Cost-effective Strong Consistency on Scalable Geo-diverse Data Replicas

Yunxiao Du † , Zichen Xu #, Kanqi Zhang #, Jie Liu #, Christopher Stewart *, Jiacheng Huang #
† Nanchang University/Zhejiang University, # Nanchang University, * Ohio State University

Abstract—The Raft algorithm maintains strong consistency across data replicas in Cloud. This algorithm places nodes, i.e., leader and follower, to serve read/write requests spanning geo-diverse sites. As the workload increases, Raft shall provide proportional scale-out performance. However, traditional scale-out techniques are bottlenecked in Raft with an exponentially increased performance penalty when provisioned sites exhaust local resources. To provide scalability in Raft, this paper presents a cost-effective mechanism that enables elastic auto scaling in Raft, called BW-Raft. BW-Raft extends the original Raft with the following abstractions: (1) *secretary* nodes that take over expensive log synchronization operations from the leader, relaxing the performance constraint on locks. (2) *observer* nodes that handle reads only, improving throughput for typical data intensive services. These abstractions are stateless, allowing elastic scale-out on unreliable yet cheap spot instances. In theory, we prove that BW-Raft can preserve the strong consistency guarantee from Raft at scale-out, handling 50X more nodes, compared to the original Raft. We have prototyped the BW-Raft on key-value services and evaluated it with many state-of-the-arts on Amazon EC2 and Alibaba Cloud. Our results show that within the same budget, BW-Raft incurs 5-7X less resource footprint increment than Multi-Raft. Using spot instances, BW-Raft can reduces costs by 84.5%, compared to Multi-Raft. In the real world experiments, BW-Raft improves goodput of the 95th-percentile SLO by 9X, thus, serves an alternative for distributed service scaling out with strong consistency.

Index Terms—Strong Consistency, Scalability, Geo-diverse, Spot Instance

1 Introduction

The Raft algorithm is invented to support strong consistency in networked services, serving as a simplified alternative for Paxos [26]. Distributed systems use Raft to keep consensus between software/system components. For example, components must agree on locking ownership and operations in synchronized queues. Raft is used in practice by many large-scale platforms, such as Google Kubernetes [11], Core OS [9], and Oracle [10]. These platforms, i.e., the clients of Raft, suffer inflated costs when Raft employs inefficient scaling techniques.

By design, Raft shall be easy to learn and implement. Software threads in Raft are either *leaders* or *followers*. At all times, Raft allows zero or one leader elected by followers. Followers periodically heartbeat the leader, checking for failures. Upon leader failure, followers call for a leader election to name a new leader. By design, the leader has a more demanding workload: it pushes writes to all available followers, tracks which followers have confirmed the most recent writes, and dispatches reads to these confirmed followers. The leader is the performance bottleneck of Raft.

When workload demands overwhelm the leader, the Raft infrastructure must acquire and use new resources to improve its throughput. One approach is to run the leader on more powerful computers (*scale up*). When these powerful computers are unavailable, the leader's load must be split (*scale out*). There raises one key issue: *Raft permits one leader only*. One leader is essential to make the Raft algorithm understandable, correct, and easy to implement. Multi-Raft [21] scales out by replicating leaders and followers, and splitting data between replicas. Each replica implements a Raft, providing strong consistency. Between replicas, a 2-phase commit provides strong consistency. Multi-Raft can improve throughput as workload demand increases, but it is undesirably expensive. Each scale out operation doubles resource footprint. Besides, these algorithms require complex procedures to handle partitions, fork threads, and remove threads. Subtle programming mistakes can invalidate strong consistency guarantee, leading to costly bugs.

With the global proliferation of networked devices, follower nodes in Raft are increasingly geo-distributed. Consider a global, coordinated release of some streaming content, Raft coordinates when such content is accessible. Placing followers in geo-distributed data centers ensures low access latency for all users. On the other hand, the cost and amount of cloud resources requested by followers vary from site to site. By naïvely replicating leaders and followers, the Multi-Raft approach inevitably scales out at expensive sites. Our research seeks a solution that selectively excludes expensive sites during scale out, without compromising throughput or latency.

In all, there are three main issues in designing a practical Raft algorithm for large-scale platforms, namely *strong consistency*, *cost efficiency* (i.e., avoid temporally expensive sites), *simplicity* (i.e., one leader scheme). To address these challenges, we propose Black-Water Raft, or *BW-Raft*, a Raft extension that scales out well with geo-distributed resources. For strong consistency at scale out, BW-Raft supports 5 types of software threads: follower, candidate, leader, secretary, and

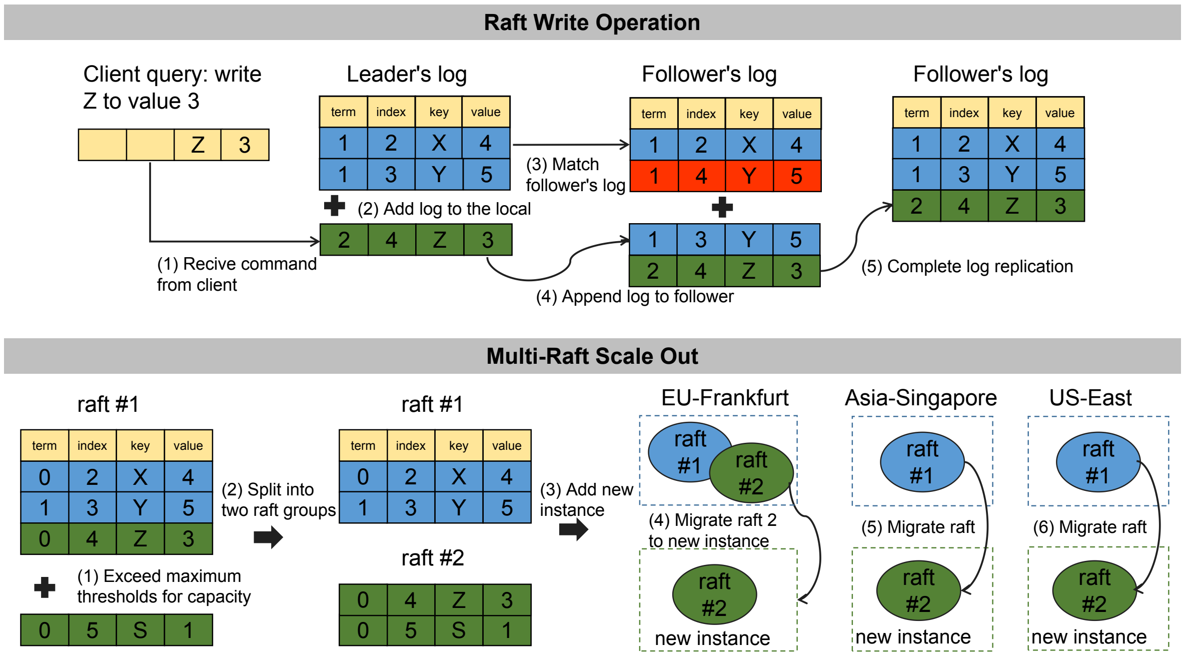


Fig. 1. Execution of writes in Raft (top) and geo-distributed scale out in Multi-Raft (bottom).

observer. Inherited from the original Raft, BW-Raft permits zero or one leader based on majority votes by followers (i.e., *simplicity*). In BW-Raft, leader outsources expensive log appending and replication to secretaries, reducing workload imbalance. Observers answer read-only queries, reducing workload for followers. BW-Raft allows multiple secretaries and observers to run simultaneously, on spot instances, thus ensuring the cost efficiency. BW-Raft preserves the safety guarantee of Raft. In theory, BW-Raft adopts *state irrelevancy* [24] to prove that secretary and observer failures do not affect correctness.

In BW-Raft, secretaries and observers are stateless and execute at any geo-distributed site. They can be deployed incrementally to achieve high throughput at a low cost as demanding workload grows. We present an online approach to discover global, high throughput, and low cost configurations for BW-Raft. This online algorithm allows BW-Raft to “peek and peak”, as rafting in a blackwater. BW-Raft always puts a default node configuration for the arrival workload (i.e., peek), then reconfigures it after a fixed period, for cost optimization (i.e., peak). In this optimization, our approach accepts time sensitive parameters on instance expense and workload latency. This allows BW-Raft to exploit cheap geo-distributed resources on spot markets, i.e., cloud markets with cheap but failure prone resources. For example, in Amazon AWS EC2, spot instances can cost 90% less than on-demand instances [30], [31], [42], [44]. Though with a cheaper price, introducing unstable spot instances into Raft infrastructure may hurt overall reliability [38]. To avoid fatal failures due to spot instances while harvesting cost benefits, BW-Raft uses safe, on-demand instances for leader and followers, and spot instances for temporary secretaries and observers only. In the geo-distributed setting, BW-Raft is cheap *per se*, as it leases spot instances from the cheapest market combined (i.e., cost effective). In an environment without spot instance, BW-Raft can also scale out with some on demand instances hired as secretaries and observers. But this will increase the cost of BW-Raft.

We deployed BW-Raft on Amazon EC2 and Alibaba Cloud for more than 3 months serving key-value lookups based on Google traces [6]. BW-Raft purchases low-cost spot instances in geo-distributed sites to scale out the performance within the budget. BW-Raft adaptively boosts read and write throughput which incurs less than 85% overhead compared to Raft [32]. BW-Raft scales in increments with a 5-7X smaller resource footprint than the Multi-Raft, the state of the art. BW-Raft reduces costs by 84.5%, as compared to the Multi-Raft, and improves goodput (i.e., the application-level throughput) of 95^{th} percentile SLO by 9X. BW-Raft operates key-value services for over 3 months without losing data or crash.
This paper contributes as follows:

- We identify the scale out performance problem of Raft, a widely used consistency algorithm, and how today’s solution fails at expensive costs.
- We propose BW-Raft, an extended Raft design that achieves cost efficiency by exploiting cheap but unreliable geo-diverse spot instances.
- We prove in theory that BW-Raft achieves strong consistency between nodes, and preserves the single-leader policy of Raft.
- We present an online approach to manage the BW-Raft infrastructure.
- We have built BW-Raft and deployed it for 2 months on public clouds, reporting that BW-Raft reduces costs by 84.5% and improves goodput by 9X, compared to state of the art mechanisms.

The remainder of this paper is organized as follows. Section 2 overviews Raft and the Cloud market. Section 3 describes BW-Raft and proves that it retains safety guarantees of Raft. Section

4 describes BW-Raft implementation over geo-distributed spot markets. Section 5 illustrates the BW-Raft can greatly reduce the cost. Section 6 provides a brief on consistency algorithms and related work. At last, Section 7 concludes the paper.

2 Background

2.1 A Consensus Algorithm

In the past decade, Paxos [26] was the algorithm to build distributed storage service. Due to the difficulty of understanding a Paxos implementation, it is not practical to build a complete Paxos system that is clean and easy to understand from a third party verification. Researchers promote Raft as it can ensure the linearizability as in Paxos and its notably simplicity and easy to verify [32]. Raft has attracted attentions from open source communities in data management. Raft is usually used to implement a key-value store for data intensive services, as follows:

- $\text{revision id} \leftarrow \text{write}(\text{key k}, \text{value v})$
- $\{\text{value v}, \ \text{revision id}\} \leftarrow \text{read}(\text{key k})$

Raft is designed to handle all queries by the leader node, which sets a global processing order for read/write queries and ensuring that subsequent queries could return the same value. In other words, Raft supports linearizable consistency.

In Raft, we assume all nodes can fail inevitably, e.g., hardware failures, some node's logs can diverge from other's. Raft uses a leader-follower structure to ensure that all nodes always agree on the leader. As Figure 1 (top) shows, there is a conflict between follower and leader log. The leader heartbeats the "victim" follower, keeping checking the log conflict and overwriting them until all logs are in consistent. Besides, Raft forces followers to accept log append from the leader. For example, some clients send a request to the leader. The leader saves the message locally and broadcasts it to all followers. The leader only responds to the client when the majority nodes have successfully replicated the log.

Raft maintains its consistency against failure through an election process. In Raft, software threads are either the leader, followers, or candidates. The leader handles all the requests from clients and maintains its role by sending heartbeat messages. After receiving a heartbeat, followers reset a random election time. If a follower has not received a heartbeat after the election time, the follower increments its token, announces its candidate role, and calls for a election. Other followers can only vote for a candidate whose log token is not larger their own. If a candidate gets the majority votes, it is elected to be the new leader.

With this design, the Raft algorithm has some spare room to improve. There are many interesting papers on optimizing Raft. For example, Craft [37] is proposed to to reduce network cost and storage space. Another C-Raft [14] defines a hierarchical model of consensus to improve upon throughput in globally distributed systems. In this paper, we target at the scalability of Raft with a focus on the cost efficiency.

Scaling out in Raft. Raft permits only one leader at a time and the leader has a demanding workload. Naïvely adding follower nodes does not improve throughput. Instead, it often degrades throughput by causing more log matching [43]. Multi-Raft is a widely used approach to scale out Raft. As shown in Figure 1 (bottom), Multi-Raft sets up multiple Raft services, splits the key space and assigns each split to one Raft. Multi-Raft then migrates each Raft service to its own resources. In geo-distributed cloud settings, each Raft service uses instances at each site. For example, Figure 1 shows migration to new instances at AWS EU-Frankfurt, Asia-Singapore, and US-East sites. Each Raft is independent and handles their own update, log commit, and leader elect tasks (e.g., Raft #1 and Raft #2 in Figure 1). Between leaders of these Rafts, they communicate and keep in consistency based on the 2-phase commit. The Multi-Raft approach preserves linearizability within key ranges and scales out. However, it is not cost efficient, nor proportional to the number of nodes. Each scale-out can double the resource footprint to resolve light bottlenecks at only one leader node [43].

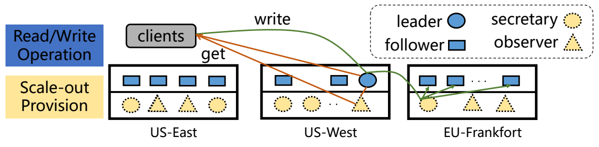


Fig. 2. BW-Raft scales out on spot market resources.

2.2 The Cloud Spot Market

In today's cloud market, spot instances [1] are short-lived instances offered by cloud providers for a very low cost compared to on-demand or reserved instances. Since customers' demand for cloud resources is dynamic, cloud providers use spot market to map extra resources for peak demand thus monetizing the cloud capacity. The price of spot instances vary with the supply and demand. On average, users can save up to 90% expenses, compared to on-demand instances. Although spot instance can save lot of money for users, the instance can be interrupted at any time when it has been outbid. With the growth of cloud services in recent years, more and more cloud providers have launched their own spot services to maximize resource utilization and revenue, such as Azure's Low-priority VM [2] and Google Cloud's Preemptible VM [5]. In the past, spot instances can only provide unstable services, which makes the use of the spot instance very limited. In our paper, we propose to use spot instances to extend the Raft protocol into BW-Raft that can use unstable instances with strong consistency.

3 Design

In this section, we describe the overall design of BW-Raft. As aforementioned in Section 1, BW-Raft handles the Raft protocol across a list of sites to provide data services with strong consistency. Such performance is obtained using our BW-Raft mechanism. BW-Raft operates original Raft among sites while hiring spot instances as secretaries and observers to offload jobs from the leader and followers, respectively. As such, we allow one leader to handle a larger number of followers, compared to the original Raft.

BW-Raft extends Raft with two new types of software threads: *secretary* and *observer*. Secretaries offload the heavy log appending and log checking jobs from the leader. Observers relieve heavy read demand pressure from followers. Secretaries and observers are stateless, allowing elastically scaling up and down. BW-Raft ensures linearizable consistency. Like Raft, BW-Raft allows only one leader thread. The leader is selected from followers as the original election. Tentative nodes, secretaries and observers, can not vote. As such, BW-Raft supports any number of secretaries and observers.

Secretaries and observers can run on cheap, failure-prone resources, making them well suited for geographically distributed spot markets, i.e., cloud resources that are heavily discounted but can be revoked at any time. Figure 2 provides an overview of our BW-Raft. Clients issue write queries to the leader. The leader offloads log matching and related tasks to a secretary which runs on a spot instance. Read queries are handled by either followers or observers. Note, the number of secretaries and observers varies from site to site and fluctuates. This gives our design the feasibility of chasing after cheap spot markets.

The rest of this section first details the algorithm on the leader election, secretary and observer failures, and the main safety guarantee. Then, we present a modeling approach for managing the global cloud resources effectively.

3.1 The BW-Raft Algorithm

BW-Raft, like the original Raft, initializes cloud instances at all sites and runs leader and follower software threads on them. Figure 3 shows the whole state transfer in BW-Raft, starting from leader election.

BW-Raft Leader Election: The leader manages log matching and replication for followers and secretaries. The execution trace of commands for the state machine is based on logging.

PROPERTY 3.1. **(Leader Election Safety).** *Like Raft, there is at most one leader per term in BW-Raft.*

The leader maintains its role by sending heartbeat messages. After receiving a heartbeat, followers set a random timer. If a follower does not receive a message before the timer triggers, the follower calls for an leader election (i.e., Step (1) in Figure 3). The follower increments its term and tells other followers that its a candidate for leader. Followers vote for a candidate whose log is not older than its own. If election times out, the election would restart again. If a candidate gets the majority votes from most followers, this secretary becomes the new leader and BW-Raft provisions secretaries for the new leader (i.e., Step (2) in Figure 3).

BW-Raft starts each term ($\bar{T}$) with a leader election. When a new leader is elected, it broadcasts an empty log to notify higher term. Meanwhile, the new leader in BW-Raft always tells followers which secretaries are responsible for this log, such that log management can be offload to assigned secretaries (i.e., Step (4) in Figure 3). If the election is fail, a new term starts with a new election.

PROPERTY 3.2. **(BW-Raft State Machine Safety)** *Each replicated copy of the state machine executes the same commands in the same order.*

Established in Woos and Wilcox's prior research in Raft [39], this property provides consistency guarantees for logs between all leader, secretaries, followers, and observers in the protocol.

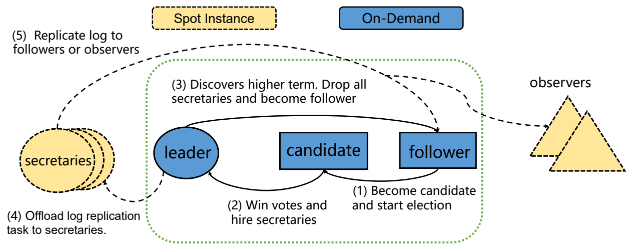

Spot Instance
On-Demand
(5) Replicate log to followers or observers
(3) Discovers higher term. Drop all secretaries and become follower
observers
secretaries
leader
candidate
follower
(4) Offload log replication task to secretaries.
(2) Win votes and hire secretaries
(1) Become candidate and start election

Fig. 3. The state machine and stateless nodes in BW-Raft. The central part in rectangle preserves the classic Raft state machine.

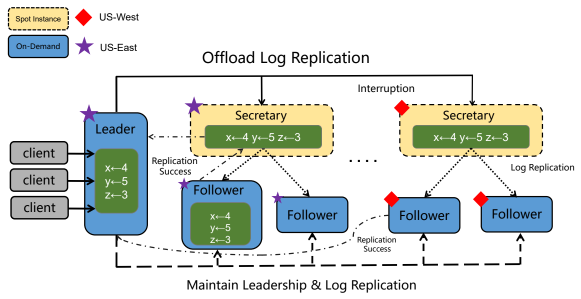


Fig. 4. Secretary offloads log replication from the leader.

Log Replication: BW-Raft targets at optimizing the performance in the log replication phase. In log replication, the list of commands to execute on the state machine is kept in log, and the position of a command in the log is called its index. Each node has its own copy of the log from leader, and state machine safety reduces to maintaining agreement between all copies of the log. Figure 4 shows an example of BW-Raft. When a client sends a write to the leader, the leader first appends a new log entry containing that command to its local log. Then the leader sends an appended message containing the entry to an assigned secretary, which is responsible to replicate the log to the followers in the same site. Each follower in the same site appends the entries to its log and sends the acknowledgment to the assigned secretary. To ensure that followers' logs stay in consistent with the log of the leader, secretary's messages include the index and term of the previous entry in its local log; the follower checks whether it has an entry with the index and term before appending new entries to its log. This consistency check guarantees the following property:

PROPERTY 3.3. **(Log Matching).** *If any two logs contain same index and term, then the logs are identical in BW-Raft.*

In BW-Raft, all logs are replicated from leader to follower, or from leader to secretary to follower. After the secretary learns the majority followers have acknowledged the new entry, it notifies the leader the number of follower and agreed index. Once the leader learns the majority followers have acknowledged the new entry, it executes the command contained in the committed entry in the state machine and responds to the client with the output. The followers and observers can safely execute the command on their state machines after receiving the heartbeat with the committed index. Thus, in both leader election and log replication, both secretaries and observers are state irrelevant.

PROPERTY 3.4. **(State Irrelevancy).** *If any secretaries/observers fail during the term at different locations, then the logs are still matched because state machines in both leader and followers are irrelevant to secretaries and observers.*

We first prove the State Irrelevancy in secretaries. Given an execution trace τ of BW-Raft, assuming one secretary fails during the trace, we shall find an existing σ such that τ linearizes to σ. To find the σ, we revert BW-Raft back to a single Raft, then we pick the sequence of commands executed by the followers on their local state machines. State machine safety guarantees that all nodes agree on this sequence.

The remaining job is to show that without failing certain secretaries, we have τ' linearizes to σ. Let τ' be the sequential input–output trace corresponding to σ in τ. That is, for each command executed in σ, τ' contains an input immediately followed by the corresponding output for that command. All commands from/to secretaries can be omitted in τ', which makes it a permutation of τ that respects the ordering condition of properties. Each of these is established as a separate invariant by induction on the execution. The proof for observers are similar since they never affect the order of local state machine of followers.

With properties inherited from Raft, while the additional roles in BW-Raft are state-irrelevant, we have,

PROPERTY 3.5. **(linearizability)** *Properties 3.1–3.4 imply linearizability in BW-Raft.*

The proof trivially follows the linearizability proof in Raft [39]. Once an entry is committed, it becomes durable, BW-Raft operates log replication with linearizability.

PROPERTY 3.6. **(Liveness).** *BW-Raft supports changes in cluster membership.*

BW-Raft also provides a liveness guarantee: if there are sufficiently few failures, then the system will eventually process and respond to all client commands.

3.2 Global Resource Management

BW-Raft offloads some operations to secretaries and observers, by exploiting cheap but revocable resources, such as spot instances, for cost efficiency. This section answers a critical infrastructure question, for a given data service, how many and which sites the required instances are purchased.

In our design, BW-Raft manages its global resource via "peek and peak". Whenever a new data service arrives, BW-Raft use "peek and peak" mechanism to predict a configuration for it. For example, BW-Raft can run under the original Raft model at the beginning. However, in practice, to keep the handler open on accessing spot instances, we set at least one secretaries and observers when Bw-Raft is initialized, allowing the BW-Raft implementation to keep the connection without rebooting the instance create API. Once the target service load increases beyond the original Raft's capacity, BW-Raft starts to scale, predict, and assign new resources, as rafting in black water, or "*peek*". On the other hand, if the target service scales out configuration has been determined, we would like to pick at least k nodes to meet the demand. This decision recurs every epoch, which allows BW-Raft to select best cost-efficient instances online, or "*peak*" efficiently.

Peek in BW-Raft. Given a newly arrived service, BW-Raft runs for a fixed period of time T, e.g., 1 hour. The value of T should be set according to the Service Level Agreement (SLA) require-

Symbol	Description
ρ	The unit price of spot instance
β	The unit price of on-demand instance
ϑ	The available budget
k_s	The number of secretaries in cluster
k_o	The number of observers in cluster
Δk_s	The number of new secretaries
Δk_o	The number of new observers
k	The number of spot instances that need to be rented
N_r	The number of read requests in last period
N_r'	The number of read requests in current period
$\mathcal{A}$	The growth rate of the number of read requests
ϖ	The write ratio threshold
ζ	The write ratio in current period
m	The number of data centers
F_i	The number of followers in i_{th} data center
k_{oi}	The number of observers in i_{th} data center
f	The number of followers one secretary can handle
$\mathbb{C}$	A linear function of network cost

Algorithm 1 The Algorithm of the Top-K sites Decision

```
Input: The parameters mentioned in Table 3.1
Output: $k$: the number of new spot instance, including secretary and observer
1: initial $k_s = 0$, $k_o = 0$, $\varpi$=30%;
2: for every period time $T$ do
3:     $k_s' \leftarrow \sum_{i=1}^{m} \frac{F_i + k_{oi} + \frac{f+1}{2}}{f}$
4:     $\Delta k_s \leftarrow k_s' - k_s$
5:     if $\zeta \leq \varpi$ then
6:         $\mathcal{A} \leftarrow \frac{N_r' - N_r}{N_r}$
7:         if $\mathcal{A}$>10% then
8:             $\Delta k_o \leftarrow m$
9:             $\Delta k_o \leftarrow min(\Delta k_o, \frac{min(\rho\Delta k_o, \vartheta)}{\rho})$
10:        else if $\mathcal{A} < -10\%$ then
11:            $\Delta k_o \leftarrow max(-k_o, -m)$
12:        end if
13:        $\vartheta \leftarrow max(0, \vartheta - \rho\Delta k_o)$
14:        $\Delta k_s \leftarrow min(\Delta k_s, \frac{\vartheta}{\rho})$
15:        $\vartheta \leftarrow max(0, \vartheta - \rho\Delta k_s)$
16:    else
17:        $\Delta k_s \leftarrow min(\Delta k_s, \frac{\vartheta}{\rho})$
18:        $\vartheta \leftarrow max(0, \vartheta - \rho\Delta k_s)$
19:        $\Delta k_o \leftarrow min(m, \frac{\vartheta}{\rho})$
20:        $\vartheta \leftarrow max(0, \vartheta - \rho\Delta k_o)$
21:    end if
22:    $k_s \leftarrow k_s + \Delta k_s$
23:    $k_o \leftarrow k_o + \Delta k_o$
24:    $k \leftarrow \Delta k_o + \Delta k_s$
25: end for
```

Algorithm 2 Select Top-K Spot Instances

Input: $score$: queue of spot instances's score; $discard$: array of discarded instance in last T period; k: the number of spot instances selected;

Output: $topk[]$: array of selected spot instances's index

```
1: initial $L = 1$ and $R = len(score)$;
2: function TOP-K($k, L, R, score$)
3:     if $k > 1$ then
4:         $m \leftarrow binornd(R - L + 1, \frac{1}{2})$;
5:         $l \leftarrow \lfloor \frac{k}{2} \rfloor$;
6:         TOP-K($l, L, L + m - 1, score$);
7:         $l_{th} \leftarrow$ the $l$ smallest element in first $m$ samples;
8:         for $i = L + m \rightarrow R$ do
9:             $val \leftarrow score[i]$;
10:            if This instance was discard in last $T$ period then
11:                $discard.append(i)$
12:            end if
13:            if $val > l_{th}$ & $topk.size() < k$ then
14:                $topk.append(i)$;
15:            end if
16:        end for
17:    else
18:        $len \leftarrow R - L + 1$;
19:        $p \leftarrow floor(\frac{len}{exp(1)})$;
20:        $mx \leftarrow score[L]$;
21:        for $i = L \rightarrow L + p - 1$ do
22:            $val \leftarrow score[i]$;
23:            if $val > mx$ then
24:                $mx \leftarrow val$;
25:            end if
26:        end for
27:        for $i = L + p \rightarrow R$ do
28:            $val \leftarrow score[i]$;
29:            if $val > mx$ then
30:                $topk.append(i)$;
31:                return
32:            end if
33:        end for
34:        $topk.append(L)$;
35:    end if
36: end function
37: if $topk.size() < k$ then
38:    for $i \in discard$ do
39:        if $topk.size() < k$ then
40:            $topk.append(i)$
41:        end if
42:    end for
43: end if
```

ment. During this period, BW-Raft collects request statistics and the price of instances in the past T time. Based on this information, BW-Raft determines the number of spot instances needed for expansion by algorithm 1. Since the resources of the spot instance are limited, BW-Raft needs to prioritize the number of secretaries and observers based on the read-write ratio of workload. For example, if the write ratio ζ in current period time is less than ϖ=30%, BW-Raft gives the priority to the number of observers (line 5-15). Otherwise, the number of secretaries is given priority (line 16-20). ϖ is an user-defined variable in practice. When prioritizing observer, BW-Raft first computes the growth rate of read requests $\mathcal{A}$ (line 6). Because observers are only used to process read requests, the number of required observers is related to the amount of read requests. Taking system fluctuations into account, we argue that it is unnecessary to rent new observers or withdraw old observers when $|\mathcal{A}| \leq 10\%$, for the sake of avoiding extra overhead. When $\mathcal{A}$>10%, BW-Raft rents new observers. However, subject to the constraint of budget ϑ and unit price ρ, in m data centers BW-Raft can hire at most Δk_o new observers (line 8-9). When $\mathcal{A}< -10\%$, BW-Raft can cut down at most m observers (line 11) to rent secretaries or directly update available budget (line 13-15).

When the read burst varies frequently, BW-Raft can handle it within a bounded overhead. Reads start to burst yet BW-Raft does not have sufficient nodes to handle it. BW-Raft will start to hire more nodes as observers, to compute reads, in the next epoch t+1. In this way, if such burst continues in epoch t+1, the overhead from insufficient scheduled resources is limited up to one period duration, i.e., epoch length T. If such burst drops in epoch t+1, BW-Raft may make a bad decision on hiring nodes, thus suffers from less cost savings. Note that, the spot instance is much cheaper than on-demand instances. Thus, this would not inflate too much cost and we allow to pay this price, in order to make a simple design against unpredictable workload dynamics in practice instead of making a complex yet inefficient workload prediction. Now the key design issue is how to find a proper T such that we do not make decision too often in BW-Raft yet a bad decision would not cost us too much. When prioritizing secretary, we assume one secretary can manage f followers, then BW-Raft can get the total number of secretaries needed to offload leader's workload in current cluster (k'_s, line 3) and the number of new secretaries (Δk_s, line 4). Just like hiring observers, BW-Raft hires new secretaries with the rest budget and calculate available budget (line 17-20). After Algorithm 1 calculates the total number of new secretaries and new observers in cluster (i.e., k_s and k_o), The estimated total expense can be calculated by equation (1):

$$cost = \sum_{i=1}^{m} \{\beta F_i\} + \beta + \rho(k_s + k_o) + \mathbb{C} \tag{1}$$

where $\mathbb{C}$ is a linear function of network cost related to the total number of instances in cluster.

In extreme cases, In some extreme cases, the number of read and write requests in multiple consecutive periods could be small. BW-Raft may roll back up to k_s secretaries and k_o observers to save cost. This recall process takes m observers and $\frac{m}{f}$ secretaries in each period to avoid drastic fluctuations from the resource estimation in BW-Raft. On the other hand, when dealing with heavy read and write workloads, we have two cases. (1) When the resource budget is sufficient, BW-Raft can rent as many spot instances as requested to mitigate the load peak. (2) In the case of insufficient budget, BW-Raft prioritizes all budgets for renting spot instances as secretaries. Therefore, BW-Raft rents $\frac{\vartheta}{\rho}$ secretaries and rest instances to observers. In this way, both Algorithm 1 and Algorithm 2 can always adapt to these edge cases.

Peak in BW-Raft. Spot instance is quantified by the following estimation function 2, based on CPU capacity c, available memory ϕ, average price ϱ and revocation probability ξ. The revocation probability ℓ_i can be calculated by *RevPred* model [27].

$$score = \frac{\ell_1 c + \ell_2 \phi + \ell_3 \frac{1}{\varrho}}{\xi} \tag{2}$$

When BW-Raft needs k new spot instances, we use Multiple-Choice Secretary Algorithm (MCSA) [25] to select Top-K spot instances. Since the spot instances can be revoked at any time, any static offline algorithm could fail. We choose MCSA for two reasons, (1) we need to find a set of instances that provides the best revenue within a bounded time; (2) the algorithm shall be sufficiently simple and effective. The Multi-Choice Secretary Algorithm (MCSA) outstands other candidates because (1) the selection algorithm follows a linear behavior; (2) the near-optimal algorithm performs within a reasonable performance upper bound. MCSA can choose k elements and maximize their return within a bounded time at a competitive ratio of $1 - O(\sqrt{1/k})$.

In our implementation the Algorithm 2 is designed recursively under the following rules: If $k = 1$, find the maximum number mx of previous $\lfloor n/e \rfloor$ elements (line 18-26), then iterate over the remaining elements and select the next element which exceeds mx if such an element appears (line 27-34). If $k > 1$, search space is divided into two groups. Choose up to $\lfloor k/2 \rfloor$ elements among the first $binornd(n, 1/2)$ samples (line 4-6), then, pick elements from the left $n - m$ elements which larger than the $\lfloor k/2 \rfloor$ smallest element in prior $binornd(n, 1/2)$ samples (line 7-16). Since the instance pool changes dynamically in BW-Raft, new instances can add or discard at any time. When encountering a new instance, BW-Raft can get its priority score by equation (2). Considering additional cost caused by frequently discarding and renting the same instance, if a instance is deprecate in last T period, we do not consider it temporarily (line 10-11) unless there is no adequate instances to meet our demand (line 37-43). When BW-Raft discards instances in past T period while running Algorithm 2 (line 2-36), it can choose some new instances from them when needed (line 37-43). According to Algorithm 2, BW-Raft can get the best top k instances and maximize their scores. The time complexity of algorithm 2 can be evaluated as follow:

$$T = O(\frac{n}{2^0} + \frac{n}{2^1} + \frac{n}{2^2} + ... + \frac{n}{2^{logk}}) \tag{3}$$

$$= O(n(2 - \frac{1}{2^{logk}})) \tag{4}$$

Note that due to the randomness of binomial distribution, time complexity of our algorithm is difficult to specifically figure out. However, it is clear that the worst time complexity is $O(n)+m$, where m is the sum of $binornd(n, 1/2)$ in recursion.

Gradually, BW-Raft calculates k based on the statistical information of the previous T time (peek), and then selects the best k spot instances to allocate new secretaries and observers (peak). In practice, T is changeable, if T is set to a fine-grained manner, BW-Raft can approach to a real time system.

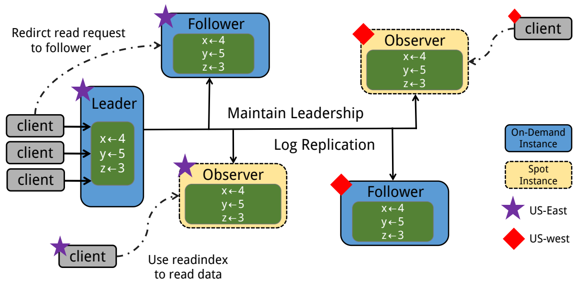


Fig. 5. Observer offloads read request from the follower.

4 Implementation

We have implemented a prototype of BW-Raft as an extension from the original Raft protocols. The whole implementation covers the two roles of nodes, i.e., secretary and observer. The BW-Raft prototype supports basic key-value store operation. To adapt the mordern cloud services, we use LevelDB [8] as the storage engine and GRPC [7] for communications between nodes.

```
Roles:  Follower|Leader|Secretary|Observer
Message Entry: {Index, Key, Value}
// Leader|Follower|Candidate
service BW-RAFT {
    rpc RequestVote(RequestVoteArgs)
        returns (RequestVoteReply){};
    rpc AppendEntries(AppendEntriesArgs)
        returns (AppendEntriesReply){};
    rpc GetReadindex(ReadIndexArgs)
        returns (ReadIndexReply){};
}
// Secretary
service BW-Secretary {
    rpc L2SAppendEntries(AppendEntriesArgs)
        returns (L2SAppendEntriesReply){};
}
// Observer
service BW-Observer{
    rpc AppendEntries(AppendEntriesArgs)
        returns (AppendEntriesReply){};
}
// Client|Server
service BW-KV {
    rpc PutAppend(PutAppendArgs)
        returns (PutAppendReply){};
    rpc Get(GetArgs)
        returns (GetReply){};
}
```

Listing 1. Remote Procedure Call in BW-Raft.

4.1 Original Raft Implementation

When all spot instances are not available or too expensive to hire, BW-Raft gradually shrinks to a original Raft design, with nodes as leader, follower, and candidate. As shown in Listing 1, we have defined the BW-Raft service to provide functions, such as, RequestVote and AppendEntries. When a follower has not received the heartbeat for a certain period of time, it increases its term and turns into a candidate. Once a candidate is created, the candidate starts a new election and uses RequestVote RPC to collect votes. If the term and index of the candidate's last log are at least up-to-date as logs from other followers, the follower votes for the candidate. Otherwise, the candidate's voting request gets rejected. After a round of election, if a candidate collects more the majority votes (more than 50%), this candidate becomes the new leader. Otherwise, a new round of election starts. When a new leader is elected, the leader starts to synchronize the logs with each node and sends heartbeat to prevent new election.

4.2 Secretary Implementation

Since the secretary is used to offload the leader's log replication task. As shown in Listing 1, we have defined the BW-Secretary service to synchronize the leader and secretary logs. When a new leader is elected, the leader runs Algorithm 1 to determine how many secretaries to hire according to the current workload. Then the spot instance used to run these secretaries is selected by Algorithm 2 to get lower cost and higher throughput. When a leader hires a secretary, the leader specifics some followers to secretary and lets the secretary replicate log to the specified followers. Figure 4 shows an example of BW-Raft how to hire a local spot instance as secretary to offload leader's log replication task. When the leader sends a new log to a secretary, the secretary replicates the log to the followers. After the log has been appended to the majority of followers or after the set time has passed, the secretary sends the number of followers who successfully replicated the log and the agreed index to the leader. As spot instance can be interrupted at any time, all the heartbeat messages are sent by leader to maintain leadership. BW-Raft can also hire global secretary to offload the communication task of confirming the leader status. Global secretary is used to further reduce the communication pressure of leader when there are too many nodes in the decision space. When there are multiple secretaries and one secretary fails, the leader can assign tasks of failed secretary to other secretaries. If the leader cannot communicate with the global secretary, the leader directly sends a heartbeat to confirm the status.

4.3 Observer Implementation

In order to deal with a large number of hot data reads, we implement an interface to call Algorithm 1 and Algorithm 2 to make BW-Raft dynamically rent spot instances as observer to offload client's read operation. Figure 5 shows an example of BW-Raft how to hire a local spot instance as observer to offload reads from followers. In BW-Raft, if a observer receives a read request, it uses GetReadindex to get the newest *readindex* from the leader, and return to the client after the state machine executes the readindex. However, with no additional measures, readindex operations would run at the risk of returning staled data, since the responding message might have been superseded by a newer leader. In original Raft, before the leader responses to the client's read request, the leader heartbeats to check the leadership. In BW-Raft, we hire a global secretary to do this. Therefore, as long as the leader communicates with global secretary , the replacement status is obtained.

We have published one version of BW-Raft in practice, which has been included in original Raft project *https://raft.github.io*. Those who are interested may refer *https://github.com/eraft-io/eraft* for more information.

5 Evaluation

BW-Raft is designed to achieve a cheap and scale-out consistency protocol by using on-demand instances and spot instances, which can handle different workflows and run on some unstable machines. In this section, we evaluate the performance of BW-Raft by prototyping it onto our physical testbed and Amazon AWS EC2 [1], [29].

System Setup: We deploy BW-Raft onto Amazon AWS EC2's instances (t2.small) to evaluate the runtime performance of BW-Raft by using Google workload. For the burstable spot instance, we report 0.415$/H on average. Spot instances are up to 90% cheaper than corresponding regular on-demand instances. We also build a physical testbed to illustrate the performance of BW-Raft by using YCSB [15] workload. The testbed contains 12 servers with Intel Xeon 2603 1.70GHz and 32GB DDR4 memory, 2TB HHD and connected by a switch router Ruijie RG-S2952G.

Software Setup: The operating system is built as Ubuntu16.04 (kernel version 4.15.0) and the version of YCSB is 0.17.0. Our experiments are built based on a client-server model, as shown in Figure 2. The client sends batched workload as reads/writes operation, with different CPU/memory demands, arrived in a Poisson distribution.

In this paper, we mainly evaluate BW-Raft with the following baselines:

- *Original* implements a state-of-the-art Raft design from Ongaro et al. [32];
- *Multi-Raft* is a state-of-the-art multi-raft implementation using sharding [20];
- *Oracle* is a theoretical best baseline based on offline analysis.

Workloads and Traces: We verify the performance of BW-Raft using real world workloads and traces. We use the popular Google cluster trace [6] which contains one-month job statistics in Google cluster. Workloads are random reads/writes, controlled ratio R/W batches, and read/write-only workloads. All workloads are tested with small, medium, and large block size, namely 256KB, 1024KB, and 2048KB, respectively.

- **Read** is a batch of workload with read-only queries.
- **Write** is a batch of workload with all writes.
- **α-Static** is an α controlled workload batch with α as the read/write ratio.

Performance Snapshots: First, we report the performance snapshot of running *BW-Raft*, *Multi-Raft*, and *Original*. The whole experiment lasts for 1200 epochs (i.e., 50 days) in Amazon EC2. Figure 6 plots the average latency when executing **Read** (top) and **Write** (bottom). For reads, BW-Raft provides the shortest average response time (i.e., 1.26s), which is 27% of Multi-Raft (4.6s) and 15% of Original (7.9s). However, in some

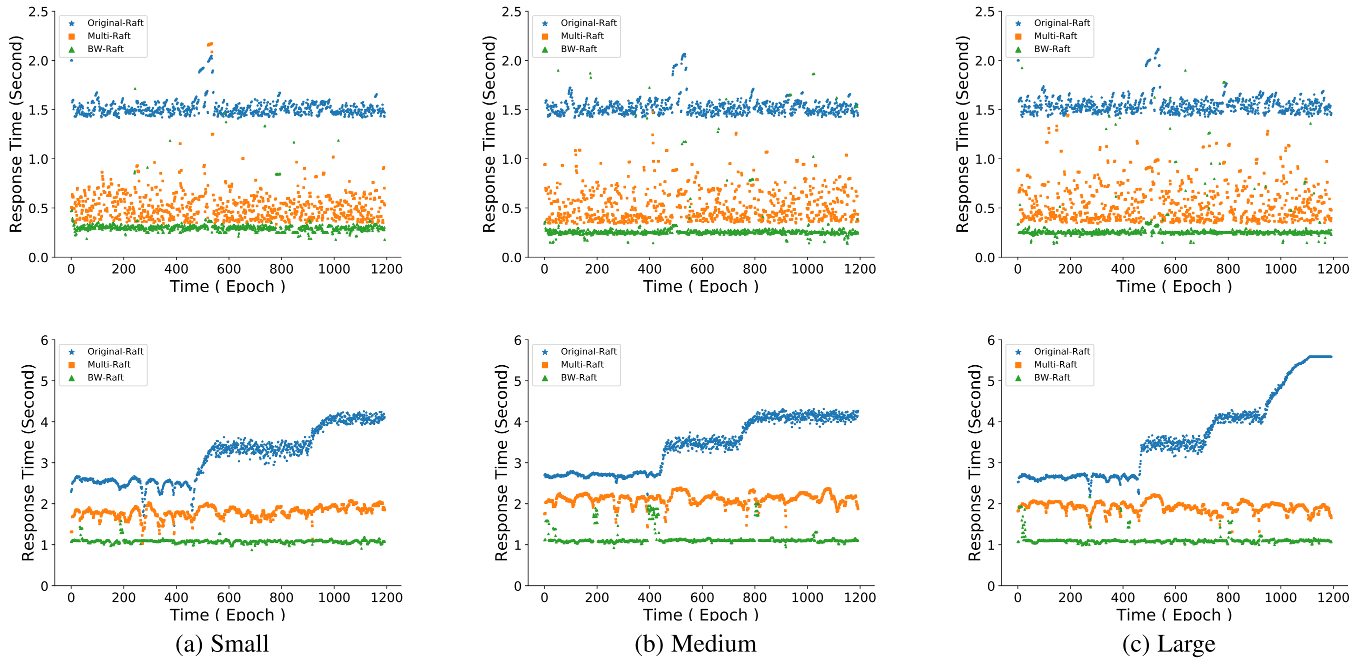


Fig. 6. 50 days Performance snapshots running **Read** (top) and **Write** (bottom) workloads. Y axis is in log scale.

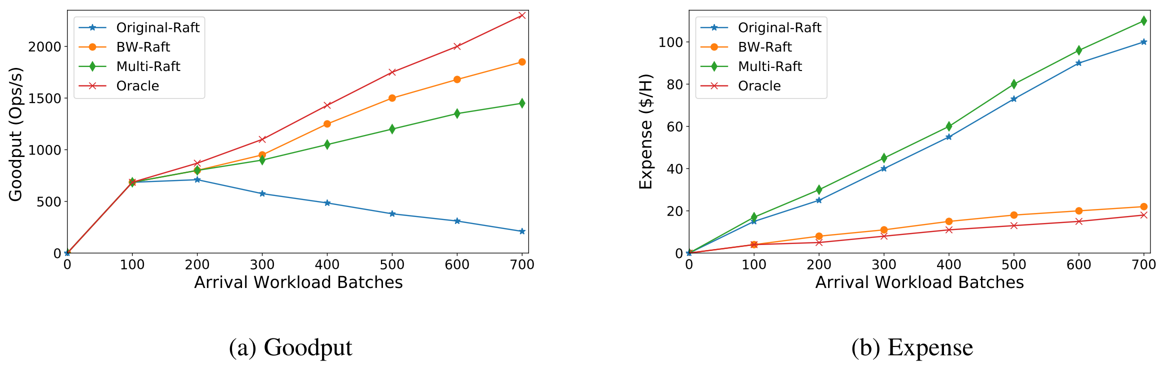
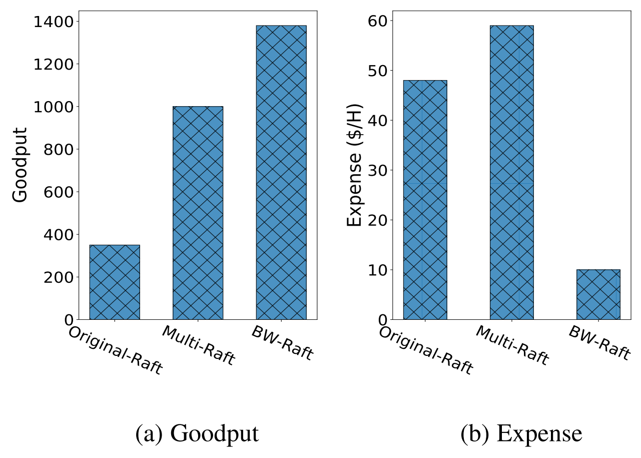


Fig. 7. Performance (a) and cost (b) of BW-Raft and other baselines at scale-out. Each batch contains 10k random read and write operations and the footprint of each operation is 4kB.

extreme cases, BW-Raft can perform badly, such as overshoots in epoch 410 and 578 in Figure 6(a) top. Similar overshoot happens when BW-Raft executes larger reads. Such overshoots happen when the majority of *observers* failed and BW-Raft had to reschedule workloads to correspondent *followers*. The overshoot can be mitigated when we spreading observers onto many sites instead of a few cheap ones, which is a tradeoff between performance and cost.

For the write operation, BW-Raft scales out in proportion to the ever-increasing updates, nicely. The secretary in BW-Raft runs on the spot instance. Sometimes, the secretary may fail or lack sufficient spot instances to run. In this case, the write response delay could be large because of the chain effect of write operation. In the next period, BW-Raft reallocates secretary resources, which makes the response time fairly stable. Multi-Raft also scales, however, with a price of 3X larger response time due to maintaining the 2-PC communication between leaders. Original cannot handle constant updates when scaling-out. When the size of appended logs increases to a limit in all nodes, Original fails at the leader blocking the overall write performance, slowing down 2.5X, compared to BW-Raft. Overall, BW-Raft shows significant performance improvement, as it scales in increments 3-12X compared to Multi-Raft and Original.

Scalability: BW-Raft can easily scale in proportional to performance and cost. Figure 7 demonstrates such proportionality when the workload batch size increases by 700X. In Figure 7(a), both BW-Raft and Multi-Raft grow in proportional to the workload size. BW-Raft exhibits a better scale-out performance than Multi-Raft, and is close to the theoretically best performance in Oracle. Original does not scale. When the workload increases, Original suffers from managing too many logs at the leader



(a) Goodput (b) Expense

Fig. 8. Performance (a) and cost (b) comparison between BW-Raft and other baselines.

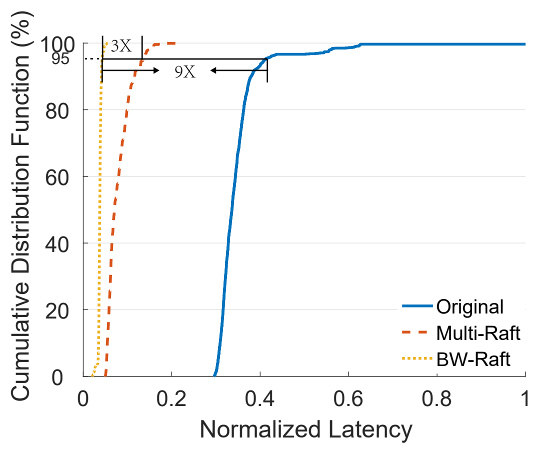


Fig. 9. CDF of BW-Raft and baselines.

node, which prevents scale-out. Multi-Raft can scale out at a much higher cost. As shown in Figure 7(b), Multi-Raft leases more on-demand nodes in scaling out while BW-Raft harnesses the salient feature of cheap spot instances. Compared to Oracle, BW-Raft still has rooms to improve, especially in the workload and resource provisioning.

Overall Statistics: Figure 8 shows the overall performance and expense comparison between BW-Raft and other baselines. In goodput (i.e., correct queried result over unit time), BW-Raft is 7X and 1.5X, larger than Original and Multi-Raft, respectively. Considering both reads and writes, BW-Raft has smaller variation than Multi-Raft. BW-Raft has significantly smoothed write delay curve due to secretaries often reside in more sites than *followers*, which reduces unexpected long wide-area network delay. For expenses, BW-Raft exploits cheap spot instances for secretaries and observers in many sites. BW-Raft spends 86% and 80% less than Multi-Raft and Original, respectively. Multi-Raft usually costs more than Original due to its multiple leaders thus expensive resource footprints. Note that, it seems unfair to show this expense comparison while only BW-Raft can use spot instances. However, there is no existing design on both Raft and Multi-Raft that can exploit spot instances. If we deploy spot instance in Original and Multi-Raft, they can be drown in a nonstop leader re-election and provide almost zero performance.

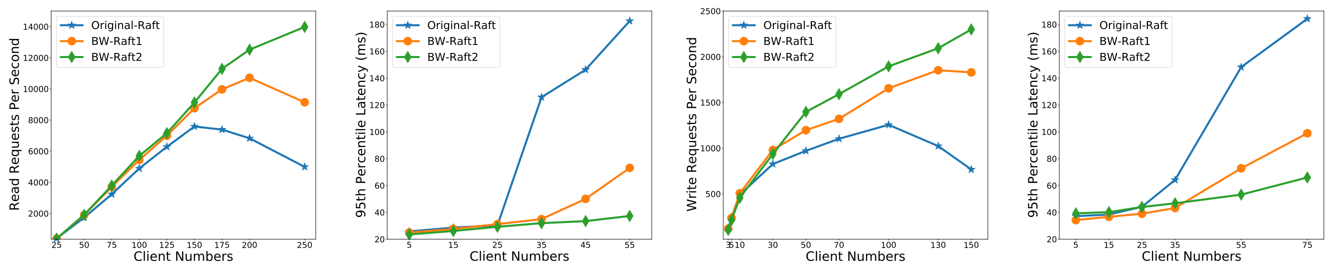
Performance Distribution. Figure 9 demonstrates the curriculum distribution functions of all jobs running in BW-Raft and other baselines. A small portion of jobs do suffer in BW-Raft due to runtime spot instance failures and errors in resource provisioning. In the worst case, BW-Raft could shrink back as a Raft handling a small number of jobs. However, BW-Raft has a much shorter tail than Multi-Raft in the latency distribution. Comparing the 95th-percentile SLO, BW-Raft performs 3X better than Multi-Raft, and 9X better than Original.

BW-Raft's Performance with Different Configuration: Because BW-Raft extends the role of secretary and observer to offload the read and write requests, we evaluate the performance of Bw-raft under different numbers of secretary and observer. In Figure 11(a), different YCSB benchmark tests show that BW-Raft's average throughput is 1.5-2 higher than original Raft. However, as Figure 10(a)(b) shows, the improvement of reading and writing performance is not significant when the request batch is small. Besides, adding secretary can lead to some performance degradations as shown in Figure 10(d), due to the communication latency between the leader and the secretary. With the increasing number of instances, more and more network connections and data pile up, the CPU footprints of Raft leader soon be exhausted, as shown in the Figure 11(c).The response delay rockets. In this case, BW-Raft increases the number of secretary, reducing the write latency and improving the performance. Besides, due to part of the log replication tasks are offloaded from the leader to the local secretary, the network bandwidth of the leader is greatly reduced, which can also improve the performance in the limited network bandwidth in Figure 11(c). In the case of read-only workload, appropriately increasing the number of observers can greatly improve the throughput and reduce the response overhead as in Figure 10(a)(b).

Impact of Design Factors: In our provision process, we have two major factors (i.e., workload R/W ratio α and spot instance failure rate ϕ). The factor α affects the provision process in BW-Raft, while the factor ϕ could collapse the provision decision. Figure 12 illustrates the impact of α. The average goodput is increasing linearly when BW-Raft serves more reads than writes. BW-Raft can handle reads well because BW-Raft abusively employs many observers to serve reads. These observers are cheap, thus the overall expense in BW-Raft grows much slower than performance gain.

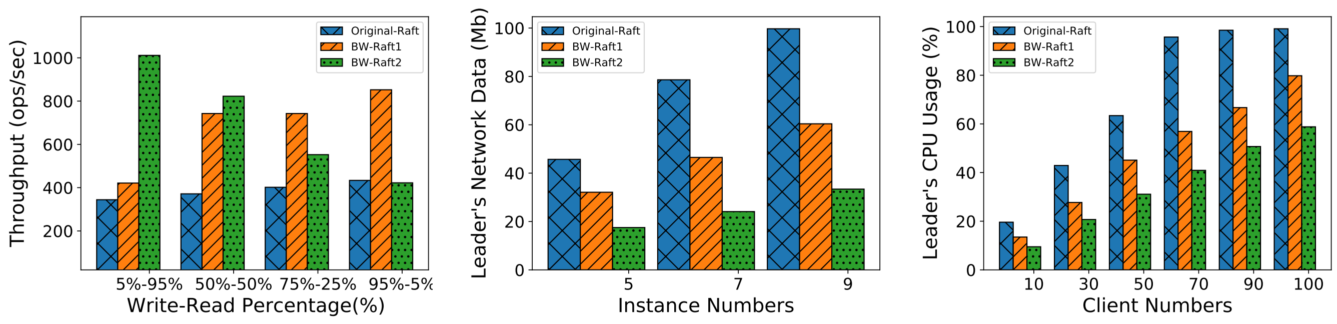
Figure 13 shows how spot instances fail affects BW-Raft. In our model, we assume a static ϕ based on history analysis, however, ϕ is hard to predict at runtime. When this failure rate increases in a single site, BW-Raft gradually reduces the number of secretaries at this site, and increases the number of observers in other sites. As such, the number of followers decreases since secretaries decrease. BW-Raft reduces secretaries on purpose, in order to reduce the possibility of update failure, and thus the cost from the consensus management. While BW-Raft suffers performance loss from writes, it hedges performance gain from reads as it hires more observers.

Limitations of BW-Raft in the Wild: We report the server-side



(a) Read Goodput (b) Read Latency (c) Write Goodput (d) Write Latency

Fig. 10. Goodput and latency comparison between BW-Raft with different numbers of secretaries and observers. In figure(a)(b) BW-Raft1 and BW-Raft2 means BW-Raft with one observer and two observers. In figure(c)(d) BW-Raft1 and BW-Raft2 means BW-Raft with one secretary and two secretaries.



(a) YCSB Benchmark (b) Network Data Usage (c) CPU Usage

Fig. 11. Comparison of YCSB's benchmark result and leader resource usage after BW-Raft scale out. In figure(a) BW-Raft1 and BW-Raft2 means BW-Raft with one secretary and one observer. In figure(b)(c) BW-Raft1 and BW-Raft2 means BW-Raft with one secretary and two secretaries.

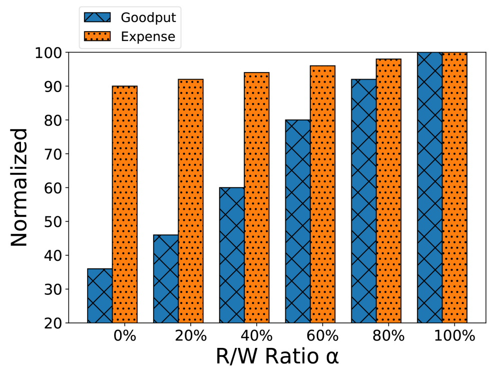


Fig. 12. The impact of R/W ratio α in BW-Raft.

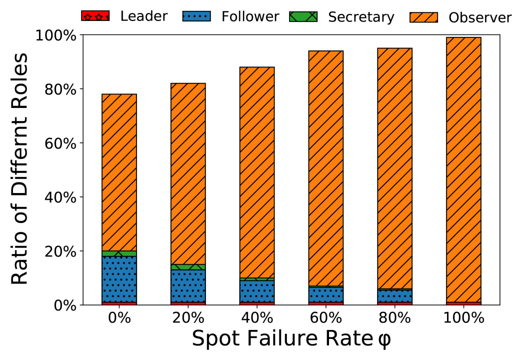


Fig. 13. The impact of spot failure rate ϕ in BW-Raft .

statistics in Figure 14. In order to provide large-scale node raft scenarios, we assign a setup as, EU-Frankfort 20 node, Asia-Singapore 55 node, US-East 70 nodes, US-West 75 nodes, from demand instances.

As shown in Figure 14, BW-Raft hires 15-20X more spot instances than on-demand ones. On the one hand, when BW-Raft leases on-demand instance, it takes the maximum use of the resource, exhibiting more than 80% utilization in all sites. On the other hand, BW-Raft abusively leases spot instances without fully exploiting its resources, with an average of 35% utilization. It is mainly because of the failure-prone behavior of spot instances that BW-Raft only uses a short period during their living period. If the burstable period of spot instance can be accurately predicted, BW-Raft can provide much better performance at scale-out. In addition, BW-Raft can only run in a non Byzantine environment. If there are malicious nodes involved, BW-Raft will not be able to guarantee the consistency of the system.

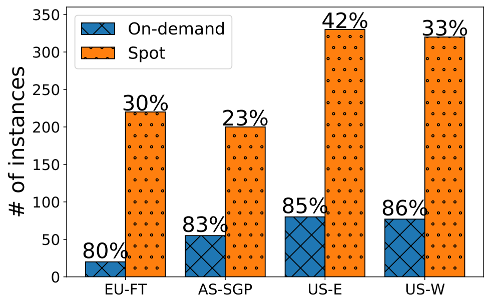


Fig. 14. Number of leased instances and average utilizations (on top of the bar) of BW-Raft in EU-Frankfort, Asia-Singapore, US-East, and US-West.

6 Related Work

Consensus algorithms, such as Paxos and Raft, are designed to maintain consensus across multiple shared data replicas. These algorithms scales by sharding [34] or automatically adding/-dropping followers [17]–[19], [36]. However, for high write throughput, applications turn to over-provisioned sharding, multiplying inefficiency. Our work, BW-Raft scales incrementally and does so using cheap, failure-prone spot instances.

Raft Consensus Algorithm: In the past few decades, researchers have proposed a large number of consensus algorithms [26], [32]. Raft algorithm [32] is one of the most widely used [4], [12]. Many companies have found that Raft is easy-to-implement and provides good performance. Pâris et al. reduced energy footprint of Raft [33]. Gramoli et al. put forward a fast consensus-based dynamic reconfigurations method which can speedup a primary-based rolling upgrade [18]. Copeland et al. propose a Byzantine Fault Tolerant variant of the Raft consensus algorithm [16]. These approaches have not explored efficient scaling out on Raft, while our work focusing on improving the scale-out performance with Raft on spot instances.

Geo-Replication: Droopy and Dripple [28] , two sister approaches, reduce latency by dynamically reconfigure leader set. Tuba [13] improves utility by automatic reconfiguration. SPANStore [40] offers low cost storage services making use of the price difference between suppliers. Cadre [41], Lynx [45], and Flutter [22] achieve low latency by avoiding long distance transmission.

Spot Instance Market: In the cloud market, suppliers provide on demand instance and spot instance. Spot instance is usually much cheaper than on-demand instance. However, due to spot instance is unstable and may stop at any moment, it is not reliable for data tasks, especially in maintaining data consistency. There are many research reduce cost by using spot instances. EAIC [23] reduce cost by adaptive checkpointing. And PADB [35] algorithm was put forward get maximized mean profit.

7 Conclusion

In a cloud computing environment, it is important to support intensive data service at scale out. While preserving strong consistency, it is expensive and complex support this data-intensive services at geo-diverse sites. In this paper, we proposed BW-Raft, an extended Raft framework that offloads log management to secretaries and offloads read request to observers as a new scheme to support strong consistency between services. We designed a global resource management to make this management cheap. In practice, we prototype a key-value store service in BW-Raft framework. Our source code can be found at [3]. We analyzed the performance of BW-Raft with many other protocols, and concluded that that (1) BW-Raft significantly boosts throughput by up to 9X, compared to Raft, and (2) BW-Raft is 84.5% cheaper than Multi-Raft. In general, BW-Raft is a practical framework to support scale out data-intensive computing across geo-diverse sites. In future work, we intend to extend BW-Raft to private and federated clouds to obtain the performance and cost saving of BW-Raft under limited spot instance resources.

References

[1] *Amazon AWS EC2 Services*. https://aws.amazon.com/ec2/.
[2] *Azure Low Priority VMs*. https://azure.microsoft.com/en-us/services/virtual-machines/.
[3] *BW-Raft Source Code*. (https://good.ncu.edu.cn/gitlab/yunxiao/BW-RAFT).
[4] *Cockroach Labs*. https://www.cockroachlabs.com.
[5] *Google Cloud Preemptible VMs*. https://cloud.google.com/preemptible-vms/.
[6] *Google Cluster Data*. https://github.com/google/cluster-data.
[7] *Grpc*. https://www.grpc.io/.
[8] *Leveldb*. https://github.com/google/leveldb.
[9] *Open source, containers, and Kubernetes — CoreOS*. https://coreos.com.
[10] *Oracle — Integrated Cloud Applications and Platform Services*. https://www.oracle.com.
[11] *Production-Grade Container Orchestration - Kubernete*. https://kubernetes.io.
[12] *TiDB*. https://github.com/pingcap/tidb.
[13] Masoud Saeida Ardekani and Douglas B. Terry. A Self-Configurable Geo-Replicated Cloud Storage System. *OSDI '14*, pages 367–381, 2014.
[14] T. Castiglia, C. Goldberg, and S. Patterson. A hierarchical model for fast distributed consensus in dynamic networks. 2020.
[15] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.
[16] Christopher Copeland and Hongxia Zhong. Tangaroa: a byzantine fault tolerant raft, 2016.
[17] Nan Deng, Christopher Stewart, Daniel Gmach, Martin Arlitt, and Jaimie Kelley. Adaptive green hosting. In *Proceedings of the 9th international conference on Autonomic computing*, pages 135–144. ACM, 2012.
[18] Vincent Gramoli, Len Bass, Alan Fekete, and Daniel W Sun. Rollup: Non-disruptive rolling upgrade with fast consensus-based dynamic reconfigurations. *IEEE Transactions on Parallel and Distributed Systems*, 27(9):2711–2724, 2016.
[19] Aaron Harlap, Andrew Chung, Alexey Tumanov, Gregory R Ganger, and Phillip B Gibbons. Tributary: spot-dancing for elastic services with latency slos. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 1–14, 2018.
[20] Chia-Chen Ho, Kuochen Wang, and Yi-Huai Hsu. A fast consensus algorithm for multiple controllers in software-defined networks. In *Advanced Communication Technology (ICACT), 2016 18th International Conference on*, pages 112–116. IEEE, 2016.
[21] Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. Raft refloated: do we have consensus? *ACM SIGOPS Operating Systems Review*, 49(1):12–21, 2015.
[22] Zhiming Hu, Baochun Li, and Jun Luo. Time-and cost-efficient task scheduling across geo-distributed data centers. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):705–718, 2018.
[23] Itthichok Jangjaimon and Nian-Feng Tzeng. Effective cost reduction for elastic clouds under spot instance pricing through adaptive checkpointing. *IEEE Transactions on Computers*, 64(2):396–409, 2015.

[24] Nicholas K Jong and Peter Stone. State abstraction discovery from irrelevant state variables. In *IJCAI*, volume 8, pages 752–757, 2005.

[25] Robert D. Kleinberg. A multiple-choice secretary algorithm with applications to online auctions. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, 2005.

[26] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[27] Y. Li, B. An, J. Ma, D. Cao, and H. Mei. Spottune: Leveraging transient resources for cost-efficient hyper-parameter tuning in the public cloud. *IEEE*, 2020.

[28] Shengyun Liu and Marko Vukolić. Leader set selection for low-latency geo-replicated state machine. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):1933–1946, 2017.

[29] Nathaniel Morris, Christopher Stewart, Lydia Chen, Robert Birke, and Jaimie Kelley. Model-driven computational sprinting. In *Proceedings of the Thirteenth EuroSys Conference*, page 38. ACM, 2018.

[30] Yipei Niu, Fangming Liu, Xincai Fei, and Bo Li. Handling flash deals with soft guarantee in hybrid cloud. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[31] Yipei Niu, Bin Luo, Fangming Liu, Jiangchuan Liu, and Bo Li. When hybrid cloud meets flash crowd: Towards cost-effective service provisioning. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1044–1052. IEEE, 2015.

[32] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[33] Jehan-François Pâris and Darrell DE Long. Reducing the energy footprint of a distributed consensus algorithm. In *Dependable Computing Conference (EDCC), 2015 Eleventh European*, pages 198–204. IEEE, 2015.

[34] Jun Rao, Eugene J Shekita, and Sandeep Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proceedings of the VLDB Endowment*, 4(4):243–254, 2011.

[35] Yang Song, Murtaza Zafer, and Kang Won Lee. Optimal bidding in spot instance market. In *Proceedings - IEEE INFOCOM*, pages 190–198, 2012.

[36] Cheng Wang, Bhuvan Urgaonkar, George Kesidis, Aayush Gupta, Lydia Y Chen, and Robert Birke. Effective capacity modulation as an explicit control knob for public cloud profitability. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 13(1):2, 2018.

[37] Zizhong Wang, Tongliang Li, Haixia Wang, Airan Shao, Yunren Bai, Shangming Cai, Zihan Xu, and Dongsheng Wang. Craft: An erasure-coding-supported version of raft for reducing storage cost and network cost. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 297–308, 2020.

[38] Rich Wolski, John Brevik, Ryan Chard, and Kyle Chard. Probabilistic guarantees of execution duration for amazon spot instances. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Doug Woos, James R Wilcox, Steve Anton, Zachary Tatlock, Michael D Ernst, and Thomas Anderson. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 154–165. ACM, 2016.

[40] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. SPANStore: cost-effective geo-replicated storage spanning multiple cloud services. *SOSP '13*, pages 292–308, 2013.

[41] Zichen Xu, Nan Deng, Christopher Stewart, and Xiaorui Wang. Cadre: Carbon-aware data replication for geo-diverse services. In *2015 IEEE International Conference on Autonomic Computing*, pages 177–186. IEEE, 2015.

[42] Zichen Xu, Christopher Stewart, Nan Deng, and Xiaorui Wang. Blending on-demand and spot instances to lower costs for in-memory storage. In *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE*, pages 1–9. IEEE, 2016.

[43] Zichen Xu, Christopher Stewart, and Jiacheng Huang. Elastic, geo-distributed raft. In *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*, pages 1–9. IEEE, 2019.

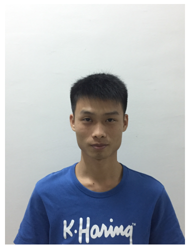
[44] Xiaomeng Yi, Fangming Liu, Zongpeng Li, and Hai Jin. Flexible instance: Meeting deadlines of delay tolerant jobs in the cloud with dynamic pricing. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 415–424. IEEE, 2016.

[45] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. *SOSP '13*, pages 276–291, 2013.

Yunxiao Du received his B.Eng. degree from Nanchang University. He is currently a M.Eng. student in Zhejiang University. His research interests include Distributed system , Database system.

Zichen Xu received the Ph.D. degree from the Ohio State University. He is a full professor with Nanchang University, China. His research spans in the area of data-conscious complex system, including profile analysis, system optimization, and storage system design/implementation. He is a member of the IEEE and ACM.

Kanqi Zhang is studying in Nanchang University at present. She is going to the University of Chinese Academy of Sciences to pursue her Ph.D. degree in research of network and computer architecture .



Jie Liu received his B.Eng. degree from Nanchang University. He is now a senior engineer in DiDi infrastructure department. His team is work in develop a distributed kv storage database.

Christopher Stewart received the Ph.D. degree from the University of Rochester. He is an Associate Professor in the Computer Science and Engineering Department at The Ohio State University. His research accomplishments span multiple fields with award winning papers in forums ranging from performance modeling to computer networking to sustainable systems engineering. He was the founding Chief Editor of the Sustainable Computing Register, the regular publication of the IEEE STC on Sustainable Computing.

Jiacheng Huang received his B.Eng. degree from Nanchang University. He is currently a M.Eng. student in Wuhan University. His research interests include Distributed system, Embedded system.