyIndex >= readIndex then
15:         execute(readReq);
16: function Client :: Read(readReq) :
17:     response ← sendNrecv(readReq, leaderID);
18:     if response.type = Redirect then
19:         readIndex, followerId ← reponse.get();
20:         readReq.readIndex ← readIndex;
21:         readReq.type ← Redirect;
22:         response ← sendNrecv(readReq, followerId);
```

3.3 Linearizability of vRaft

Although vRaft changes both the write and the read procedures compared with Raft, it does not break the linearizability [16] guaranteed by Raft.

Theorem 1. *vRaft does not break the linearizability, which means once a new value has been written or read, all the subsequent reads see the new value, until it is overwritten again.*

Proof (sketch): Once writing a new value is finished, it triggers that the apply index is updated to a_1 and we assume the current commit index is c_1. Thus the commit index must be larger than or equal to the apply index (i.e., $c_1 \geq a_1$). The read index r of any subsequent read request will be equal to or larger than the current commit index c_1 (i.e., $r \geq c_1 \geq a_1$). Because vRaft also guarantees that only when the apply index is greater than or equal to the read index, the read request can be executed, the apply index (i.e., a_2) when serving a subsequent read request, which has the same or larger read index than r, needs to be equal to or greater than r, i.e., $a_2 \geq r \geq c_1 \geq a_1$. Therefore, the state machine of the version a_2 definitely contains the written value in the version a_1, which will make sure the newly written value can be read by all the subsequent read requests.

When a new value has been read, assuming the current *apply index* is a_1. A subsequent read request's *read index* r_2 is equal to the current *commit index* c_2, which is larger than a_1, i.e., $r_2 = c_2 \geq a1$. And the new read request with the read index r_2 will also be served by a node with the *apply index* a_2 which is larger or equal to r_2. So we can get $a_2 \geq r_2 = c_2 \geq a_1$. Similar to the above case, because of $a_2 \geq a_1$, we can make sure the subsequent read requests can get the new value.

4 Implementation and Evaluation

Raft has been widely implemented in the industrial community, such as famous open-source systems like Etcd [4] and TiKV [5]. Etcd is based on a memory-based state machine, adopting a single Raft group; it is designed to store a small amount of data such as metadata. Different from Etcd, TiKV adopts multiple Raft groups and the disk-based state machine for massive data. Therefore, we implemented our proposed vRaft and integrated it into TiKV for evaluations.

As a distributed key-value storage system based on Raft, TiKV utilizes RocksDB [17] as the key-value engine for the state machine on each node, and employs another system called Placement Driver (PD) [18] to manage the data distribution of TiKV. In fact, TiKV contains more than 100K LOC of *Rust*, which is already one of the largest open-source projects in the *Rust* community.

4.1 Experimental Setup

The experiments were performed in a cluster consisted of eight virtual nodes in Tencent Cloud; each virtual node is coupled with Linux Centos 7.4.1708, 8

GB DRAM, and a 200 GB Solid State Drive (SSD). Six of the virtual nodes serve as TiKV nodes, one as PD, and the last one runs the benchmark tool, i.e., go-YCSB [19].

Go-YCSB is a *Go* language version of the widely used YCSB benchmark [20] for evaluating key-value systems. In the experiments, the used workloads are listed in Table 2, including *Load* (insert-only), *Workload A* (50:50 read/update), *Workload B* (95:5 read/update), and *Workload C* (read-only). Each key-value pair contains a 16-B key and a 1-KB value, and each data block has 3 replicas in TiKV. The default thread number of the go-YCSB client is 200. Other configurations all adopt the default ones in the go-YCSB specification [21].

Table 2. The YCSB workloads used in evaluations.

Name	Description
Load	Insert only
Workload A	50:50 Read/Update
Workload B	95:5 Read/Update
Workload C	Read only

In the following experiments, we adopt the system throughput (i.e., operations per second or ops/sec), the average latency, and the 99th percentile latency as the performance metrics.

4.2 Overall Results

In the overall experiments, we first load 100-GB data into the TiKV cluster, and then perform the workloads A, B, and C of YCSB, respectively, accessing 100 GB of data respectively.

As Fig. 9 plots, vRaft achieves higher throughput than Raft in most cases, i.e., 80% higher for *Load*, 64.2% higher for *Workload A*, and 2.7% higher for *Workload B*. This is because more writes make the differences in the apply indexes of different nodes be greater, thereby vRaft gains more performance improvement compared with Raft. For the read-only workload (i.e., C), there is no difference for the apply indexes of the nodes, so vRaft achieves almost the same throughput as Raft in *Workload C*.

Figure 10 exhibits the results of the average latency and the 99th percentile latency. vRaft achieves lower average latency than Raft in most cases, i.e., 44.2% lower for *Load*, 38.1% lower for *Workload A*, and 1.7% lower for *Workload B*. And the reduction on the 99th percentile latency of vRaft is more significant compared with Raft, e.g., 86.3% lower for *Load*, 88.5% lower for *Workload A*, and 13.9% lower for *Workload B*. Because when the leader writes slowly, Raft has to wait for the requests to be written one by one on the leader node, while vRaft can return the read and write results to clients through the faster progress of

followers. In *Workload C*, vRaft's average latency and the 99th percentile latency are close to Raft's, since the apply indexes of all the nodes do not change at all due to no new writes coming.

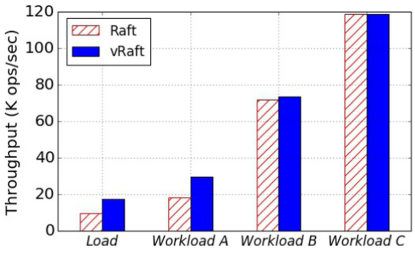


Fig. 9. Overall throughput results.

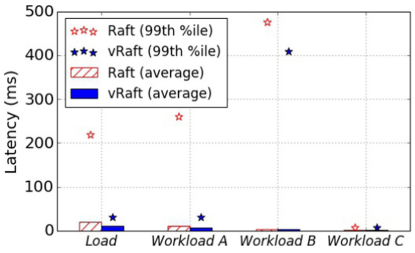


Fig. 10. Overall latency results.

4.3 Impacts of the Number of Replica

In this part, we measure the performance of vRaft and Raft under different numbers of replica configurations, including 3, 5, and 7 replicas. We adopt the read/write-balanced workload, i.e., *Workload A*, to perform the experiments. Specifically, we first load 10 GB of data into the cluster, and then perform *Workload A* of 10-GB data. The throughput and the latencies of performing *Workload A* are exhibited in Fig. 11 and Fig. 12.

As Fig. 11 plots, vRaft can achieve 46.2%–63.5% higher throughput compared with Raft under all different numbers of replica. In addition, Fig. 12 exhibits the results of the average latency and the 99th percentile latency. vRaft shortens the average latency by 30.4% to 38.9% and reduces the tail latency by 5.6% on average compared to Raft under all these cases. Because the loaded data is 10 GB, less than the amount of 100 GB in the overall results, the tail latency is not reduced as significantly as the above experiments.

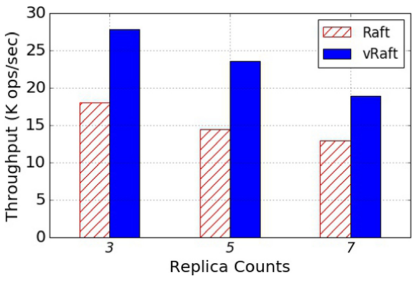


Fig. 11. Throughput for different replica counts.

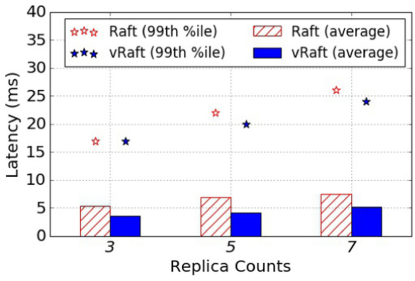


Fig. 12. Average and tail latencies for different number of replica.

4.4 Impacts of System Load

In this part, we measure vRaft and Raft under different system loads. The low, medium, and high system loads are configured by setting different numbers of client threads of *go-YCSB*. The thread number of low system load is only 10, the thread number of medium load is 50, and the number of high load is up to 200. In the experiments, we perform 10-GB *Workload A* based on an existing data set of 10 GB.

As Fig. 13 exhibits, no matter how much the system load is, vRaft can increase the system throughput significantly (i.e., 41.7% to 67.8% higher). Figure 14 indicates that vRaft can reduce the average latency by 30.4% to 46.1% and at the same lower the 99th percentile latency by 3.6% on average under all kinds of system loads. Of course, under the medium or the high system load, the advantage of vRaft can be fully exploited.

4.5 Scalability Evaluation

In order to evaluate the scalability of vRaft, we performed experiments on clusters with different counts of TiKV nodes (i.e., 3, 6, 12, 18, 24, or 30 TiKV nodes). All nodes here indicate virtual nodes. In the evaluation, we perform 10-GB *Workload A* based on an existing data set of 10 GB.

Figure 15 exhibits the results of the relative throughput, the relative average latency and the relative 99th percentile latency between vRaft and Raft under different system scales. As Fig. 15 shows, vRaft increases the throughput by 37.4% to 54.4% compared with Raft. vRaft reduces the average latency by 25.5% to 35.5% and reduces the tail latency by 9.6% on average compared with Raft. The results indicate that vRaft has good scalability and can improve the performance compared with Raft stably under different system scales.

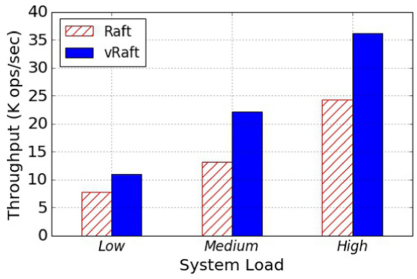


Fig. 13. Throughput under different system loads.

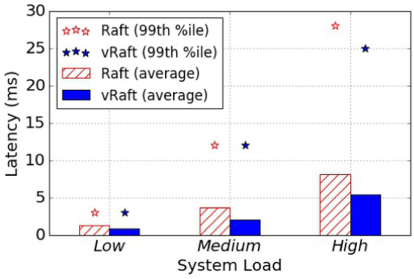


Fig. 14. Average and tail latencies under different system loads.

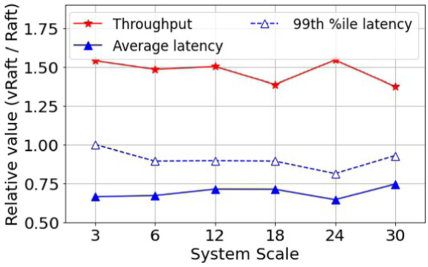


Fig. 15. Performance comparison under different system scales.

5 Related Work

Raft Optimization. Due to the importance of the Raft protocol for distributed systems, there are many existing works for optimizing Raft from different aspects. Some work optimizes the leader election mechanism of Raft, tuning the parameters about the election to make the election procedure faster [22,23]; some other work speeds up the leader election when some failures happen [24]. In addition, some researchers combine Raft with Software Defined Networking (SDN) [25–27].

As the number of nodes in the cluster increases, the throughput may decline because the only leader becomes the bottleneck of communication. In consequence, Sorensen et al. [28] proposed Obiden, a variation of the Raft protocol for distributed consensus with a president and a vice president to provide higher performance in large clusters. Besides, PolarFS [6] implements a parallel Raft to allow Raft's logs to be processed out of order, breaking Raft's strict limitation that logs have to be submitted in sequence, with the benefit of increasing concurrency.

Hanmer et al. [29] found that Raft may not work properly under overloaded network conditions. A request such as a heartbeat cannot be returned within a specified time, thereby being considered as a failure. The heartbeat failure may cause the leader to be converted to a follower, restarting the slow leader election procedure. Furthermore, the leader election may be repeated again and again under the poor network condition, delaying the request processing seriously. Therefore, they proposed DynRaft [29], a dynamic extension of Raft to optimize the performance when the network is overloaded.

Copeland et al. [30] proposed BFTRaft, a Byzantine fault tolerant variant of the Raft consensus protocol. BFTRaft maintains the safety, the fault tolerance, and the liveness properties of Raft in the presence of Byzantine faults, and keeps the modified protocol simple and readily comprehensible, just as Raft does.

Paxos Optimization. In order to reduce the high latency of the Paxos protocol, Wang et al. proposed APUS [31], the first RDMA-based Paxos protocol that aims to be fast and scalable to client connections and hosts. Ho et al. [32] proposed a Fast Paxos-based Consensus (FPC) algorithm which provides strong consistency.

FPC adopts a controller priority mechanism to guarantee that a proposal must be elected in each round and no additional round is needed, even more than two proposers get the same votes.

In summary, the existing optimization work about the Raft-based distributed consensus does not consider the performance heterogeneity and fluctuation problem of virtual nodes in the cloud environment. Our proposed vRaft solution is the first method to solve this new problem under virtualized environments.

6 Conclusion

In a virtualized cloud environment, the performance of each virtual nodes may be heterogeneous, and they often affected seriously by the behavior of other virtual nodes located on the same physical node, thus keeps fluctuating. Therefore, the Raft protocol, which has been widely used in many distributed systems to achieve consensus, will encounter new performance problems when the leader node is temporarily slow, because both the read and the write requests have to wait for the leader's processing to be finished in Raft, even if some follower nodes are obviously faster.

In order to break the too strict leader limitation in Raft and to fully utilize the fast follower to accelerating the request processing, we propose a new version of Raft for performance optimization in virtualized environments, called vRaft. vRaft contains two new mechanisms, i.e., Fastest Return and Optimal Read, to accomplish the processing of both the write and the read requests ahead of time compared with Raft, respectively, through involving fast followers in the processing. Besides, we have implemented our proposed vRaft in an industrial level distributed key-value systems (i.e., TiKV). And the experiments based on the Tencent Cloud platform indicate that vRaft can effectively and stably improve all the key performance metrics at the same time, including the throughput, the average latency, and the tail latency.

Acknowledgement. This work is supported by the National Key Research and Development Program of China (No. 2019YFE0198600), National Natural Science Foundation of China (No. 61972402, 61972275, and 61732014).

References

1. Lamport, L.: Paxos made simple. ACM SIGACT News **32**(4), 18–25 (2001)
2. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference (USENIXATC 2014), pp. 305–319 (2014)
3. Ongaro, D.: Consensus: bridging theory and practice. Stanford University (2014)
4. Etcd. https://github.com/etcd-io/etcd
5. TiKV. https://github.com/pingcap/tikv
6. Cao, W., Liu, Z., Wang, P., et al.: PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. Proc. VLDB Endow. **11**(12), 1849–1862 (2018)

7. Where can I get Raft? https://raft.github.io/#implementations
8. Kernel-based Virtual Machine. https://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine
9. Xen. https://en.wikipedia.org/wiki/Xen
10. docker. https://www.docker.com/
11. cgroups. https://en.wikipedia.org/wiki/Cgroups
12. Performance interference and noisy neighbors. https://en.wikipedia.org/wiki/Cloud_computing_issues#Performance_interference_and_noisy_neighbors
13. Misra, P.A., Borge, M.F., Goiri, Í., et al.: Managing tail latency in datacenter-scale file systems under production constraints. In: Proceedings of the Fourteenth EuroSys Conference, p. 17. ACM (2019)
14. Tencent Cloud. https://intl.cloud.tencent.com/
15. Flexible I/O Tester. https://github.com/axboe/fio
16. Kleppmann, M.: Designing Data-intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media Inc., Sebastopol (2017)
17. RocksDB. http://rocksdb.org/
18. PD. https://github.com/pingcap/pd
19. go-ycsb. https://github.com/pingcap/go-ycsb
20. Cooper, B.F., Silberstein, A., Tam, E., et al.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 143–154. ACM (2010)
21. go-ycsb workloads. https://github.com/pingcap/go-ycsb/tree/master/workloads
22. Howard, H., Schwarzkopf, M., Madhavapeddy, A., et al.: Raft refloated: do we have consensus? ACM SIGOPS Oper. Syst. Rev. **49**, 12–21 (2015)
23. Howard, H.: ARC: analysis of Raft consensus. Computer Laboratory, University of Cambridge (2014)
24. Fluri, C., Melnyk, D., Wattenhofer, R.: Improving raft when there are failures. In: 2018 Eighth Latin-American Symposium on Dependable Computing (LADC), pp. 167–170. IEEE (2018)
25. Sakic, E., Kellerer, W.: Response time and availability study of RAFT consensus in distributed SDN control plane. IEEE Trans. Netw. Serv. Manage. **15**(1), 304–318 (2017)
26. Zhang, Y., Ramadan, E., Mekky, H., et al.: When raft meets SDN: how to elect a leader and reach consensus in an unruly network. In: Proceedings of the First Asia-Pacific Workshop on Networking, pp. 1–7. ACM (2017)
27. Kim, T., Choi, S.G., Myung, J., et al.: Load balancing on distributed datastore in opendaylight SDN controller cluster. In: 2017 IEEE Conference on Network Softwarization (NetSoft), pp. 1–3. IEEE (2017)
28. Sorensen, J., Xiao, A., Allender, D.: Dual-leader master election for distributed systems (Obiden) (2018)
29. Hanmer, R., Jagadeesan, L., Mendiratta, V., et al.: Friend or foe: strong consistency vs. overload in high-availability distributed systems and SDN. In: 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 59–64. IEEE (2018)
30. Copeland, C., Zhong, H.: Tangaroa: a byzantine fault tolerant raft (2016)
31. Wang, C., Jiang, J., Chen, X., et al.: APUS: fast and scalable paxos on RDMA. In: Proceedings of the 2017 Symposium on Cloud Computing, pp. 94–107. ACM (2017)
32. Ho, C.C., Wang, K., Hsu, Y.H.: A fast consensus algorithm for multiple controllers in software-defined networks. In: 2016 18th International Conference on Advanced Communication Technology (ICACT), pp. 112–116. IEEE (2016)