RAFTing Over on Geo-Diverse Spot markets

Zichen Xu[1] Christopher Stewart[2] Jiacheng Huang[1]
[1]Generic Operational and Optimal Data Lab, The Nanchang University
[2]Computer Science and Engineering, The Ohio State University

Abstract

Raft is a networked service used to synchronize cloud workloads at different locations. It is widely used, especially by workloads that span geographically distributed sites. As these workloads grow, Raft's costs should grow proportionally. However, auto scaling approaches for Raft inflate costs by provisioning at all sites when one site exhausts its local resources. This paper presents a Raft implementation, i.e., gRaft, that enables precise auto scaling on spot markets. gRaft extends Raft with the following abstractions: (1) secretaries which relieve log processing from leader nodes and (2) observers which relieve read requests from followers. These abstractions are stateless, allowing for elastic auto scaling, even with unreliable spot instances. gRaft preserves strong consistency guarantees provided by Raft. We set up and tested gRaft with multiple auto scaling techniques using traces from Google. gRaft scales in resource footprint increments 5-7X smaller than Multi-Raft, the state of the art. Using spot instances, gRaft reduces costs by 84.5% compared to Multi-Raft. gRaft improves goodput of 95th-percentile SLO by 9X.

I. Introduction

Raft is a networked service that supports strong consistency. Distributed systems use Raft to reach consensus between software components. For example, components must agree on locking ownership and elements in synchronized queues. Raft is used in practice by large-scale platforms, such as Google Kubernetes [3]. These platforms, i.e., the clients for Raft, suffer inflated costs when Raft employs inefficient scaling techniques.

Software threads in Raft are either *leaders* or *followers*. At all times, Raft allows zero or one leader as elected by followers. Followers periodically message the leader to check for failures. Upon leader failure, followers call for an election to name a new leader thread. The leader has a more demanding workload: it pushes writes to all available followers and tracks which followers have confirmed the most recent writes. The leader bottlenecks Raft throughput.

When workload demands overwhelm the leader, 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 more powerful computers are unavailable or unaffordable, the leader's workload must be split (*scale out*). Herein lies the problem: Raft permits one leader only. One leader is essential to make the algorithm understandable, correct and easy to implement. Multi-Raft [10] scales out by replicating leaders and followers, and splitting data between the two replicas. Each replica implements Raft, providing strong consistency. Between replicas, 2 phase commit provides strong consistency. Multi-Raft can grow throughput as workload demand increases, but it is costly. Each scale out operation doubles resource footprint.

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

We present *gRaft*, an extension that scales well with geo-distributed resources. gRaft supports 4 types of software threads: followers, leaders, secretaries and observers. Like original Raft, gRaft permits zero or one leader based on majority votes by followers. In gRaft, leaders outsource a portion of their workload to secretaries, reducing workload imbalance. Observers answer read-only queries, reducing workloads for followers. gRaft allows multiple secretaries and observers to run simultaneously. Despite the addition of secretaries and observers, gRaft preserves the safety guarantees of Raft. We employ *state irrelevancy* [13] to prove that secretary and observer failures do not affect correctness.

Secretaries and observers are stateless and execute at any geo-distributed site. They can be deployed incrementally to achieve high throughput at low cost as client workloads grow. We present a cost modeling approach to discover high throughput, low latency, and low cost settings for gRaft. Our approach accepts time sensitive parameters on cost and latency. This allows gRaft to use 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 [24], [16], [17], [25]. gRaft uses safe, on-demand instances for leaders and followers, and spot instances for secretaries and observers.

We deployed gRaft on Amazon for 9 weeks serving Google traces [2]. gRaft purchases low-cost spot instances in geo-distributed sites to scale-out the performance within the budget, and boosts both read and write throughput, compared to Raft [18]. gRaft adaptively boosts read and write throughput which incurs less than 85% overhead compared to Raft. gRaft

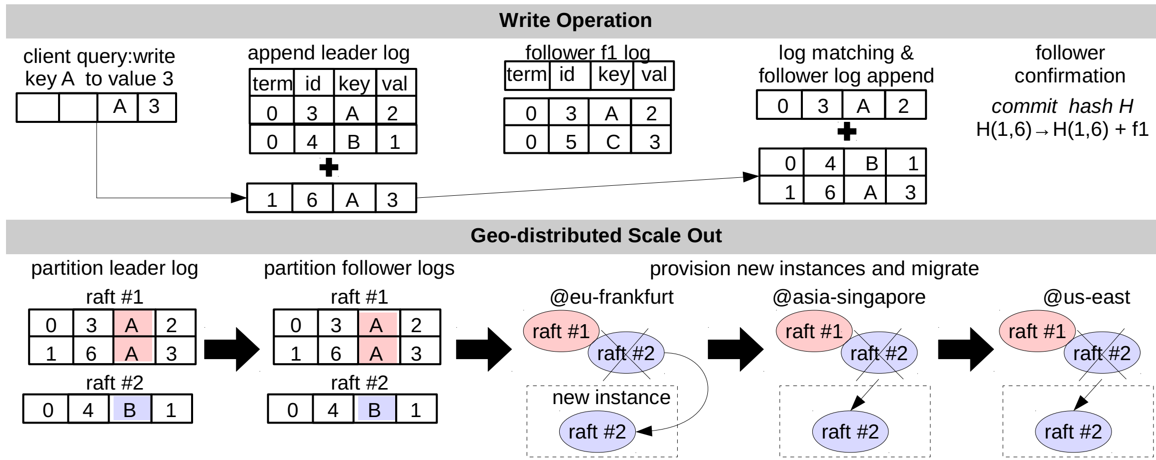

Fig. 1. Execution of writes in Raft (top) and geo-distributed scale out in Multi-Raft (bottom).

scales in increments 5-7X smaller than Multi-Raft, the state of the art. gRaft reduces costs by 84.5% compared to Multi-Raft and improves throughput of 95th percentile SLO by 9X.

Contributions are listed below:

- We present gRaft, an extension to the widely used Raft algorithm that can use cheap spot instances.
- We use state irrelevancy to prove that gRaft preserves key features of Raft.
- We present a cost modeling approach to manage cloud instances for gRaft.
- We built gRaft and deployed it for 9 weeks. gRaft reduces costs by 84.5% compared to Multi-Raft.
- We used real systems to evaluate gRaft, showing that offload can proceed quickly enough to speedup consensus protocols.

II. Background

Raft is commonly realized as a networked key-value store, supporting the following queries:

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

Raft sets a global processing order for write queries and ensures that subsequent read queries will return the same value or the value of a later write query. In other words, Raft supports linearizable consistency.

Figure 1 (top) depicts write query execution. The client provides key and value when issuing the query. The leader uses current state of its log to append term and revision id. Leader elections increment term. Write queries increment revision id.

Hardware failures and network partitions can make leader and follower logs diverge. However, Raft ensures that nodes always agree on one leader. As Figure 1 (top) shows, the leader gets logs from followers and align them with their own. Raft requires followers to accept updated logs from the leader. The process of aligning logs is called *log matching*.

Scaling out with Raft: Raft permits only one leader at a time and the leader has a demanding workload. Naively adding follower nodes does not improve throughput. In fact, it often degrades throughput by causing more log matching.

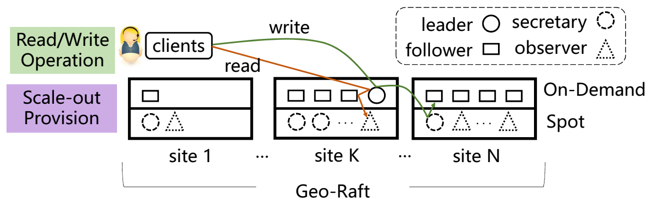

Fig. 2. gRaft can scale out using spot market resources.

Multi-Raft is a widely used approach to scale out. As shown in Figure 1 (bottom), it 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 election (e.g., raft #1 and raft #2 in Figure 1). Between leaders of these Raft, they communicate and keep in consistency based on 2 Phase Commit.

The Multi-Raft approach preserves linearizability within key ranges and scales out well. However, it is not cost efficient. Scale-out can double the resource footprint to resolve light bottlenecks at only one leader nodes.

III. Design

gRaft extends Raft with two new types of software threads: *secretaries* and *observers*. Secretaries relieve log analysis from the leader and observers relieve read requests from followers. Secretaries and observers are stateless, allowing for scale up and down.

gRaft ensures linearizable consistency. Like Raft, gRaft allows only one leader thread. Also, the leader is selected from followers via election. Secretaries and observers can not vote. As such, gRaft can support any number of secretaries and observers (and handle their failures as well). In this section, we will show that gRaft provably maintains correctness guarantees provided by Raft.

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 gRaft. 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 as spot prices change.

This section first details our algorithm, with a focus on leader election, secretary and observer failures, and the main safety guarantee. Then, we present a modeling approach for managing cloud resources cost effectively.

A. The gRaft Algorithm

gRaft, 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 gRaft, starting from leader election.

gRaft 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 III.1. *(**Leader Election Safety**). Like Raft, there is at most one leader per term in gRaft.*

The leader maintains its role by sending heartbeat messages. After receiving a heartbeat, followers set a random timer. If they do not receive a message before the timer triggers, the follower calls for an election and stops all secretary threads (i.e., Step (1) in Figure 3). The follower increments the term in its logs and tells other followers that it is a “candidate” for leader. Followers vote for a candidate whose log is not older than its own. If voting times out, the election will restart again (not shown in Figure 3). If a candidate/follower gets the majority votes from most followers, this node will become a new leader and gRaft provisions secretaries for the new leader (i.e., Step (2) in Figure 3).

gRaft starts each term ($\bar{T}$) with a leader election. The leader orchestrates normal operations (log management), notifies higher term when new update, as shown in Step (3) in Figure 3. Meanwhile, the leader in gRaft always tells followers which secretaries are responsible for this log, such that log management can be offload to assigned secretaries (i.e., Step (5) in Figure 3). If the election is a failure, a new term starts, with a new election.

Property III.2. *(**gRaft 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 [21], this property provides consistency guarantees for the logs between all leader, secretaries, and followers in the protocol.

Stateless Operations in gRaft Leader Election: Besides the whole leader election process, gRaft employs stateless threads as *secretaries* and *observers*. After a successful election, when

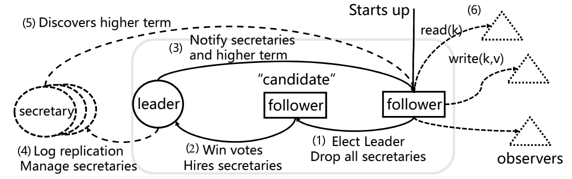

Fig. 3. The state machine and stateless nodes in gRaft. The central part in rectangle preserves the classic Raft state machine.

the term ($\hat{T}$) is longer than a predefined period ($T < \hat{T}$), leader will re-provision secretaries and offload logs (i.e., Step (4) in Figure 3) by the end of this period. gRaft adopts this re-election mechanism to mitigate the fail-prone feature from spot instances, without incurring noticeable delay/cost.

Followers that operate in gRaft can arbitrarily hire cheap local spot instances as observers. Each time a follower receives a write, it will append this write to all its linked observers without waiting for the commit. When the follower receives a read, it reroute the job to committed observers. If no observers commit previous appending log, this follower execute the read itself (i.e., Step (6) in Figure 3).

Once the secretary learns the majority followers have acknowledged the new entry, it will notify the leader and mark this write as committed. Note that, the leader need not receive commit from every single secretary before proceeding requests. The majority of secretary acknowledgment confirms the majority notes from followers. Then, the leader then executes the command contained in the committed entry on the state machine and responds to the client with the output of the command. The followers are informed that they can safely execute the command on their state machines, gossiping with its observers. Thus, in both leader election and log replication, both secretaries and observers are state irrelevant.

Property III.3. *(**State Irrelevancy**). If any secretaries/observers fail during the term at different locations, then the logs are still matched thus state machines in both leader and followers are irrelevant to secretaries and observers.*

Proof. We first prove the State Irrelevancy in secretaries. Given an execution trace τ of gRaft, assuming one secretary fails during the trace, we shall find an existing σ such that τ linearizes to σ. This σ is easy to find. We revert gRaft 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, so the choice is well defined.

□

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

Theorem III.4. *Properties III.1–III.3 imply linearizability in gRaft.*

The proof trivially follows the linearizability proof in Raft [21]. Once an entry is committed, it becomes durable,

in the sense that its effect will never be forgotten by gRaft. In all, gRaft operates log replication with linearizability.

Property III.5. ***(Linearizability)****. gRaft implements a linearizable state machine.*

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

B. Managing resources for gRaft

Leader elections disrupt normal query processing, causing processing times to deviate from standard distributions. These delays force clients to wait to acquire locks, affecting the rate at which they issue new queries. To manage resources, we model processing times and arrival rates with a G/GI/m model, i.e., a model that supports generic distribution of arrival rate and generic independent distribution for mean processing time. There are m cloud instances to process queries. Allen-Cunneen approximates [5] mean response time for G/GI/m models. Equation 1 models mean response time ($\bar{t}$).

$$\bar{t} = \frac{1}{\mu} + \frac{\rho}{\mu(1-\rho)}\left(\frac{C_A^2 + C_B^2}{2}\right) + \frac{2E[I] + \lambda E[I]^2}{2(1+\lambda E[I])} \tag{1}$$

In this equation, generic independent distribution for processing time is $\frac{1}{\mu}$, mean query arrival rate is λ, mean utilization of a server is $\rho = \frac{\lambda}{\mu}$, C_A^2 and C_B^2 represents the squared coefficient of variation of request inter-arrival times and request sizes, respectively, and $E[I]$ is mean initial set-up time for processing the first query.

Given a set of m instances, we separate them into m_l and m_f as leader and followers, using on-demand instances; m_s and m_o represent secretaries and observers, using spot instances. Thus, the total number of required instances is $m = m_l + m_f + m_s + m_o$. The estimated expense is in Equation (2).

$$c = c_1 m_l + c_2 m_f + c_3 m_s + c_4 m_o + C_0(m) \tag{2}$$

where c_k is the local unit price of selected instance in data center k, and C_0 is a linear cost function of networking based on the total number of instance m.

As shown in Equation 2, the total cost (c) of one instance provision to deploy gRaft depends on the price of instances for each software thread (c_i). The number of instances (m_i) also affects cost. For example, leasing too many failure-prone spot instances can increase costs enough to negate price savings. Also, arbitrarily increase the frequency of adapting spot instances that increase risks of failure and thus violating SLOs. We model the total revenue as a weighted function of (1) expense to lease instances and (2) failure rate of picking certain instances. Equations (3)-(9) show the model:

$$\text{Max } R \tag{3}$$
$$R \triangleq \{c_1, c_2, \ldots, c_{(m,n)}\} \tag{4}$$
$$c_i = r_j^a \vartheta_j \tag{5}$$
$$r_j = (1-\tau)\iota_j \cdot \rho + \tau\iota_j \cdot \rho(1-\phi) \tag{6}$$
$$\vartheta_j \triangleq \frac{\zeta}{\ell_j^{k_j}} \tag{7}$$
$$k_j = \begin{cases} 1, & \zeta \leq \varpi \\ \infty & \zeta > \varpi \end{cases} \tag{8}$$
$$\ell_i = \sum p_j T \tag{9}$$

In these equations, i indexes the ith leasing period of combined instances and j indexes the jth instance. R is the possible maximum revenue when examining a vector of possible selections, i.e., $\{c_1, c_2, \ldots, c_{(m,n)}\}$, where m is the total number of selected instances and n is the total number of instances available. c_j is the total revenue from serving workloads from selection j. The total revenue is a nonlinear combination of financial profit (ϑ_j) and correspondent performance variable (r_j). Parameters a is the weight coefficient.

Performance variable r_i captures performance from both on-demand and spot instances. τ is the ratio of hired spot instances in m nodes. one decision of selecting ι_j, ρ is the provisioned utilization from Equation 1 based on queuing theory. ϕ captures the spot failure rate. The total cost ϑ_j captures gaining from the possible SLO (ζ) while spending (ℓ_j) for leasing the ith selection. Note that we use k_j as the weight coefficient when SLO ζ violates certain threshold ϖ. Usually, we set $\varpi = 99.999\%$. In our current setup, we do not want any SLO violation in gRaft . Thus, violating the threshold ϖ can lead to infinitely large expense, thus a zero revenue. The parameter (ℓ_i) calculate the total expense from the ith selection in the period T. For the ease of our analysis, we set $T = 1$ hour.

gRaft creates secretaries for leader. When a new leader voted or all secretaries dead in the previous epoch, gRaft hires secretaries. If one leader fails, secretaries will be wasted and could be recollected in next epoch.

IV. Evaluation

gRaft is designed to maintain a cheap and scale-out raft protocol across on-demand and spot instances. In this section, we evaluate the performance of gRaft by prototyping it onto our physical testbed and Amazon AWS EC2 [1], [15].

System Setup: We built a physical testbed to illustrate the runtime performance of gRaft using Google workload. The testbed contains 96 nodes with Intel Xeon 2620 2.60GHz and 64GB DDR4 memory each. We also deploy gRaft onto instances (t2.small) in Amazon AWS EC2. For the burstable spot instance, we report 0.415$/H on average. Spot instances were up to 90% cheaper than their regular on-demand instances. The average lifetime of a spot instance was 32 minutes and prices between sites were not correlated.

In this paper, we mainly evaluate gRaft with the following baselines:

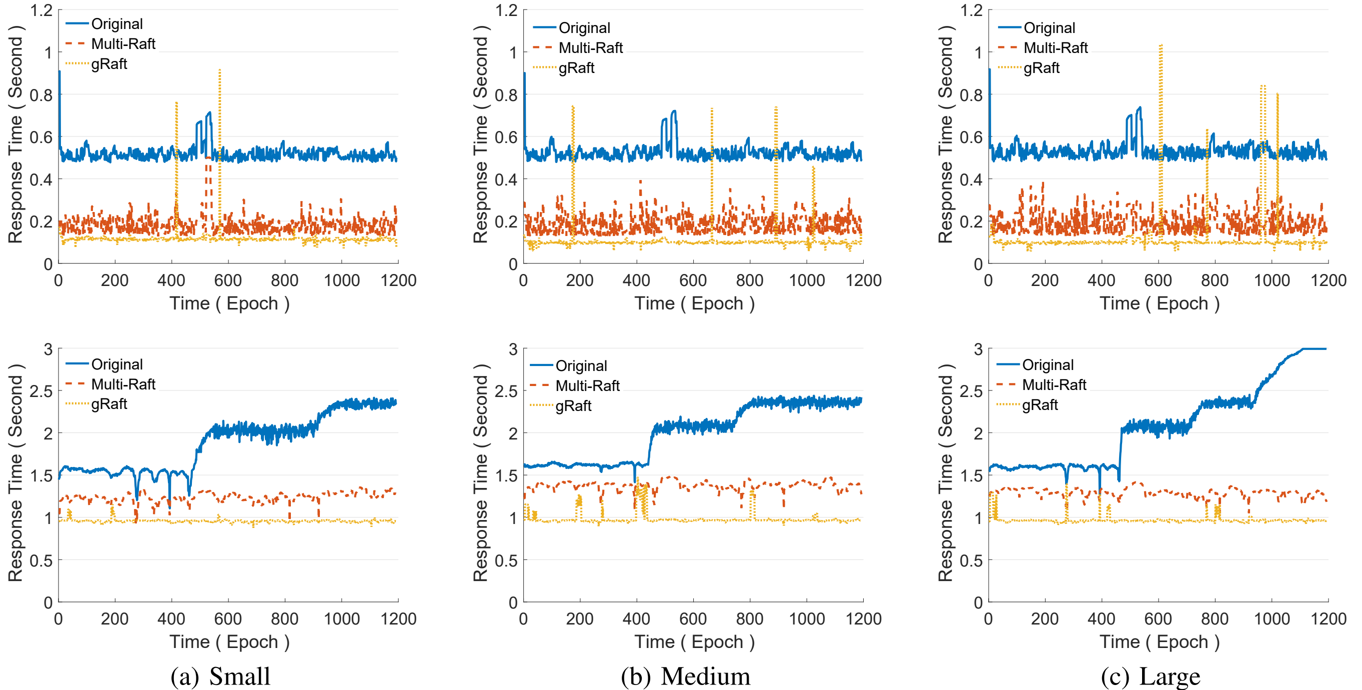

Fig. 4. 50 days Performance snapshots running **Read** (top) and **Write** (bottom) workloads. Y axis is in log scale.

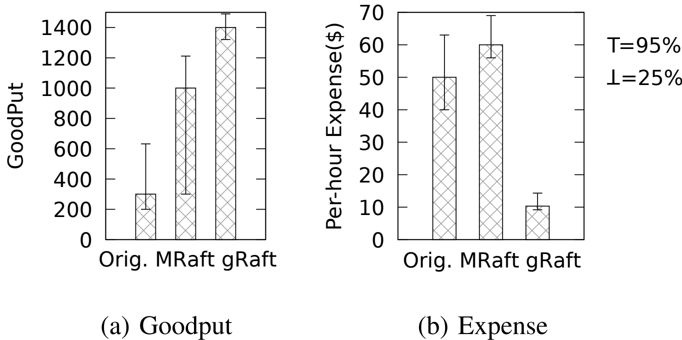

Fig. 5. Performance (a) and cost (b) comparison between gRaft and other baselines.

- *Original* implements a state-of-the-art Raft design from Ongaro et al. [18];
- *Multi-Raft* is a state-of-the-art multi-raft implementation using sharding [9];
- *Oracle* is a baseline based on offline analysis.

Workloads and Traces: We verified the performance of gRaft using real world workloads and traces. We use the popular Google cluster trace [2] which contains one-month job statistics in Google clusters, 2011. Workloads are random reads/writes, controlled ratio R/W batches, and read/write-only workloads.

A. Results

Performance Snapshots: First, we report the performance snapshot of running *gRaft* , *Multi-Raft*, and *Original*. The whole experiment lasts for 1200 epochs (i.e., 50 days) in Amazon EC2. Figure 4 plots the average latency when executing **Read** (top) and **Write** (bottom). For reads, gRaft 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 extreme cases, gRaft can perform badly, such as overshoots in epoch 410 and 578 in Figure 4(a)top. Similar overshoot happens when gRaft executes larger reads. Such overshoots happen when the majority of *observers* failed and gRaft 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 writes, gRaft scales out in proportion to the ever-increasing updates, nicely. Multi-Raft also scales, however, with a price of 3X larger response time due to maintaining the 2 pc commit between leaders. Original cannot handle constant updates when scaling-out. When the size of appended logs increases to a certain limit in all nodes, Original fails at the leader blocking the overall write performance, slowing down 2.5X compared to gRaft. Overall, gRaft shows significant performance improvement, as it scales in increments 3-12X compared to Multi-Raft and Original.

Overall Statistics: Figure 5 shows the overall performance and expense comparison between gRaft and other baselines. In goodput (effective throughput), gRaft is 7X and 1.5X, larger than Original and Multi-Raft, respectively. Considering both reads and writes, gRaft has smaller variation than Multi-Raft. gRaft has significantly smoothed write delay curve due to secretaries often reside in more sites than *followers*, which reduces unexpected long wide-area network delay. On the expense side, gRaft exploits cheap spot instances for secretaries and observers in many sites. gRaft 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 gRaft 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 barely zero performance.

V. Related Work

Consensus algorithms, such as Paxos and Raft, are designed to maintain consensus across multiple shared data replicas. These algorithms scales by sharding [19] or automatically adding/dropping followers [7], [8], [20]. However, for high write throughput, applications turn to over-provisioned sharding, multiplying inefficiency. Our work, gRaft 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 [12], [18]. Raft algorithm [18] is one of the most widely used [4]. Many companies have found that Raft is reliable and provides good performance. Ho et al. proposed a fast Paxos based consensus algorithm (FPC) which provides better performance compared to Raft but it harder to implement [9]. These approaches have not explored scaling with spot instances.

Geo-Replication: Droopy and Dripple [14] , two sister approaches, reduce latency by dynamically reconfigure leader set. Tuba [6] improves utility by automatic reconfiguration. SPANStore [22] offers low cost storage services making use of the price difference between suppliers. Cadre [23], Lynx [26], and Flutter [11] achieve low latency by avoiding long distance transmission.

VI. Conclusion

Raft, a widely implemented algorithm, does not scale well. The Multi-Raft extension can scale out, but inflates costs in geo-distributed settings. We proposed gRaft, a Raft extension that scales in increments 5-7X smaller than Multi-Raft, reducing cost inflation. gRaft transforms log management into a stateless task, allow Raft leaders to offload work to secretaries. By design, gRaft can exploit low cost but failure-prone spot instances—significantly reducing scale out costs. We prototyped gRaft on realistic clouds, and run continuously for over 2 months. We have observed that (1) gRaft significantly boosts throughput by up to 9X, compared to Raft and (2) gRaft is much 80% cheaper to deploy than Multi-Raft.

VII. Acknowledgement

This research was supported by the National Science Foundation China (NSFC) grant (617022501006873), and Jiangxi Province Science Foundation Youths Grant (708237400050).

References

[1] *Amazon AWS EC2 Services*. https://aws.amazon.com/ec2/.
[2] *Google Cluster Data*. https://github.com/google/cluster-data.
[3] *Production-Grade Container Orchestration - Kubernete*. https://kubernetes.io.
[4] *TiDB*. https://github.com/pingcap/tidb.
[5] F. Ahmad and T. Vijaykumar. Joint optimization of idle and cooling power in data centers while maintaining response time. In *ACM Sigplan Notices*, pages 243–256. ACM, 2010.
[6] M. S. Ardekani and D. B. Terry. A Self-Configurable Geo-Replicated Cloud Storage System. *OSDI '14*, pages 367–381, 2014.
[7] N. Deng, C. Stewart, D. Gmach, M. Arlitt, and J. Kelley. Adaptive green hosting. In *Proceedings of the 9th international conference on Autonomic computing*, pages 135–144. ACM, 2012.
[8] A. Harlap, A. Chung, A. Tumanov, G. R. Ganger, and P. B. Gibbons. Tributary: spot-dancing for elastic services with latency slos. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 1–14, 2018.
[9] C.-C. Ho, K. Wang, and Y.-H. 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.
[10] H. Howard, M. Schwarzkopf, A. Madhavapeddy, and J. Crowcroft. Raft Refloated. *ACM SIGOPS Operating Systems Review*, 49(1):12–21, 2015.
[11] Z. Hu, B. Li, and J. Luo. Time-and cost-efficient task scheduling across geo-distributed data centers. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):705–718, 2018.
[12] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, page 9. Boston, MA, USA, 2010.
[13] N. K. Jong and P. Stone. State abstraction discovery from irrelevant state variables. In *IJCAI*, volume 8, pages 752–757, 2005.
[14] S. Liu and M. Vukolić. Leader set selection for low-latency geo-replicated state machine. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):1933–1946, 2017.
[15] N. Morris, C. Stewart, L. Chen, R. Birke, and J. Kelley. Model-driven computational sprinting. In *Proceedings of the Thirteenth EuroSys Conference*, page 38. ACM, 2018.
[16] Y. Niu, F. Liu, X. Fei, and B. Li. Handling flash deals with soft guarantee in hybrid cloud. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.
[17] Y. Niu, B. Luo, F. Liu, J. Liu, and B. 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.
[18] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.
[19] J. Rao, E. J. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proceedings of the VLDB Endowment*, 4(4):243–254, 2011.
[20] C. Wang, B. Urgaonkar, G. Kesidis, A. Gupta, L. Y. Chen, and R. 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.
[21] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. 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.
[22] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: cost-effective geo-replicated storage spanning multiple cloud services. *SOSP '13*, pages 292–308, 2013.
[23] Z. Xu, N. Deng, C. Stewart, and X. Wang. Cadre: Carbon-aware data replication for geo-diverse services. In *2015 IEEE International Conference on Autonomic Computing*, pages 177–186. IEEE, 2015.
[24] Z. Xu, C. Stewart, N. Deng, and X. 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.
[25] X. Yi, F. Liu, Z. Li, and H. 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.
[26] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. *SOSP '13*, pages 276–291, 2013.