Raft API Comparisons on Both Performance and Developer Experience

1st Alejandro Ciuba
School of Computing and Information
University of Pittsburgh
Pittsburgh, USA
alejandrociuba@pitt.edu

1st Winston Osei-Bonsu
School of Computing and Information
University of Pittsburgh
Pittsburgh, USA
wio6@pitt.edu

***Abstract*—Consensus algorithms are a central tenant in creating reliable, stable distributed systems. While many consensus algorithms have been proposed and implemented, the Raft algorithm stands out due to its main design principle: intuitiveness. Thanks to this trait, dozens of officially documented Raft implementations across a wide variety of programming languages have been developed. However, very little work has been done comparing these implementations against each other. Our project seeks to compare implementations of Raft in a series of standardized experiments. From our experiment we create a table which compares the tested implementations across metrics like CPU utilization and messages passed. Lastly, given the fact Raft was originally designed with simplicity in mind, we conduct a brief on GitHub issues found across all three repositories with some basic LDA topic modeling. All our work is available on GitHub[1].**

I. Introduction

To create distributed environments whose node information is consistent and agreed upon throughout is the research behind consensus algorithms. While there have been many specific algoitihms for consensus, such as Paxos [1], recent years have shown the rise of a new leader-based consensus algorithm, Raft [2]. What makes Raft interesting is its design philosophy, which focused on algorithm intuitiveness. It is an algorithm specifically designed for students to understand, which has no doubt contributed to its widespread adoption. Due to Raft's popularity for consensus in distributed computing, the original researchers behind Raft have made a website[2] to host the plethora of activities, lectures and, most importantly for our study, implementations. There are over 100 documented implementations of Raft there, spanning programming languages of all levels. The table also highlights important guarantees each system can provide: leader election and log replication, persistence, membership changes and log compaction.

A. Problem Statement & Research Questions

Given the importance of Raft and consensus algorithms in general for consistent, reliable, fault tolerant distributed systems, we believe it is important to understand the nuances between different implementations of Raft. We know the theory and implementation of algorithms can differ greatly,

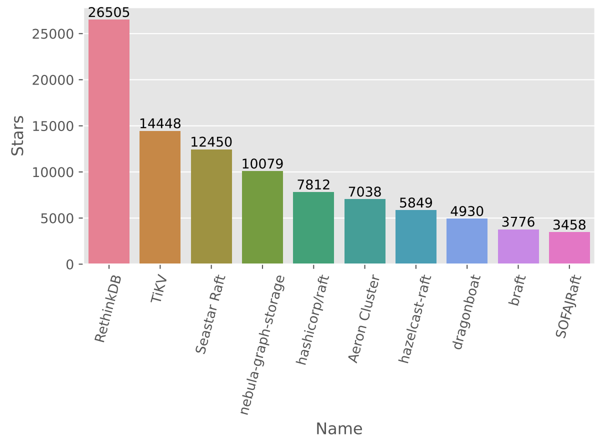

Fig. 1. Top ten most popular Raft repositories by number of stars on GitHub.

and that implementations can vary in speed, effectiveness and usability. **Our project seeks to compare different implementations of Raft on a benchmark test and compare their performance and examine common issues developers face by examining their GitHub issues.** We focus on the following research questions:

RQ1: Which implementation will require the least amount of memory?
RQ2: Which implementation will be the fastest to fully commit a log entry?
RQ3: Which implementation produces (on average) the lowest amount of messages in our simulations?
RQ4: Is the number of closed issues dependent on the repository?

II. Related Work

A. Consensus Algorithms

Consensus algorithms allow systems to achieve reliability and agreement across nodes as they coordinate to accomplish an overarching task. These algorithms maintain consistency via system component states, allowing for each node to correctly progress toward the intended objective. We define consensus to be achieved when all non-faulty components concur on a specific matter at a given moment [3]. Given this, the design of

[1]https://github.com/AlejandroCiuba/CS2510-Final-Project
[2]https://raft.github.io

consensus algorithms must focus on how nodes communicate and synchronize their actions, the types of component failures they can withstand, and the protocols needed maintain any achieved consensus.

Systems that attempt to achieve consensus can vary in the degree of coordination between their components. While some systems may employ a fully synchronous model—coordinating operations in batches and confirming among components, more practical systems often adopt a partially synchronous approach. In such systems, operations are not strictly coordinated, but message delivery between nodes is eventually guaranteed (assuming a reliable means of communication between nodes). Thus, addressing component failures becomes the next hurdle in achieving consensus. Failures are typically classified into crash failures[3] and Byzantine failures[4]. Not all consensus algorithms are designed to tackle both types of failures, but an effective algorithm will have strategies to handle the anticipated failure modes within its operational context. The consensus protocols themselves establish the rules for network communication and how to deal with component failures, ensuring a consistent state across nodes. Protocols for establishing consensus can so far be categorized into proof-based consensus algorithms, and voting-based consensus algorithms [4].

1) Proof-Based: These algorithms are commonly seen in contexts where security and decentralization is of high priority—a modern example of this being blockchain technology. In such approaches, components need to convey proof that they've expended some defined resource to be able to partake in the process of consensus. Examples include Proof of Work (PoW) and Proof of Stake (PoS), among others.

2) Voting-Based: These algorithms employ a more democratic process, where all nodes have the opportunity to cast votes to arrive at collective decisions. This approach is particularly effective in networks with known and trusted participants. Examples of these types of consensus algorithms would be those like Practical Byzantine Fault Tolerance (PBFT), which enables consensus in the face of Byzantine faults. Consensus protocols like Paxos and Raft, are commonly used in practice.

B. Raft

As mentioned previously, the Raft consensus algorithm is intended to be the focus of this project. Generally, Raft is considered to one of the more understandable and easier to implement consensus algorithms compared to those like Paxos, and it is able to do this without sacrificing correctness and performance. Part of this is has to do with how it has been presented and communicated in academic and practical contexts [5]. As a general overview, Raft clusters are composed of several nodes, one of which is elected as the leader. The leader handles all client interactions that would modify the state of the system, as well as log entry replication and consistency. If the leader fails, a new leader is elected. This election process ensures that there is always a single leader at any given time, simplifying the management of the replicated log. Once a leader is elected, it begins to process client requests, each of which contains a command to be executed by the replicas. The leader appends the command to its log as a new entry and then replicates this entry to the follower nodes. Because raft logs keep note the current election term in which they were created, the system is able to detect inconsistencies and ensure log entries are replicated in the correct order. An entry is considered committed when it's safely replicated on a majority of the nodes. The leader keeps track of this and communicates the commit index to followers, indicating which entries are safe to apply [2].

[3]A component abruptly halts and does not resume.

[4]A component acts maliciously or shares incorrect information with the other components.

Name	Language	Developer	Created	Stars
PySyncObj	Python	Filipp Ozinov	Dec 2015	680
HashiCorp Raft	Go	HashiCorp	Nov 2013	7.8k
TiKV	Rust	TiKV	Jan 2016	14.5k

TABLE I
THE THREE CHOSEN IMPLEMENTATIONS. ALSO NOTE THAT PYSYNCOBJ CAN BE REFERRED TO AS "BAKWC." FOR API DETAILS, SEE III-C.

C. Algorithm Implementation Comparison

The comparison of differing algorithms is a fundamental research area of computer science. Our research is not the comparison of different consensus algorithms, but different implementations of the same algorithm. We are inspired by the recent work of [6] who compared a traditional Raft implementation to their "energy efficient" version. Their main metric was the number of messages exchanged within the system. Our project goes beyond this, considering other metrics like CPU utility and maximum memory usage. Additionally, given that a core foundation of Raft is its ease-of-learning, we want to see if its implementations follow that tradition by examining some aspect of their developer experience [7]. We try to examine the status of GitHub issues.

III. Methodology

A. Implementation Selection

When choosing our implementations, we wanted to pick 2-5 which satisfied the following criteria:

- Stand-alone functionality.
- Raft website endorsement (linked from the website).
- Their core implementation was in a different language from other selected versions; although their interface could be in the same language.
- All four criteria listed in the Raft website table (leader election and log replication, persistence, membership changes and log compaction).

Another criteria we originally had was that accessing the node cluster should be possible through a RESTful framework. This requirement was loosened to allow for our final selection TiKV. The three selected implementations are described in Table I. Both TiKV and HashiCorp are among the top ten most starred repositories according to the Raft website (Figure 1).

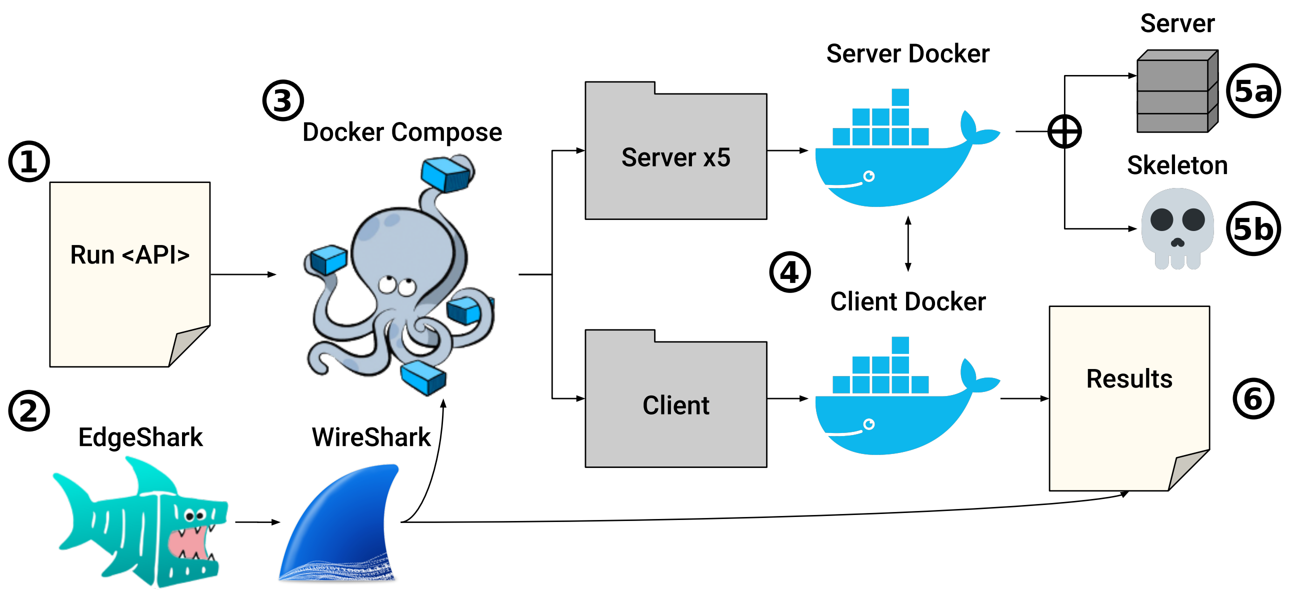

Fig. 2. The full layout of our standard experimental setup. Full step description in Section III-C.

B. Evaluation Metrics

Our evaluation metrics focus on three aspects: the CPU, the RAM and the functionality of the algorithm.

- **Commit Time:** The average time it takes to complete a single commit in seconds.
- **CPU Utilization:** The total user and system time spent on the CPU during the experiment as a percentage of the process' entire lifespan.
- **Memory Usage:** The maximum amount of RSS memory in kibibytes a single server process used in the experiment.
- **Log Size:** The final size of the log in bytes. This is the most subjective of our measurements as what can be considered "the log" varies greatly between the implementations. We also only consider the in-memory log, not anything potentially written to disk.
- **Average Messages Passed:** The number of messages passed between the leader and follower nodes. This is gotten by looking at all the packets sent from and to the leader node by other server ports and averaged across all runs.

C. Experimental Setup

Our experiment saw each implementation make a simple key-value storage system: a user can send the leader node key-value pairs to save. We ran each storage system 10 separate times back-to-back. The client sends 2,000 key-value pairs synchronously directly to the leader of a 5-node clusters. The client then reads back all keys. Each storage system was designed such that key-value committing on the log was done completely synchronously as well; the client sends the key-value pair and the storage system does not reply back until it has been fully committed on all node logs. Figure 2[5] overviews our experimental process.

1) We run `run.sh` to start our experiment and save the Docker Compose output to a results text file. The batched version also parses the results file into a CSV.
2) Simultaneously, we run WireShark (footnote five) with the EdgeShark plugin developed by Siemens [8]. The EdgeShark extension allows us to monitor packets specifically within the Docker Compose network via WireShark.
3) The bash script runs the implementation's Docker Compose (footnote five) YAML. The compose file launches five server instances, trying to ensure that the `servera` service is made the leader. The `client` service is always launched last. After sending and requesting the key-value pairs, it retrieves the statistics from the HTTP endpoint.
4) The leader server and client communicate with each other through HTTP endpoints with TiKV being the exception as you access the broker node via a built-in object.
5) The servers launched by the compose file are either a set of the actual full Raft implementation servers (5a) or "skeleton" servers (5b) which measure the process statistics without any Raft components. This accounts for the overhead our measurement tools and HTTP access point add.
6) Finally, all statistics from the HTTP endpoint are saved into a text file and packet information for all ten runs of a given implementation are saved into a PCAPNG file.

[5]Logos from https://github.com/siemens/edgeshark; https://www.wireshark.org; https://github.com/docker/compose; https://www.docker.com/company/newsroom/media-resources/

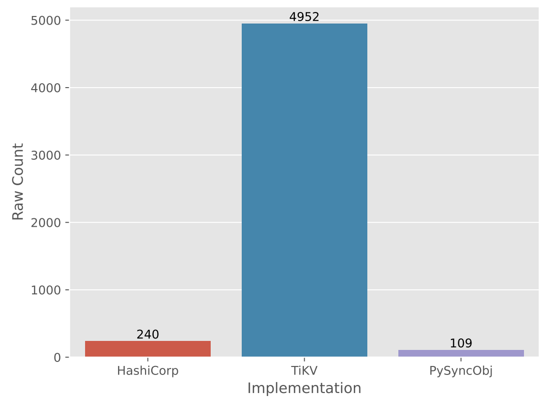

Fig. 3. The raw counts of all open and closed issues per repository.

All experiments were ran on a personal desktop computer running Ubuntu 22.04.3 64-bit with an AMD Ryzen 7 5800x 8-core processor and 64 GiB RAM. Each implementation posed unique challenges regarding how to build the key-value storage system as well as how to obtain the metrics.

1) PySyncObj [9]: This was the easiest to implement as it had a key-value dictionary-like object already made in its library which committed its entries synchronously by default among a cluster of nodes Raft-style. We then built a simple Flask server[6] to interact with the leader from the client. We used `pympler` [10] and Python's built-in `resource` library for metrics[7] which we get from a special HTTP endpoint on the leader server.

2) HashiCorp [11]: We utilized an existing example [12] of a RESTful key-value storage example linked from the main repository[8]. We first forked this to our own version and modified it to give it another HTTP url to retrieve statistics. Log size is calculated manually since it is a simple struct with only two Go-style byte slices, of which we can get the capacity (x and y). The full formula [13] for the log size is:

$$\begin{aligned}\text{size(Log)} &= \text{size(Index)} + \text{size(Term)} \\ &\quad + \text{size(Log Type)} + \text{size(time.Time)} \\ &\quad + \text{size(Data)} + \text{size(Extensions)}\end{aligned} \tag{1}$$

$$\text{size(Log)} = 8 + 8 + 1 + 16 + 5 \text{ word} + x \text{ bytes} + y \text{ bytes} \tag{2}$$

The `pidusage` [14] library was used for process statistics as it directly gets its information from Linux's `/proc/` files.

3) TiKV [15]: This was the most difficult to measure and unfortunately, we unable to get the leader's CPU utility, RSS or log size information. We tried using a service called Prometheus [16] to obtain these statistics, but to no avail. We were able to get the average mesages passed in a similar fashion to the other two implementations. We also got the "pseudo-commit time" by measuring how long the commit took from the client perspective. This added unaccountable overhead as the message first had to be sent to the leader, which, in turn, had to return an acknowledgement. However, we argue this is still a valid and comparable measurement as client server access requires a client-side TiKV RPC stub to be used. This is because clients do not connect directly to TiKV Raft Servers, but rather a broker node which manages the requests sent to all servers. Since this is required to make a commit, it should be part of the total commit times. Unlike the two other options, where choice of framework is entirely on the developer.

D. GitHub Issues Examination

To tackle the developer experience side of our project, we utilized GitHub's CLI [17] to automatically collect all closed and open issues from the three repositories[9]. We got 5,301 issues (Figure 3). From there, we performed a Chi-square ToI on issue status per repository and did basic LDA topic modeling [18] with Gensim [19] on issues given the "bug" label in the PySyncObj and HashiCorp repositories and "type/bug" label in the TiKV repository. For each collection, the model had four passes, a chunk size of 2,000 with ten topics and seed 42. We choose ten topics after some experimentation. We used GenSim to obtain each models CV Coherence metrics.

IV. Evaluation

Name	**Commits**	**Commit Time (s)**	**Avg CPU Util (%)**
PySyncObj	2,000	0.0991±1.46E-5	2.07±0.01
HashiCorp	2,000	0.0001±6.53E-7	10.00±0.11
TiKV	2,000	0.0015±1.43E-4	—

TABLE II
METRICS FROM OUR EXPERIMENTATION RUNS WITH 95% CONFIDENCE INTERVALS. ALL METRICS ARE BASED ON THE LEADER NODE.

Name	**Max RSS (KiB)**	**Log Size (B)**	**Avg Messages**
PySyncObj	1.42E4±219.63	3.5E5±69.92	26550
HashiCorp	1.31±0.08	121.0 ± 0	20240
TiKV	—	—	24610

TABLE III
METRICS FROM OUR EXPERIMENTATION RUNS WITH 95% CONFIDENCE INTERVALS. ALL METRICS ARE BASED ON THE LEADER NODE.

Status / Repo	**PySyncObj**	**HashiCorp**	**TiKV**
Open	0 (1)	6 (3)	203 (205)
Closed	11 (10)	13 (16)	1316 (1314)

TABLE IV
STATUSES FOR ALL REPOSITORIES (AND THEIR EXPECTED VALUES)

[6]https://flask.palletsprojects.com/en/3.0.x/

[7]We also created a special timing library to track multiple function call times using context managers.

[8]The Dockerfile for the servers automatically clones and compiles our fork.

[9]We used the issues from the main HashiCorp repository, not the one which hosts the key-value storage example.

A. RQ1: Memory

HashiCorp's implementation performs significantly better. One reason for this could be Go's statically typed variables and its simple log structure. This means that the variables are explicit and checked at compile-time, which could allow for optimized allocation and de-allocation of memory. This contrasts with Python's dynamic typing, which incurs additional overhead to determine types at runtime. Moreover, it's possible that Go's design philosophies encourage a more disciplined use of memory through values—instead of references—that are required of Python. This would mean that a Go implementation would potentially have less to store on the heap to accomplish the same task.

B. RQ2: Entry Commit Time

HashiCorp also wins for shortest commit time. It has the most consistent commit times as seen by its tiny confidence interval. We attribute this to the fact that PySyncObj's pure Python implementation of Raft left a lot to be desired in speed. While the commit messages and agreement on what to commit could happen quickly, the actual time it takes to commit a key-value pair in a Python object could take much longer. **HashiCorp's log is lean and efficient with the majority of its size coming from two Go slices. TiKV's speed needs to factor in the RPC calls from the client to the broker node and from the broker node to the servers, which adds significant overhead.**

C. RQ3: Messages Passed

When it comes to the number of messages passed, HashiCorp preforms best once again. HashiCorp's advantage in the average messages passed could be attributed to a variety of factors that enhance its communication efficiency. We speculate that the underlying efficiency of HashiCorp's implementation might stem from its use of Go, which is known for high-performance networking and concurrency control. Go has built-in features for concurrent execution that allow for smarter message batching, and extension, potentially more effective intra-cluster communication strategies. In turn, this could reduce the number of messages needed to maintain synchronization across distributed systems, resulting in fewer messages compared to PySyncObj's implementation. While speculative, given the popularity of HashiCorp over PySyncObj combined with the Go's reputation for being suitable for projects involving distributed services leads us to believe that this is likely the core reason.

D. RQ4: GitHub Issues

A Chi-square test of independence was done for repository vs issue status on bug-related issues (Figure IV). This test was chosen because it is non-parametric and makes no assumption of a normal distribution nor relies on there being the same amount of data per categorical split [20]. Setting $\alpha = 0.05$, our hypotheses from this test were:

H_0: There exists no relation between the number of open/closed issues and the repository.

H_A: There exists a relation.

Given a $\chi^2 = 7.06$ and a $p = 0.03 < \alpha$, we can reject H_0. From our results, we can see that PySyncObj and TiKV have closed more bug-related issues than expected while HashiCorp has closed less. **One could interpret this as HashiCorp resolving bugs slower than what should be expected. However, one most also consider the fact that only TiKV and HashiCorp are widely used** (based on GitHub Stars) and that the last commit made to PySyncObj as of April 25th, 2024 was made on October 27th, 2023.

E. Bug Report Topic Modeling

Unfortunately, many of the clusters our LDA model produced do not yield clearly discernible topics. Many topics are "filler" topics which only contain words frequent in the entire dataset, not necessarily words frequent in that topic. There are exceptions though. For example, topic six for HashiCorp is strongly associated with the behavior of the log, its channels and various log-related functions like `appendentries()` and topic 4 seems to be about its fuzzy testing module. Most of TiKV's topics feature the Rust-specific "panic" keyword. Its first topic seems to be related to errors with its built-in region and dynamic log system. Topic five again seems to be related to the log. Finally, PySyncObj—being the smallest of the repositories—had mostly filler topics, but there were a couple with strong in-cluster relevancy for the words because there were only 11 bug-related issues in total. For example, topic two is dedicated to one issue[10]. Manually looking at the clusters, it seems that most issue revolve around consistency and nodes viewing/adding themselves to the cluster.

V. Limitations & Future Work

While we feel the work done on this project is substantial for the semester, there were several limiting factors we faced. Namely, the inability to easily adapt TiKV to our pipeline meant that some metrics are not entirely fair towards it. Due to time and resource constraints, we also could not expand the variety of experiment simulations. Future work could look at other important Raft aspects like leader-loss recovery and log compaction rate. We faced issues figuring out how to measure within Docker Compose and future work could look at expanding our metrics or making process metrics easier to get within a Docker Compose environment. Other avenues of future work include expanding the implementations compared, having a more thorough setup and examination of TiKV and incorporating more Raft implementations into the study. Lastly, our developer experience look is rather basic and more work could be done on examining repository issues and factor in the frequency of posts to give a fairer comparison.

VI. Conclusion

Our project compares three different Raft implementations on their CPU utilization, maximum RSS, average commit time, ending log size and average messages passed as a

[10]https://github.com/bakwc/PySyncObj/issues/116

performance comparison. We found HashiCorp Raft to be the best performing, but one should not count out TiKV as it seems to be a most robust design for large-scale database creation. In addition, we investigate the number of closed and open GitHub issues and found a statistically significant relationship between the status of an issue and the repository; however, these results also need to consider other factors like repository popularity and frequency of issues. We ended our project with a brief examination of common bugs for each implementation via LDA topic modeling. We hope our work can be expanded upon by researchers and would like to see this start a trend of evaluating implementations beyond performance, with more focus on the developer experience.

VII. Contributions

A. Alejandro Ciuba

Alejandro was responsible for managing the experiment pipeline. He also set up the HashiCorp and PySyncObj implementations. Lastly, he was responsible for the GitHub issues extraction, test and LDA topic modeling. The modified `hraftd` used for HashiCorp's key-value storage can be found at his GitHub[11].

B. Winston Osei-Bonsu

Winston was responsible for setting up the TiKV implementation, along with researching potential implementations of Raft. He also explored various methods and strategies for acquiring metrics to evaluate the performance and reliability of these implementations.

References

[1] L. Lamport, "Paxos Made Simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, Dec. 2001, edition: ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001). [Online]. Available: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[2] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014, pp. 305–319.

[3] S. S. Shetty, C. A. Kamhoua, and L. L. Njilla, Eds., *Blockchain for Distributed Systems Security*. Hoboken, New Jersey: IEEE Computer Society; John Wiley Sons, 2019. [Online]. Available: https://books.google.com/books?id=dhaMDwAAQBAJ

[4] M. A. Khan, M. T. Quasim, F. Algarni, and A. Alharthi, Eds., *Decentralised Internet of Things: A Blockchain Perspective*, ser. Studies in Big Data. Cham, Switzerland: Springer Nature Switzerland AG, 2020, vol. 71, corrected publication 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-38677-1

[5] H. Howard and R. Mortier, "Paxos vs Raft: have we reached consensus on distributed consensus?" in *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, ser. PaPoC '20. New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1–9. [Online]. Available: https://dl.acm.org/doi/10.1145/3380787.3393681

[6] T. Nakagawa and N. Hayashibara, "Energy Efficient Raft Consensus Algorithm," in *Advances in Network-Based Information Systems*, L. Barolli, T. Enokido, and M. Takizawa, Eds. Cham: Springer International Publishing, 2018, pp. 719–727.

[7] L. Murphy, T. Alliyu, A. Macvean, M. B. Kery, and B. A. Myers, "Preliminary analysis of REST API style guidelines," *Ann Arbor*, vol. 1001, p. 48109, 2017.

[8] Apr. 2024. [Online]. Available: https://github.com/siemens/edgeshark

[9] F. Ozinov, "Pysyncobj: A library for replicating your python class between multiple servers, based on raft protocol." [Online]. Available: https://github.com/bakwc/PySyncObj

[10] Apr. 2024. [Online]. Available: https://github.com/pympler/pympler

[11] [Online]. Available: https://github.com/hashicorp/raft

[12] P. O'Toole, "otoolep/hraftd," Apr. 2024. [Online]. Available: https://github.com/otoolep/hraftd

[13] J. Hall, "Answer to "how many bytes are in a golang time object"," Jul. 2019. [Online]. Available: https://stackoverflow.com/a/56966008

[14] d. zh, "strucoder/pidusage," Apr. 2024. [Online]. Available: https://github.com/struCoder/pidusage

[15] [Online]. Available: https://tikv.org/

[16] Prometheus, "Prometheus - monitoring system time series database." [Online]. Available: https://prometheus.io/

[17] [Online]. Available: https://cli.github.com/

[18] U. Chauhan and A. Shah, "Topic modeling using latent dirichlet allocation: A survey," *ACM Computing Surveys*, vol. 54, no. 7, pp. 145:1–145:35, Sep. 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3462478

[19] R. Rehurek and P. Sojka, "Gensim–python framework for vector space modelling," *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic*, vol. 3, no. 2, 2011, citation Key: rehurek2011gensim.

[20] M. L. McHugh, "The chi-square test of independence," *Biochemia Medica*, vol. 23, no. 2, p. 143–149, Jun. 2013. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3900058/

[11] https://github.com/AlejandroCiuba/hraftd